Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Charlie Britton

April 2023

Fuzzing With Neural Networks Trained on Instrumented Code in LibAFL

Project supervisor: Dr. Ahmad Atamli
Second examiner: Dr. Graeme Bragg

A project report submitted for the award of
MEng Computer Science with Cyber Security

Abstract

Fuzz testing is a method of software security testing that has historically been implemented with genetic algorithms. This project will build on recent advances in fuzzing with the use of neural networks and gradient descent to create and train a neural network that is trained on testcase inputs to deduce the most important bytes to mutate as part of the gradient value. This is written as a modular library that integrates with LibAFL, the standard modular fuzzing library.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.
- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.
- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

I have acknowledged all sources, and identified any content taken from elsewhere.

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

~~I have not used any resources produced by anyone else.~~

My work uses LibAFL [1] to extract heuristics from code, which are then used in a neural network that is similar in setup to how NEUZZ [2] works. I have made use of the `tch-rs` library and used some of their example code.

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

I did all the work myself, or with my allocated group, and have not helped anyone else.

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/ code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Context

Fuzzing is a very well established method of testing software for vulnerabilities, and compliments other forms of software testing. It is an important research area as bugs may be found in deep code paths which traditional types of testing such as unit testing and integration testing may not exercise.

Fuzzing in general is a very active research area, with many new fuzzing algorithms and programs being published every year, with most of these using a genetic approach to exploring the program paths. This approach randomly mutates an input and relies on successive generations to produce data, which results in program crashes. LibAFL [1], AFL++ [3], libFuzzer [4] and Honggfuzz [5] all make use of this type of fuzzer. Genetic algorithms work well because they are modelled on biological genetic evolution and will gradually improve over many iterations.

Over time, fuzzing has transitioned from simplistic random inputs to programs, as in the very early fuzzers such as [6] to more complicated methods that instrument source code and use heuristics to provide information to the mutation algorithms that are used. This family of fuzzers are commonly known as coverage guided fuzzers. The use of coverage makes the program much more efficiently calculate the inputs required by queueing testcases that may lead to higher edge coverage.

In addition to the now ubiquitous use of genetic algorithms, several authors have proposed the use of neural networks due to their ability to model many tasks accurately and with little technical overhead required. A different approach to this method can be through the use of gradient descent, as is done in NEUZZ [2] [7] and [8], however, this area of research is much less active than other types of fuzzer. In the same way that coverage guided fuzzing allows the genetic algorithm to 'learn', successive iterations of training a neural network can also produce good results as the network trains itself on more and more test data.

The use of a diverse range of these techniques is now recommended by most major fuzzing frameworks, as each technique will converge to a solution in a slightly different way to the others, possibly finding different bugs within the programs.

Within this project, extensive use of both the LibAFL and NEUZZ frameworks are used, with the former being a modern, modular framework that allows the end user to tailor their choice of fuzzer to their individual requirements, and the latter being an implementation of a neural network on an older, yet still high performing fuzzer that was the de facto standard for fuzzing, called AFL [9].

Being a modular framework for fuzzing, LibAFL provides most of the generic parts to a fuzzer that are essential to anything more than a toy implementation. In particular, emphasis is placed on the state and the metadata that is associated with any given testcase, and the actual fuzzing loop, which takes inputs from some sort of queue of mutated inputs from the corpus. The corpus is a set of different inputs to the program and will typically initially consist of samples of valid inputs that the program would usually read, for example valid JPEG or PNG images in a photo manipulation program. The fuzzing loop also has to establish whether a testcase can be deemed interesting enough to be added to the corpus, as there are typically many thousands of inputs tried per second on smaller programs, so storing all testcases would neither be space efficient nor efficient in terms of further mutations.

Combining both neural network and genetic mutation techniques is done in NEUZZ in that the network is initially seeded with testcases that have been generated through genetic mutation. Unfortunately, unless there is already a large corpus of testcase data, a pure neural network fuzzer will likely be unable to efficiently train itself on the corpus, which is an advantage that genetic mutation has because it can be run from very few input seeds.

1.2 Problem Statement

The motivation for this project is that the NEUZZ library is unmaintained and outdated, so is not easy to use for industrial fuzzing without large amounts of setup. Additionally, it has lots of inefficiencies which can be overcome with aspects of their implementation redesigned. State of the art fuzzers employ the LibAFL [1] library with many modular components, and NEUZZ cannot be used as a modular component, so is reimplemented as a library.

The main research question is how the integration of LibAFL and NEUZZ into a single library impacts the speed at which the fuzzer operates and adds new testcase entries to the corpus when run for the same amount of time. In the instance that the integration of these libraries falls short of the expected performance, the research will still be of value as it adds the foundations for which another researcher can tailor the network to increase performance and potentially surpass the current state-of-the-art performance.

1.3 Project Aims

The original project plan was to adapt and improve on NEUZZ and add support for it to the AFL++ library. After consultation with one of the maintainers of AFL++, Dominik Maier, he advised building the project on top of LibAFL due to its flexibility and extensibility in comparison with the lacklustre API that AFL++ had for custom mutators. The project aims to implement gradient descent based on the NEUZZ network structure and mutation algorithm as a library that can be used by different projects wanting to add fuzzing to their test suite, in addition to allowing customization to each aspect of the fuzzer from a simple API.

1.4 Methodology

The overall design of this project is an experiment to see whether reimplementation of the NEUZZ project in a more modern fuzzing system such as LibAFL yields more bug finds in the same runtime as NEUZZ. To achieve this, the plan is to reimplement the NEUZZ neural network, abstracting the network to a new mutator that works in conjunction with existing mutators in LibAFL, then undergoing testing on both a naïve implementation of the fuzzer in LibAFL, and the version of the fuzzer including the neural network mutations.

Evaluation on the LAVA-M [10] and Cyber Grand Challenge [11] datasets as in the original paper was also considered, but due to time constraints in the project, these were not also evaluated.

1.5 Scope and Limitations

The scope of this project involves using LibAFL in tandem with a feed-forward neural network to both generate and mutate the inputs to a fuzzing campaign.

The design of the fuzzer beyond mutation stages that generate new inputs is out of scope for the project, as is testing different neural network designs.

1.6 Structure

The remainder of the report is split into the following chapters: Chapter 2 looks at the theory behind fuzz testing, the state-of-the-art of fuzzing and implementations that make use of neural networks. Chapter 3 looks at the overall design and implementation. Chapter 4 covers the testing methodology and results. Chapter 5 looks at how the implementation compares to NEUZZ and whether it improves on the performance of NEUZZ itself and the experiences learned through the project. Finally, Chapter 6 discusses the results at a higher level and gives a conclusion to the project.

Chapter 2

Literature Review and Current Work

2.1 Fuzzer Types and Tooling

2.1.1 Types of Fuzzer

Traditional fuzzing techniques, such as those discussed in [6] randomly choose inputs to Unix utilities, with the end goal of generating a crash. This initial paper did not reference any malformed input or guiding the input by a specification (e.g., standard generalised markup language or cascading stylesheets). These applications work with predefined grammars with strict rules and sanity checks, such as ensuring that magic bytes exist at the correct locations, or that opening tags are followed by closing tags, which usually lead to a clean exit with an error provided anything even slightly random. This is called black box fuzzing.

Improvements to black-box fuzzing through the use of a grammar made fuzzing a more viable method of testing for more structured languages [12]. AFL++ has recently gained an auto dictionary feature as part of its link time optimisation (LTO) compiler, which is used to extract tokens that may comprise a grammar. This new type of fuzzer was called a grammar-based black-box fuzzer. Improvements to fuzzing followed with white-box fuzzing as in [13] and [14]. This brought access to the source code providing the ability for fuzzing engines to take advantage of symbolic execution and generate the input grammar automatically, based on passes through the source code, which is known as instrumentation.

Production-ready fuzzing tools such as [9] and [4] followed this research and allowed programs to be tested with ease. Following this came grey-box fuzzing, which is where code is instrumented to collect metrics about branch coverage is injected, as in [15] and [16]. Modern fuzzers usually use genetic algorithms and evolution to strategically mutate inputs based on the code coverage and whether the input causes a crash.

2.1.2 Tooling

Instrumentation

To inform the fuzzers about the code coverage for feedback to their mutation engine, instrumentation is the process of adding additional code to binaries to aid in debugging. It is possible to instrument code statically if the source code exists and the instrumentation can be injected at compilation time. Dynamic instrumentation is used where we do not have access to the source code, and so we run a binary in a virtual machine and collect data from the virtual CPU and memory.

Due to this overhead in emulating the CPU, these implementations are usually less performant than the originals, so emphasis on fuzzing with source code and on bare metal hardware to get the most out of the hardware is given.

Instrumentation is typically a stage that is handled by the compiler, which may be a standard such as Clang, or could be a compiler built for instrumenting code specifically, using a standard compiler as its foundation. Examples of these are the compilers that are distributed with the AFL++ library, or compilers that are custom created when using `libafl-cc`.

To instrument a binary, tools such as LLVM [17] and the Clang compiler that the project provides can inject code to monitor the current branch coverage or gather other metrics. AFL++ makes use of the QEMU [18] machine emulator to dynamically instrument binaries where original source code is not available.

Sanitization

Sanitization is the process of ensuring that bugs that might not always cause a crash (e.g., reading or writing after a `free` of memory) will crash the program or inform the fuzzer that there is a potential bug discovered in the code. Sanitizers will make the program a lot more 'sensitive' to any bugs and ensure that a nondeterministic fault is repeatable and deterministic. We can increase crash amplification further to find logic bugs such as successful authentication where we should not be able to authenticate. This demonstrates a wider use of fuzzing to verify correctness of security logic in addition to finding bugs within programs that may lead to other security vulnerabilities.

Snapshotting for In-Memory Fuzzing

As fuzzing is a very continuous process and the goal is to maximise the throughput of the fuzzer, many optimizations exist. Snapshot fuzzing is the process of starting a program from a predefined snapshot to increase coverage from a specific branch. Combined with in-memory fuzzing, where the instrumentation stores a copy of the program after initialisation to be copied for each instance of the fuzzer, this yields 10-20x increases compared to starting from disk every time, as is used in AFL++. Snapshotting is also useful if the user wishes to test a specific part of the system, but there are a lot of setup operations required to get the system into an initial state with which to test.

Corpus Minimization

When fuzzing, we want to typically use the smallest inputs possible that will still generate the same crashes or outputs. Moving less data around is better for testing, and having the fuzzer mutate fewer bytes makes the mutation more efficient.

2.2 Conceptual Fuzzer Elements

The most important parts of any given fuzzers are the state, the execution (or scheduling) engine and the mutation engine.

2.2.1 State

The state tracks all the metadata for each testcase that is run, including the fitness values and any heuristics used to inform the scheduling engine which inputs to schedule next to improve the fuzzing loop. Traditionally, in fuzzers such as [9], the state is an area of memory that would store the current bitmap of edges that have already been visited by the fuzzer in at least one testcase. The bitmaps for each individual testcase would be stored on disk. In LibAFL, the state holds the random number generator, input corpus, solutions corpus and the feedback and objective heuristics.

2.2.2 Mutation Engine

This part of the fuzzer is responsible for manipulating data from a corpus with random mutations. These mutations can be very simple, such as a bit flip or multiple bit flips within the input string (e.g., changing a byte `0100 0010` to `0101 0010` or applying some pattern: `0100 0010` XOR'd with `1010 1010`, to `1110 1000`). This would not be an uncommon occurrence in real world programs, as cosmic microwave background radiation could easily change bits, or an unreliable packet in a networking application may have some transmission error when decoded. Ensuring that applications are robust to both malicious and non-malicious faults that occur is essential.

In addition to simple bit and byte-flips, the mutation engine can also add and subtract integers, add in well known 'problem' bytes, such as values that are at the edge of data boundaries, such as $2^n - 1$ for signed integers or sequences of bytes that fall on data boundaries such as at the edge of words or pages in RAM. Another mutation which is useful is the random splicing of 2 strings at a random point within the data, modelled much after mutations in genealogy. In AFL, this is known as the havoc mutation and is used when other strategies begin to stall.

In a more advanced fuzzer, the mutation engine will track the performance of the various types of mutation within the state and choose the preferred type based on the observed performance.

2.2.3 Execution Engine

This part of the fuzzer is responsible for choosing which of the mutated strings in the queue to send to the program under test. The scheduler runs through the queue of test inputs which the mutation engine has created and either forks the program, or in more recent versions of AFL++, leverages persistent mode which stores an initialized program in memory to be copied across for each target that is being tested.

Once the fuzzer begins to stall, where the number of new edges found begins to plateau for example, the scheduling engine can instruct the mutation engine to create some more interesting (but potentially less fruitful) cases, such as by using the havoc mode to splice together random strings. At some point, once all of these modes have been exhausted, the fuzzer will reach a final plateau as it fails to find many more edges, either because they are hidden behind guards or they require checksums.

Once the queue is empty, the mutator will promote a subset of the samples from those that caused the test harness to crash to the next round, in addition to the original inputs that were given to the fuzzer. A simple mutation engine may simply run through the inputs in a FIFO queue, and LibAFL provides the extensibility and flexibility for the end user to experiment at a high level with their choice of queue.

2.3 Current Research

2.3.1 Symbolic Execution

Many of the optimizations to fuzzing are through the reduction of symbolic execution of a program. Although symbolic execution provides excellent data on how differing inputs affect the execution path of the program, it is relatively slow to execute. T-Fuzz [19] removes magic numbers and checksums from a program being instrumented when the fuzzer can no longer find new execution paths, thus removing the need to generate the correct header or checksums. Similar to T-Fuzz, Redqueen [20] works around the issues with bypassing checksums and magic headers through a lightweight approximation of taint tracking.

2.3.2 Scheduling Optimizations

MOpt [21] presents a novel method of mutation scheduling in the fuzzer by changing the priorities of different mutation operators based on the empirical evidence it sees from each type of mutation on a specific type of target. AFL++ recommends using both Redqueen and MOpt when fuzzing, as they both provide good speed improvements and work as native plugins to the underlying fuzzer.

2.3.3 Neural Networks

Although fuzzing with gradient descent and neural networks is not new, their use in industrial tools is much lower. Angora [7] uses gradient descent to solve a constraint satisfaction problem, which it models based on overall code coverage. The other main research paper in this area is NEUZZ [2], which makes use of a feed forward neural network to automatically generate test inputs as a replacement for the mutation engine. This is the paper that my project is centred around.

LibAFL [1] is a new fuzzing library that is built in the Rust programming language and is intended as an eventual replacement for their earlier work on AFL++ [3]. Instead of using a shared memory section as AFL++ did, the fuzzer is built into its own binary that communicates with an external program to store useful information about what is being run. Instrumentation of the code is also much more extensible with this new version of their fuzzer.

2.4 LibAFL and NEUZZ Specifics

2.4.1 LibAFL

Basis from AFL++ Implementation

AFL++ is a fork of the original AFL fuzzer. AFL++ works in much the same was as AFL did, but has extensibility options for integrating new plugins. In the initial design of the fuzzer, the author designed the system such that it never assumed anything about the program under test and that a wide range of operations may work well on the program.

Storing Coverage AFL++ makes use of a coverage map, which is a shared area of memory between all instrumented binaries that are tested and upon hitting an edge, the bitmap is updated to show that the edge has been covered. If the edge has already been covered by another test case, then there is nothing interesting to learn for that edge case. In AFL, this map was initially chosen to be 64 kB, such that it could fit in the level two cache of the CPU and be really efficiently loaded to and written to from each instance of the fuzzer that runs.

In addition to storing the branch counts that have been hit, the number of hits to branches are stored, so that even if there are no new branches hit but the number of branches a program visits varies distinctly, then this is used by the fitness function to increase the likelihood that the generated input will be promoted to the next round of inputs for the mutator.

Library Implementation

The LibAFL library was designed and implemented by the maintainers of the AFL++ library, with the goal that the library is a much more modular version of AFL++, with the idea that fuzz testers can bring their own parts of the fuzzer to the binary or source that they want to test. LibAFL provides each of the core parts of the fuzzers as submodules, which end users can implement and pass to other parts of the fuzzer.

For those who wish to extend LibAFL and add their own functionality, as in the case of this project, a comprehensive API is provided, with the use of Rust's generics system of trait bounds. Functions that use APIs provided by the modules that exist in LibAFL will also need to implement those traits that the generic code they borrow from or require the provided types to be of a specific generic type to be used.

This system is a much more advanced system than the API exposed by AFL++, and as such, although the library is newer, it made sense to use it for this project.

2.4.2 NEUZZ Implementation

NEUZZ views fuzzing as an optimization problem, where the aim is to use gradient descent to find test inputs that will cause the program to crash. Due to the discontinuity of a function which returns 1 for all times a program causes a crash and 0 for every other input, gradient descent will not work well because the gradient at most points will be 0, except for those either side of a crash.

Therefore, NEUZZ initially smooths the inputs to the model. This is done with Gaussian and Sigmoid functions. The inputs to the neural network are the input strings that will eventually be fuzzed by the execution engine. The output of the neural network is an approximation of the bitmap (the 64 kB shared memory region).

NEUZZ tries to predict the control flow of a program given a specific input and uses that to prioritize those inputs which are likely to explore edges that have not been taken by other inputs as yet. The algorithm used to generate new mutation bytes is based on those input bytes with the highest gradients as established by the network inputs.

Chapter 3

Design and Implementation

The main contribution area of this project is adding a new library module to the LibAFL fuzzer which provides mutated inputs to the fuzzer, working in tandem with the normal mutation engine that LibAFL uses.

The main areas where we can improve on NEUZZ are automating the initial gathering of a corpus with which to train the network on; providing the neural network capabilities as a native library as part of AFL as opposed to a discrete fuzzer; and streamlining of the processing of data for the network whilst running the fuzzer. The codebase for NEUZZ was also disjoint and modified the AFL source code, which due to its implementation within the fuzzer itself made it incompatible as a module with regular AFL and has thus been unmaintained and largely unused. The author hopes to submit a pull request with these library changes, so that the community can make use of the library that has been developed.

3.1 Overall Design

LibAFL is designed as a modular replacement for traditional fuzzers such as AFL++. As such, this thesis implements modules that are compatible with LibAFL. The system can utilise many existing aspects of the LibAFL library, saving time reimplementing basic functionality. This thesis adds a new mutator class which makes use of the neural network. It can make use of any kind of corpus, be it in memory, on disk or a combination of both. The preprocessing step of the network takes the current corpus and will automatically convert it into a form that the network can understand.

As Rust is a very generic language and makes use of traits and bounds to create an extensible system, we simply have to implement a `NeuralStage` which works with other types of mutation.

Initially, the idea was to create a new scheduler that will take inputs from a source until that source is empty, at which point it falls back to the naïve fuzzing mode. Due to limitations within LibAFL, the actual implementation adds a `NeuralStage`, which will fuzz on the neural mutations and handle the network training and evaluation.

The main focus area of the library is within the `stage` submodule that has been written, which sends the data from the trained network to the `mutator` submodule.

There is also a `preprocessing` module, which handles the conversion of data from the testcase metadata and input bytes into an implementation that the `network` submodule can handle. Other submodules have less function than the modules listed above, but are important for the overall functionality.

3.2 Data Generation and Preprocessing

Before the neural network can be effectively trained, the program needs an abundance of data to train the network on. As such, the system is designed to automatically run the fuzzer in a naïve mode until the corpus is large enough to train the network.

3.2.1 Automation of Initial Test Dataset Generation

In NEUZZ, the process of gathering the initial data in which to train the network with was a largely manual process, with the user running AFL for an hour to seed the neural network and provide it with a meaningful amount of data. In this implementation, the initial gathering of the dataset is handled by the library and runs the fuzzer without the neural network until there are enough testcases with map data that can train the network. This is a configurable parameter that the end user can alter. This is good because it allows for experimentation with different parameters and reducing the training time when running unit tests.

3.2.2 Converting Data for the Network

Through the abstraction and modularity that LibAFL offers, the fuzzer stores each `Testcase` possibly in memory or on disk, with the metadata associated with that testcase also being stored. An `UnsplitDataset` and `Dataset` structure is created to hold both the input and output tensors, which hold the data for training the network. As Rust lacks good library support for tensor-like data structures and processing unlike Python (e.g., the `sklearn` library), wrapper functions to manipulate and split the data have to be created.

There is a `preprocessing` module which handles the processing of the corpus, by filtering to those testcases that have the appropriate metadata (run with a mapping feedback at least once), then converting the data into the tensors that are relevant to the input and outputs of the network. This module is designed to handle testcases from LibAFL and works with the API that the fuzzer exposes to read through and process the testcases.

3.2.3 Normalization

As part of the preprocessing in NEUZZ, the data was converted from binary bytes represented as floats in the range of $[0, 255]$ to bytes represented as floats in the range $[0, 1]$. Whilst this is not a traditional normalization step as the data is not clamped based on a probability distribution, this is an input step to the network and is implemented the same in my library.

3.2.4 Legacy Inputs for Testing

In addition to the preprocessing module, there is a further `legacy` module, where there is the option to test the network on a dataset with edge coverage as extracted by `afl-showmap`. This is a useful step to validate that my network operates similarly to the NEUZZ network and to allow testing without running the fuzzer to allow the network to be debugged effectively.

3.3 Processing Data for Model in Memory

The datasets used to train the network are generated at runtime through the many fuzzing testcases that the fuzzer runs. This implementation adds a step to process the data that the network will be trained on whilst the fuzzer is running. This should have some speed benefits, as in NEUZZ, the fuzzer would report when it was running low on testcases to fuzz and start training the next iteration for the network as a batch operation. With this approach, the data is processed as it runs through the fuzzer and the next iteration of the network is trained whilst the current generation runs.

Preprocessing the data in memory will save lots of expensive disk reads and writes, as the input will already be in memory. The fuzzer will write both the test input to disk and then the vectorized version for the neural network to train on.

3.4 Neural Network

The project makes use of a feedforward neural network, where at a high level, the input features are binary testcase inputs and the output is the expected coverage bitmap. This network is experimentally implemented with the `tch-rs` crate, which is a relatively unstable beta that wraps around the PyTorch C++ API.

This is therefore a type of supervised learning, as the network is trained given known inputs and outputs. The network is a function f that takes inputs $I = \{\texttt{0x00}, \texttt{0x01}, \ldots, \texttt{0xFF}\}^m$, where m is the maximum permissible input size and provides the expected coverage bitmap $O = \{0, 1\}^n$, where n is the bitmap size (nominally 65 536). This function is the same as that used in NEUZZ and is given in (3.1). The overall design of the network layers is given in Figure 3.1.

In this implementation, there is only the dense layer, although the library includes support for the user to add as many hidden layers as they would like, without having to change the source code for the network themselves.

$$f : \{\texttt{0x00}, \texttt{0x01}, \ldots, \texttt{0xFF}\}^m \to \{0, 1\}^n \tag{3.1}$$

3.4.1 Hyperparameters

As the contribution area of this project is porting the NEUZZ fuzzer over to a modular library for use within LibAFL, the way the network is trained is the same as in NEUZZ. Each hyperparameter, its description and the value for said parameter are described in the following sections.

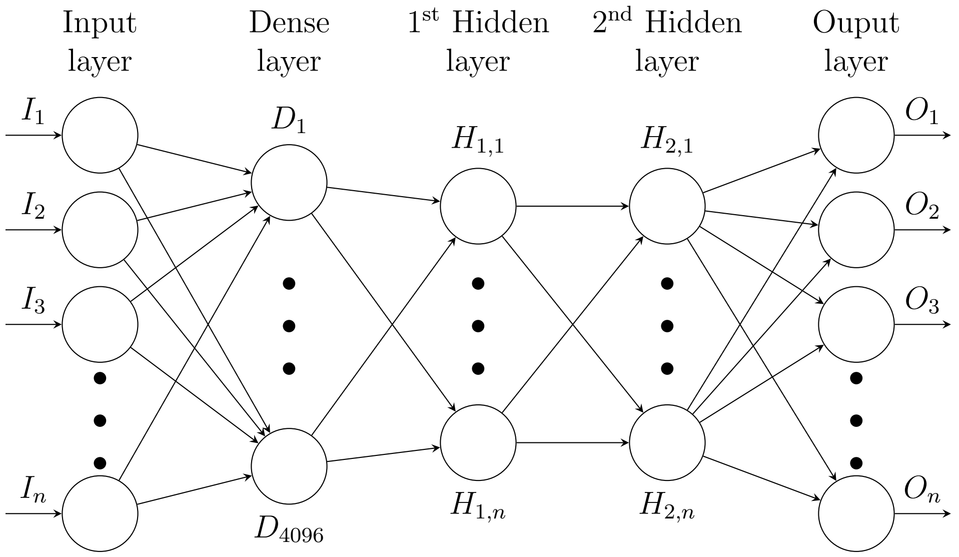


Figure 3.1: Proposed Design of the Feedforward Neural Network

Optimizer and Learning Rate

The Adam optimizer [22] was used in NEUZZ and is a well established optimizer that produces good results on most models when fuzzing. The learning rate for the network was 1×10^{-4} as in NEUZZ.

Dimensions

The input dimensions to the network were either the maximum size of the corpus input bytes or 10 kB as in NEUZZ and this ensured that there were not too many parameters to train within the network. The output size of the network was clamped at a maximum of 2000 bits, or the reduced bitmap size as taken from all the inputs, whichever was smaller. In between the inputs and outputs, there is one dense layer that reduces the input dimensions to $2^{12} = 4096$ and connects to the output layer. Both the input and output layers are provided to the network as `f32` values, with the input bytes being converted to their float representations and divided by 255.

Loss Function

The loss function that NEUZZ chose was Binary Cross Entropy (BCE). This loss function is good for modelling data where the data can belong to one of two classes. If the data contains more than 2 classes, as is the case with bitmaps (each bit is a class), then BCE will not work well and a different loss function capable of multi-label classification needs to be used [23]. The choice of BCE to train the network is not the best loss function to use, but as my implementation does not change the network model at all as that is out of scope, this was what was implemented.

3.4.2 Network Training

This network will need to be trained by providing a set of input data and the expected outputs, incrementally adjusting the weight and bias vectors used in each neuron. The `tch-rs` crate does this automatically and has a similar API to that of Keras, allowing the end user to focus on the higher level implementation and not the network specifics.

3.4.3 Other Network Considerations

During the research phase of the project, exploring using other types of neural network for the project aims was considered, including a directed graph convolutional neural network (DGCNN) as this is the underlying structure that the bitmap represents. It was decided against using this as the bitmap is already a good representation of the coverage and control flow in a compact, efficient network, but the simple feed-forward neural network that NEUZZ implemented was more efficient. Initially, it was planned that the inputs to this neural network would have been the control flow graphs of a wide variety of programs and the outputs would be a byte array that the network would have thought would crash the program.

Another option considered was attempting to get the neural network to predict an input that would produce a specific bitmap, but this proved difficult as the search space would be so large, in comparison to the system that NEUZZ implemented which reduced approximately 10 kB worth of data to a simple bitmap of 2000 bits in the worst case.

Furthermore, it was decided that modifying the network in any way other than to replicate it in Rust would be out of scope for the project. The design for the NEUZZ network seems to work well experimentally and as such, changing it would likely be poor use of development time.

3.4.4 NEUZZ Implementation

NEUZZ forked the original AFL application and added some custom code to the source so that it would use samples generated by the neural network and request more from the model as and when the queue started to run low. It would create a socket that the Python neural network application would bind to and once connected, would train the model on the initial shared set of samples from the set that was provided, whilst AFL continued to fuzz normally.

When generating the gradient, the network pulls in samples from both the original set and any new crashes that AFL has found. Once it has a training dataset, the model is created as a sequential model, which takes the maximum filesize of all the files in the directory and reduces them to 4096 edges. It then uses the ReLU activation function $f(x) = max(0, x)$ to feed into the next layer, which is the same size as the current bitmap in AFL.

3.4.5 Pre-Existing Datasets

The network runs on the collected byte inputs and the expected bitmaps for a target. Where bitmaps are created, they are specific to that specific binary and cannot be applied to a broader set of applications. This contrasts with most image recognition or text detection models, which are trained by a large computer and then the weights and biases that have been trained are distributed as a pre-trained model. The reason we cannot pre-train on a large dataset is because the network is trained on the predicted bitmap (control flow) of the program and this bitmap is highly specific to each library as different libraries have different methods of calling functions.

An initial project idea did consider training a network on a bug class to classify the input data into different bugs that it could have, but this did not align with the idea to improve on fuzzing.

3.4.6 Library Choice

The network was implemented with `tch-rs`, which is a Rust wrapper around the C++ bindings for the PyTorch API. Initially, it was planned to use TensorFlow as it had Rust support, but the documentation around the `tch-rs` in addition to the higher number of stars on GitHub showing that it was more widely used. There was also another library called Linfa [24], that provided native Rust bindings for machine learning but had less support and was not as well known as PyTorch, with very little online beyond their official documentation.

This library worked well initially, but as some of the more advanced features and functions were needed, the library was found to be poorly documented. If it were not for the rest of the project also being written in Rust, falling back to a Python implementation with the Keras API or using the Linfa library may have been a better route.

3.4.7 User Configuration Parameters

The network design allowed for a configurable `NetworkOptions` struct of data which allows the user to override the default parameters, such as batch size, number of epochs and dataset split ratio. The user can also tune the dense layering. This helps with the extensibility and modularity of the network to allow a layperson or a future researcher to tune the design.

3.5 Mutation Generation

The gradients from the network are used to inform the mutation algorithm which bytes are the most significant. The input bytes with the highest gradient values are likely to be those with the most effect on the program branch flow. The algorithm, as given in the NEUZZ paper is shown in Algorithm 1.

These mutations are generated based on the network, and the number of mutations can be tuned by tuning both the number of iterations i and the number of bytes k that we want to mutate. NEUZZ found best results with $i = 7$ and $i = 10$ so $i = 7$ was chosen by default, with the ability for the user to configure it differently, should they choose. The total number of mutations varies dependent on k and i but for $i = 7$ and $k = 100$, the total number of inputs would be close to a million.

Algorithm 1 Gradient-guided mutation

```
Input:
    $seed$                                                        ▷ initial seed
    $iter$                                                  ▷ number of iterations
    $k$                     ▷ parameter for picking top-k critical bytes for mutation
    $g$                                                ▷ computed gradient of seed
Output:
    Mutations generated at all significant locations.
 1: for $i = 1$ to $iter$ do
 2:     $locations \leftarrow top(g, k_i)$
 3:     for $m = 1$ to 255 do
 4:         for $loc \in locations$ do
 5:             $v \leftarrow seed[loc] + m * sign(g[loc])$
 6:             $v \leftarrow clip(v, 0, 255)$      ▷ make sure that byte is in byte range (0-255)
 7:             $gen\_mutate(seed, loc, v)$
 8:         for $loc \in locations$ do
 9:             $v \leftarrow seed[loc] - m * sign(g[loc])$
10:             $v \leftarrow clip(v, 0, 255)$
11:             $gen\_mutate(seed, loc, v)$
```

3.6 Module Development

3.6.1 Stage

This is implemented as a workaround to the type system issues within LibAFL, as the user cannot switch mutators and have fallback mutations. This stage runs its own fuzzing loop on each input that is extracted from the trained neural network, in addition to managing the training of successive iterations of the network.

3.6.2 Network

The main function from this is the train function, which takes a dataset and returns a tensor of the trained network. This can then be sent to the mutator submodule to process and extract the bytes that the network assumes are the most important for the mutator to use.

3.6.3 Mutator

This is a reimplementation of the mutation algorithm that is in NEUZZ, working based on the gradients from the fuzzer and mutating the bytes that the network believes to be most influential through its training.

3.6.4 Preprocessing

This takes the data from the fuzzer and processes it into the form that the network trains on. This is an important module to get right as otherwise the network will not train properly and so emphasis was placed on unit testing each aspect of this.

3.7 Other Optimizations

Unlike NEUZZ, which waits for the network to train once it runs out of testcases to fuzz, this implementation only does the initial retrieval of testcases from the corpus into the dataset in the main thread, before spawning another thread which it uses to train the network on the data. `tch-rs` is optimised for multithreaded use, so will make use of all CPU or GPU compute available to train the network.

By leveraging Rust's thread mechanism, the fuzzer can continue to run whilst the network is being trained. Should the network run out of inputs in the queue whilst the network is training, then the stage will essentially be skipped and allow the other aspects of the fuzzer to run instead.

3.8 Testing Tools

The NEUZZ repository provided precompiled versions of the tools being fuzzed through the `afl-cc` and `afl-c++` compilers. In this design, the `cargo-make` package was used to allow the end user to download and compile all the fuzzing targets and change the compiler that they use in the future. This means that a future researcher can evaluate the programs on an updated fuzzer. With these tools compiled using the LTO compiler, for example, there are fewer conflicts in the edge coverage of the bitmap with negligible performance cost as the bitmap is minimized before processing and passing to the neural network. If a future compiler alters this bitmap further, for example to instrument other statistics in addition to edge coverage, then the compilation of testing programs can be done with minimal effort.

As discussed above, the fuzzer's initial data generation and gathering is automated, with user defined limits to the amount of inputs to initially run the fuzzer with to seed the network, and this saves time gathering test data. The `preprocessing` module does allow the user to import testcases initially and seed the network as in NEUZZ, which is used to validate the model when testing. The testcases are also stored in memory or cached on disk, as opposed to working on disk as NEUZZ did.

3.8.1 Automating Testing

To ensure repeatability of the results from the project, the use of the `cargo-make` library to automatically download and compile the correct versions of the test data was employed. As part of the Cargo workspace for the project, this downloaded and installed all the dependency binaries that the project would be fuzzing on, building them with differing instrumentation passes and then automatically passing them to the fuzzers being tested. The driver program starts each instance and runs it for a predefined time on a CPU core, terminating after the program has run for the specified CPU time.

3.8.2 Instrumenting Binaries

To minimize development effort adding stubs to binaries, they were compiled using regular tooling and makefiles, with one modification to initialize the AFL shared memory area as they start. A more performant fuzzer could be attained if the binaries were instrumented as libraries and executed in memory, with many possible improvements.

3.8.3 Compilation with LTO

Initially, it was thought there might be a problem with the compilation of binaries using link time optimization, although through experimentation this was not an issue in practise.

3.9 Novelty

The design of the system is a modular set of libraries such that a future researcher or fuzz tester can choose to implement it as part of their fuzzing campaign. I believe the novelty lies in that the processing can be done entirely in memory and that the neural network mutator can be accessed as part of the LibAFL library.

By compartmentalizing the training for feedbacks and the mutation stages into separate modules, future researchers will have the option to build and expand on each of the modules separately.

The tools and technologies used in this implementation are shown in Table 3.1.

3.9.1 Using LibAFL vs. AFL++

The motivation for this project was initially focused on the AFL++ repository, with the maintainers of the project looking for somebody to port the NEUZZ library into something compatible with the project. Following an email by the maintainer who had offered a mentorship for this project, where they recommended that the project extended on LibAFL as opposed to AFL++, the author switched to using Rust for my implementation.

3.10 LibAFL Parts

As LibAFL is a library of reusable parts that make up a fuzzer, each part requires careful consideration whether it is suitable for the type of fuzzer that we want to create and the domain of that fuzzer.

As this implementation aims to port NEUZZ over to a more modern architecture and improve performance yet still be comparable for testing purposes, the fuzzer is implemented similarly to the demonstration code for the `forkserver_simple` fuzzer that LibAFL provides. Therefore, the fuzzer will take a binary that has been compiled with `afl-cc-lto` as an input, in addition to the corpus to initially fuzz on and start a fork server which allows the binary to be fuzzed.

Aspect	Tool Chosen	Function	Justification
Source Control	Git & GitLab	Source control management for the code in the project.	Git is the industry standard for SCM in code, and I use GitLab to store all of my work. Saving my work at key checkpoints can be done through commits and my project can be distributed to others who have git installed on their systems, provided the source code is made public.
Programming Languages	Rust, Python, C	Languages for code in the project.	LibAFL is written in Rust, and as such, extending this library also requires code to be written in Rust. It is a performant language and is safe, unlike C. Being a compiled language makes it easy to distribute to targets and static binaries allow there to be no prerequisites on the host machine. Python was used to establish how NEUZZ worked and to ensure that the unit testing within Rust was correct. C was used to add small stubs to the programs before compilation and to write a couple of toy examples to verify the correctness of the library.
Unit Testing	Rust Built In	Testing to verify correctness of different aspects of the program.	Rust's built-in testing ecosystem with assertions provide good basic testing functionality for each aspect of the program. This is especially useful to verify that the behaviour of the program matches the expected network behaviour, and with manual seeding of the randomness for the network, deterministic verification of loss and accuracy can be computed.
Machine Learning Framework	`tch-rs`	This is the framework that allows me to write the ML models.	This was originally TensorFlow, but as the Rust bindings for TF were less mature than the bindings for `tch-rs`, the latter was chosen. In a future project, more extensive experimentation with other frameworks would be good to ensure that the library is well documented and has good support, as I established that `tch-rs` was poorly documented once beyond the basic aspects.
Project Management	Gantt Chart	High-level overview of the project tasks and their status.	Shows a waterfall of the dependencies on tasks and overlays a calendar, so I can see when other time pressures might exist and allows me to see how on track I am with task completion.

Table 3.1: Tools Used and their Justifications

Chapter 4

Testing and Evaluation

4.1 Testing Methodology

As this work is largely based on the NEUZZ fuzzer, the testing and evaluation methodology is similar to the original library. My testing is run between LibAFL and my implementation as the aim is to establish whether a reimplementation of NEUZZ yields better results on LibAFL.

NEUZZ was evaluated against many real world programs, and my testing aims to match the versions tested in NEUZZ as closely as possible. The exact real world programs used, and their versions are given in Table 4.1.

Class	Binary	Version
`binutils`	`nm -C`	`2.30`
	`objdump -D`	`2.30`
	`size`	`2.30`
JPEG	`libjpeg`	`9c`
PDF	`mupdf`	`1.12.0`

Table 4.1: Real world programs, their uses, and their versions

The programs chosen for testing are largely structured languages and media encodings, including an implementation of JPEG and PDF files. Due to hardware limitations, my GPU is unable to run the machine learning models and so all training was done on the CPU, likely making it slower than it would otherwise be. Nevertheless, as the evaluation is all run on the CPU for all types of fuzzers, a comparison can still be drawn.

4.2 Hardware and Runtimes

Each of the real world programs ran for 1 hour on an individual CPU core. Each program was run on both LibAFL and my implementation. Statistics such as the corpus size, number of crashes, total executions and rolling executions per second statistics were tracked.

4.3 Unit Testing

Unit testing was heavily employed to verify program correctness and ensure that the expected values from NEUZZ were also seen in my program. Each submodule has extensive tests that can be run as regression tests to ensure that adding new functionality to the fuzzer does not break existing parts.

4.4 Network Implementation

The network was ported from Keras and TensorFlow to `tch-rs`, using the same semantic concepts and hyperparameters. When the accuracy of the network was found to be lower than expected, further unit testing was employed to ensure that the inputs to my network were identical. Further sanity checks were added to ensure that the outputs were of the same format, with all data points in the correct ranges and of the same data types. Experimentation with the number of dense nodes, hidden layers, optimizers, and loss functions did not improve the performance significantly, suggesting that the issue was with the library used.

4.5 Testing with GPUs

A substantial amount of time was spent managing the environment for the project, as due to the library being so new and the lack of experience with the C build toolchain and the rust build toolchain, there were many approaches tried.

The initial environment did not work very well as the project made lots of assumptions about the underlying location of lots of tools and a lot of time was spent debugging these. Eventually, a virtual machine was used running Ubuntu, which was able to run LibAFL and NEUZZ, although unfortunately without the use of GPUs.

Although the availability of the Yann server was known, a significant amount of time would have had to be dedicated to porting the implementation to run there, so the final tests are a comparison of the CPU processed speeds and corpus.

4.6 Comparison of Raw Executions per Second

This test shows the throughput of the fuzzer, which is one of the many metrics that fuzzers rely on. This is compared between vanilla LibAFL and my implementation. This can be seen in Table 4.2.

The executions per second of each is established as the total number of executions over the runtime, and is not calculated from the statistics that are logged to the console, as these provide moving windows of time.

4.7 Comparison of Corpus Size

This part of testing shows the corpus size after fuzzing for 1 hour with vanilla LibAFL and my implementation. This can be seen in Table 4.3.

Program	Executions per Second		Neural Speed as % of Vanilla
	Vanilla LibAFL	My Implementation	
`nm -C`	724	724	100%
`objdump -D`	40	58	145%
`size`	1071	1071	100%
`libjpeg`	785	774	99%
`mupdf`	1708	1708	100%
		Average	**108%**

Table 4.2: Comparison of execution speed between LibAFL and my implementation.

Program	Corpus Size		Neural Corpus as % of Vanilla
	Vanilla LibAFL	My Implementation	
`nm -C`	1016	1181	117%
`objdump -D`	1078	1008	93%
`size`	811	845	104%
`libjpeg`	1813	1596	88%
`mupdf`[1]	1	1	–
		Average	**100.5%**

Table 4.3: Comparison of corpus size after 1 hour of fuzzing on each.

4.8 Testing Script

To ensure fairness, the `forkserver_simple` example from the LibAFL repository was copied and modified to add the neural stage. A bash script was then created to automatically run both LibAFL and my implementation as a job on the system, force exiting after 1 hour. Postprocessing was then done on the results to yield the executions per second and corpus size results for the fuzzers.

[1]The initial corpus was not imported correctly, so this is discounted from the average.

Chapter 5

Discussion

5.1 Overall Contribution

Whilst the project did not reach my initial goals of having a network similar to NEUZZ that performs better than NEUZZ itself, there is still value in that this work provides a solid foundation for others to experiment with fuzzing using neural networks. Any user of LibAFL can make use of this library as a scheduler to generate mutations for their fuzzer, and they can build on the foundational steps which have been introduced, such as automated preprocessing and generational steps for their data.

This contribution also handles the corpus through the LibAFL API and as such requires many fewer reads and writes to the disk, which contributes to slower performance of the fuzzer. Had a more mature machine learning library with good community support and documentation been used, or had the author taken the machine learning modules, it is believed that these issues would be fixed with the model and get it to a state similar to NEUZZ.

5.2 Limitations

Somebody with more experience with neural networks and machine learning with the `tch-rs` crate may be able to debug the network more effectively, but from further research, it was established that the `tch-rs` library is not very widely used and most machine learning is still done with Python.

Although this paper would have liked to have changed the implementation to possibly use a different machine learning library, at the point it was established that the implementation was not performing as expected, there was not enough development time left on the project with which to refactor the code and retest this.

A future researcher may wish to look into changing the loss function or many other parameters of the network, as what was implemented in NEUZZ is not necessarily the best option and categorical cross entropy may be a better choice for an updated network. A future thesis could examine this further.

5.3 Purpose as a Reference Implementation

Although this implementation of the NEUZZ paper and algorithm does not perform as fast as NEUZZ or LibAFL running natively, it is believed that this implementation serves as a good reference for a potential future researcher to refactor, benchmark and bring it up to the expected standards for an industrial fuzzer. There are many aspects and edge cases within the code that have been left unstable, but as a prototype implementation, the code performs well on the majority of testcases in the limited scope.

5.4 Comments on the Design of LibAFL

It is believed that through implementation to be compatible with LibAFL, the semantic meanings used to define mutators, schedulers and stages has not been followed, and their reference implementation of the mutator stage is run within the stage, as opposed to the mutator adding inputs to the queue.

When implementing the NEUZZ algorithm, it was initially planned to make a custom scheduler, that schedules inputs from the neural mutator when available and fell back to the other naïve fuzzing mode when inputs are not available from the fuzzer.

Through the traits that LibAFL defines, and their implementation of the `StdMutationalStage` stage, it was established that the scheduling of each individual mutation was manually handled by the stage, as opposed to the scheduler algorithm.

The documentation of the library is currently sub-par, with in depth analysis of source code to establish what each part of the fuzzer does.

5.5 Time Management and Planning

Initially, the time management for the project worked well, and establishing the project and doing basic background research was completed in a timely manner. The progress report was well researched, and a good literature review had been done, but as the implementation phase of the project began and other deadlines and examinations approached, the project became less of a priority.

From the outset, not much emphasis had been placed on the Gantt chart and keeping to the schedule. As the implementation began, many issues were found which had not been accounted for in the research or planning phase of the project, such as the lack of GPU support for TensorFlow on AMD.

The last minute change in the chosen library just before the progress report was due also setback the research phase of the project, as at the time, LibAFL had only just been released and as such there was very little work on the fuzzer that was included in my original research.

Although the Gantt chart had detailed plans for the development of the fuzzer (as given in Appendix A), these did not adequately account for development issues, with regard to the machine learning library that was being used, my lack of experience with machine learning and the complexity of the LibAFL library.

The testing for the project was thus delayed and as such the library was unable to be tested as comprehensively.

5.6 Ethical Considerations

Whilst fuzzing tools could be used by any malevolent person with the intent to cause harm and damage through the exploitation of a bug in a program, the widespread popularity within industry and the open source community, in addition to responsible disclosure programs balances the harm that a malicious actor could do.

The author does not believe that the research provides anything that could be used to the detriment of others any more than any other fuzzer does, and hopes that my implementation will be built on or used by the community in the future to improve the state-of-the-art.

5.7 Practical Applications

As explored earlier in the literature review, having many fuzzers that operate in different ways to one another with different techniques for maximizing the edge coverage of programs is useful, as the paths discovered are often different to each other.

The fuzzer, in its current state, is unlikely to be used for any real world fuzzing as the implementation is simply too slow and still unstable. With future improvements, it could replace the reliance on NEUZZ and AFL and allow fuzz testers to implement neural fuzzing with a simple library, the foundation of which has been provided here.

5.8 Code Performance

As evidenced by the testing, the speed of the program is very similar when using the `NeuralStage` to running vanilla LibAFL, demonstrating that my code is performant and has little time penalties when running on the hardware. The author believes that this is due to the use of in memory data structures where possible, as opposed to reading from the disk and efficient processing of the network in a separate thread.

5.9 Personal Goals

The project aims also included exploration of many tools new to me, such as machine learning, Rust [25] and fuzz testing in general.

Chapter 6

Conclusion

The overall aim of this research was to establish whether porting a neural network to a library such as LibAFL could be done without significant time penalties and make use of the modularity offered by this library. As evidenced in Chapter 4, this was possible and yielded similar run speeds to the vanilla LibAFL, with very good comparative results.

The LibAFL library proves useful for allowing researchers and fuzz testers to implement their own fuzzers in a way that is compatible with LibAFL. The neural network that is used does not perform as well as expected, but I believe that this can be attributed to the design of the underlying `tch-rs` library and the lack of community support and documentation, which meant that my network was unable to be optimized in the most efficient manner possible.

Furthermore, the ability to use the precomputed metadata that is gathered at runtime instead of recalculating it and rerunning the input of the fuzzer saves processing time for the network, which is a possible issue with the NEUZZ implementation.

Providing further functionality to the LibAFL library is difficult, but achievable, and requires the implementor to have a good understanding of the underlying concepts and code that comprise the fuzzer.

Further testing and optimizations could improve this project and bring it up to the same speed at which modern fuzzers operate, with potential to surpass the performance of the genetic mutations that most fuzzers use.

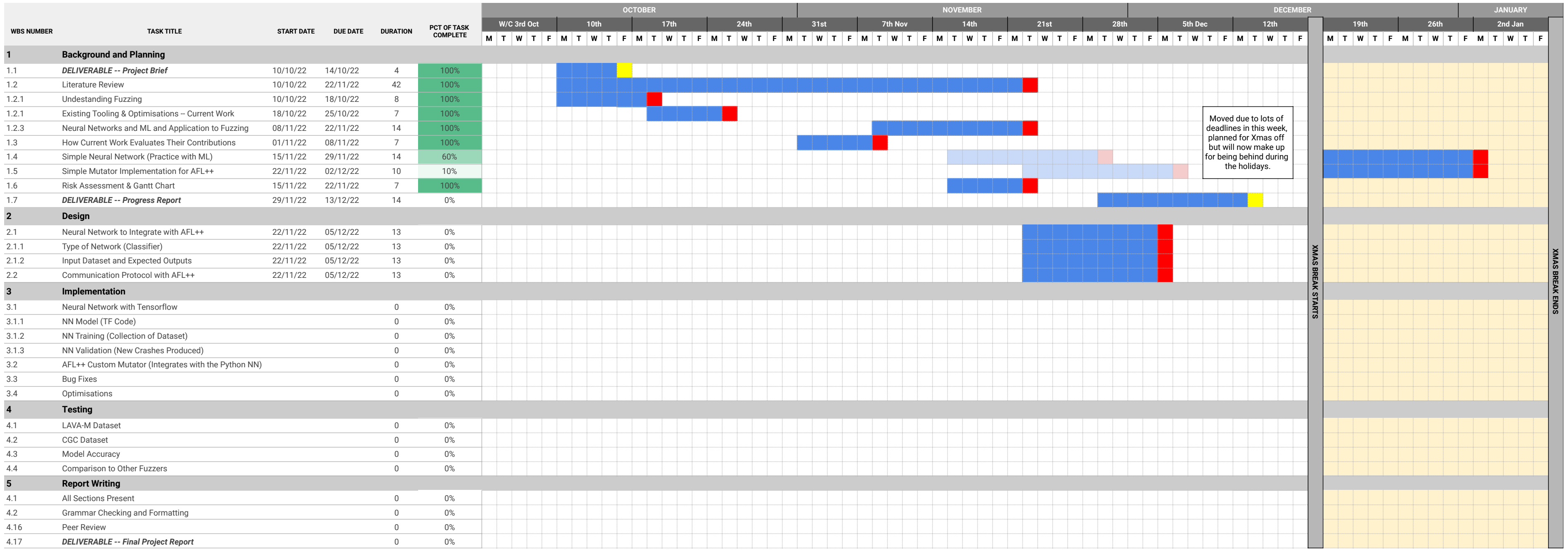
The use of neural networks to improve fuzzers remains an under researched area, as the fastest and highest coverage fuzzers all make use of genetic algorithms.

Appendices

Appendix A

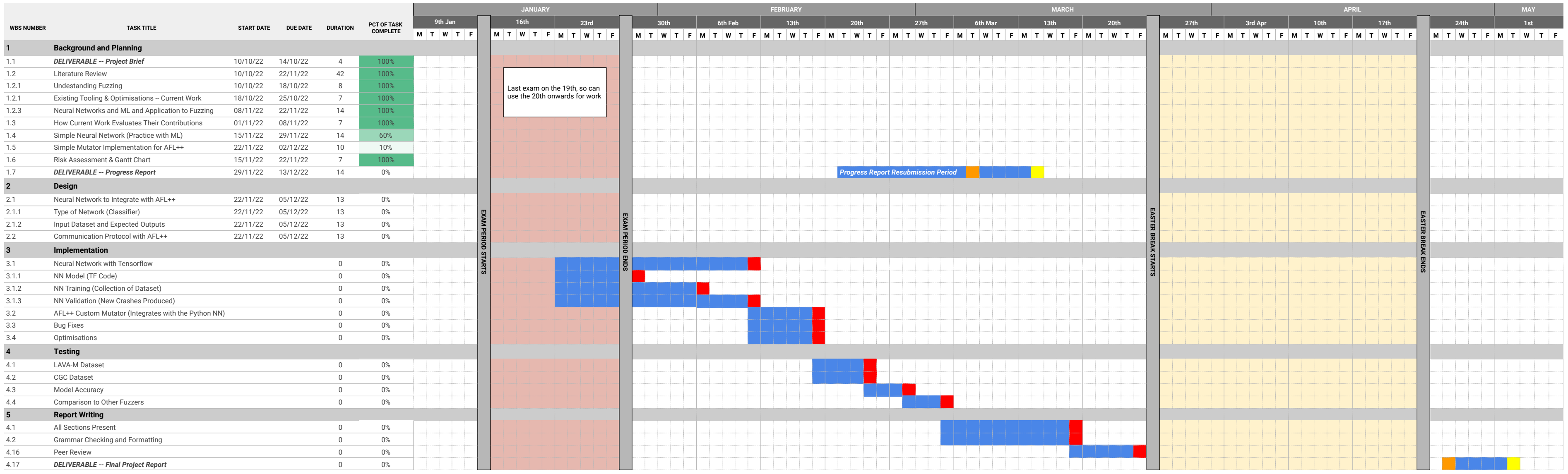
Gantt Chart

The Gantt chart can be seen in its current state in Figures A.1 and A.2. These figures are in vector format, so full detail can be seen when zooming in.



WBS NUMBER	TASK TITLE	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	**Background and Planning**				
1.1	***DELIVERABLE -- Project Brief***	10/10/22	14/10/22	4	100%
1.2	Literature Review	10/10/22	22/11/22	42	100%
1.2.1	Undestanding Fuzzing	10/10/22	18/10/22	8	100%
1.2.1	Existing Tooling & Optimisations -- Current Work	18/10/22	25/10/22	7	100%
1.2.3	Neural Networks and ML and Application to Fuzzing	08/11/22	22/11/22	14	100%
1.3	How Current Work Evaluates Their Contributions	01/11/22	08/11/22	7	100%
1.4	Simple Neural Network (Practice with ML)	15/11/22	29/11/22	14	60%
1.5	Simple Mutator Implementation for AFL++	22/11/22	02/12/22	10	10%
1.6	Risk Assessment & Gantt Chart	15/11/22	22/11/22	7	100%
1.7	***DELIVERABLE -- Progress Report***	29/11/22	13/12/22	14	0%
2	**Design**				
2.1	Neural Network to Integrate with AFL++	22/11/22	05/12/22	13	0%
2.1.1	Type of Network (Classifier)	22/11/22	05/12/22	13	0%
2.1.2	Input Dataset and Expected Outputs	22/11/22	05/12/22	13	0%
2.2	Communication Protocol with AFL++	22/11/22	05/12/22	13	0%
3	**Implementation**				
3.1	Neural Network with Tensorflow			0	0%
3.1.1	NN Model (TF Code)			0	0%
3.1.2	NN Training (Collection of Dataset)			0	0%
3.1.3	NN Validation (New Crashes Produced)			0	0%
3.2	AFL++ Custom Mutator (Integrates with the Python NN)			0	0%
3.3	Bug Fixes			0	0%
3.4	Optimisations			0	0%
4	**Testing**				
4.1	LAVA-M Dataset			0	0%
4.2	CGC Dataset			0	0%
4.3	Model Accuracy			0	0%
4.4	Comparison to Other Fuzzers			0	0%
5	**Report Writing**				
4.1	All Sections Present			0	0%
4.2	Grammar Checking and Formatting			0	0%
4.16	Peer Review			0	0%
4.17	***DELIVERABLE -- Final Project Report***			0	0%

Figure A.1: Gantt Chart from Start of Project to end of Christmas as at 2022-12-11T16:00



WBS NUMBER	TASK TITLE	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	**Background and Planning**				
1.1	***DELIVERABLE -- Project Brief***	10/10/22	14/10/22	4	100%
1.2	Literature Review	10/10/22	22/11/22	42	100%
1.2.1	Undestanding Fuzzing	10/10/22	18/10/22	8	100%
1.2.1	Existing Tooling & Optimisations -- Current Work	18/10/22	25/10/22	7	100%
1.2.3	Neural Networks and ML and Application to Fuzzing	08/11/22	22/11/22	14	100%
1.3	How Current Work Evaluates Their Contributions	01/11/22	08/11/22	7	100%
1.4	Simple Neural Network (Practice with ML)	15/11/22	29/11/22	14	60%
1.5	Simple Mutator Implementation for AFL++	22/11/22	02/12/22	10	10%
1.6	Risk Assessment & Gantt Chart	15/11/22	22/11/22	7	100%
1.7	***DELIVERABLE -- Progress Report***	29/11/22	13/12/22	14	0%
2	**Design**				
2.1	Neural Network to Integrate with AFL++	22/11/22	05/12/22	13	0%
2.1.1	Type of Network (Classifier)	22/11/22	05/12/22	13	0%
2.1.2	Input Dataset and Expected Outputs	22/11/22	05/12/22	13	0%
2.2	Communication Protocol with AFL++	22/11/22	05/12/22	13	0%
3	**Implementation**				
3.1	Neural Network with Tensorflow			0	0%
3.1.1	NN Model (TF Code)			0	0%
3.1.2	NN Training (Collection of Dataset)			0	0%
3.1.3	NN Validation (New Crashes Produced)			0	0%
3.2	AFL++ Custom Mutator (Integrates with the Python NN)			0	0%
3.3	Bug Fixes			0	0%
3.4	Optimisations			0	0%
4	**Testing**				
4.1	LAVA-M Dataset			0	0%
4.2	CGC Dataset			0	0%
4.3	Model Accuracy			0	0%
4.4	Comparison to Other Fuzzers			0	0%
5	**Report Writing**				
4.1	All Sections Present			0	0%
4.2	Grammar Checking and Formatting			0	0%
4.16	Peer Review			0	0%
4.17	***DELIVERABLE -- Final Project Report***			0	0%

Figure A.2: Gantt Chart from end of Christmas to end of Project as at 2022-12-11T16:00

Appendix B

Design Archive Index

Folder	Description
`corpus-samples`	Sample files for using to input to the fuzzer as initial corpora.
`forkserver_neural`	Initial fork server application for the neural network. Since abandoned and replaced with `forkserver_simple`.
`forkserver_simple`	Forkserver application as taken from LibAFL examples. Modified with the `NeuralStage` and recompiled to run with the neural network.
`fuzzing-targets`	Directory with source code and compiled binaries (in the `build` directory), with the stubs injected for using the shared map.
`mnist-testing`	Initial testing with the `tch-rs` library. Not used in the implementation at all.
`neumut`	Rust library that is compatible with LibAFL. Testing code and design document in directory. Within the source, each file is the corresponding module that has been implemented.
`neuzzprogs`	Similar to corpus-samples, these were used in testing for the program inputs as the corpus.
`target`	Output build files of the libraries. Most testing done within the testing ecosystem, so targets not used at all.
`testing`	Test directory, with script to automatically run all fuzzers on their own cores and test the variety of programs. Each fuzzers run information is placed in the log file within the library name subdirectory.
`xpdf`	Unfinished implementation of an XPDF fuzzer with LibAFL. This was used to aid understanding and provide some hands-on experience with LibAFL.

Table B.1: Index of the design archive.

Bibliography

[1] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles CA USA: ACM, Nov. 7, 2022, pp. 1051–1065, ISBN: 9781450394505. DOI: `10.1145/3548606.3560602`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3548606.3560602` (visited on 12/13/2022).

[2] D. She, K. Pei, D. Epstein, *et al.*, "NEUZZ: Efficient fuzzing with neural program smoothing," *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 803–817, Apr. 1, 2019, ARXIV_ID: 1807.05620 MAG ID: 2963674831 S2ID: 51652bfba5204fc4476849bffa2295187267c356. DOI: `10.1109/sp.2019.00052`.

[3] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research.," *WOOT @ USENIX Security Symposium*, Jan. 1, 2020, MAG ID: 3048197573 S2ID: ff4804de2b9db68a052d7113cae41ef2122a1c51.

[4] LLVM Project, *libFuzzer*. [Online]. Available: `https://www.llvm.org/docs/LibFuzzer.html`.

[5] R. Swiecki, *Honggfuzz*. [Online]. Available: `https://github.com/google/honggfuzz`.

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of The ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1, 1990, MAG ID: 2002934700 S2ID: 2c13dcfdc5ea2d355a46fe326c371038a00ba7f5. DOI: `10.1145/96267.96279`.

[7] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," pp. 711–725, May 20, 2018, MAG ID: 2964097210. DOI: `10.1109/sp.2018.00046`.

[8] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 50–59. DOI: `10.1109/ASE.2017.8115618`.

[9] M. Zalewski. "Technical "whitepaper" for afl-fuzz," [lcamtuf.coredump.cx]. (2013), [Online]. Available: `https://lcamtuf.coredump.cx/afl/technical_details.txt` (visited on 12/02/2022).

[10] B. Dolan-Gavitt, P. Hulin, E. Kirda, *et al.*, "LAVA: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, ISSN: 2375-1207, May 2016, pp. 110–121. DOI: `10.1109/SP.2016.15`.

[11] DARPA, *Cyber grand challenge repository*, 2017. [Online]. Available: `https://github.com/cybergrandchallenge/samples`.

[12] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of The ACM*, vol. 63, no. 2, pp. 70–76, Jan. 22, 2020, MAG ID: 3001307185. DOI: `10.1145/3363824`.

[13] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing.," Nov. 1, 2008, MAG ID: 157156687.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," vol. 43, no. 6, pp. 206–215, Jun. 7, 2008, MAG ID: 2065948900. DOI: `10.1145/1375581.1375607`.

[15] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," pp. 1032–1043, Oct. 24, 2016, MAG ID: 2535617737. DOI: `10.1145/2976749.2978428`.

[16] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," pp. 2329–2344, Oct. 30, 2017, MAG ID: 2766540688. DOI: `10.1145/3133956.3134020`.

[17] "LLVM persistent mode." (), [Online]. Available: `https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.persistent_mode.md`.

[18] F. Bellard, "QEMU, a fast and portable dynamic translator," pp. 41–41, Apr. 10, 2005, MAG ID: 1522250664.

[19] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA: IEEE, May 2018, pp. 697–710, ISBN: 9781538643532. DOI: `10.1109/SP.2018.00056`. [Online]. Available: `https://ieeexplore.ieee.org/document/8418632/` (visited on 11/07/2022).

[20] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence.," *NDSS*, Jan. 1, 2019, MAG ID: 2947182139 S2ID: f51ee4f4cb3b3d28b8ff3bbcc2628c3e94352726. DOI: `10.14722/ndss.2019.23371`.

[21] C. Lyu, S. Ji, C. Zhang, *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," presented at the 28th USENIX Security Symposium, 2019.

[22] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, Jan. 29, 2017. DOI: `10.48550/arXiv.1412.6980`. arXiv: `1412.6980[cs]`. [Online]. Available: `http://arxiv.org/abs/1412.6980` (visited on 05/14/2023).

[23] R. Gómez. "Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing names." (May 23, 2018), [Online]. Available: `https://gombru.github.io/2018/05/23/cross_entropy_loss/` (visited on 05/13/2023).

[24] *Linfa*. [Online]. Available: `https://github.com/rust-ml/linfa`.

[25] N. D. Matsakis and F. S. Klock, "The rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Nov. 26, 2014, ISSN: 1094-3641. DOI: `10.1145/2692956.2663188`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2692956.2663188` (visited on 05/15/2023).