Adaptive and Effective Fuzzing: a Data-Driven Approach

Dongdong She

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2023

Abstract

Adaptive and Effective Fuzzing: a Data-Driven Approach

Dongdong She

Security vulnerabilities have a large real-world impact, from ransomware attacks costing billions of dollars every year to sensitive data breaches in government, military and industry. Fuzzing is a popular technique to discover these vulnerabilities in an automated fashion. Industries have poured tons of resources into building large-scale fuzzing factories (e.g., Google's ClusterFuzz and Microsoft's OneFuzz) to test their products and make their product more secure. Despite the wide application of fuzzing in industry, there remain many issues constraining its performance. One fundamental limitation is the rule-based design in fuzzing. Rule-based fuzzers heavily rely on a set of static rules or heuristics. These fixed rules are summarized from human experience, hence failing to generalize on a diverse set of programs.

In this dissertation, we present an adaptive and effective fuzzing framework in data-driven approach. A data-driven fuzzer makes decisions based on the analysis and reasoning of data rather than the static rules. Hence it is more adaptive, effective, and flexible than a typical rule-based fuzzer. More interestingly, the data-driven approach can bridge the connection from fuzzing to various data-centric domains (e.g., machine learning, optimizations and social network), enabling sophisticated designs in the fuzzing framework.

A general fuzzing framework consists of two major components: seed scheduling and seed mutation. The seed scheduling module selects a seed from a seed corpus that includes multiple testcases. Then seed mutation module applies perturbation on the selected seed to generate a new

testcase. First, we present Neuzz, the first machine learning (ML) based general-purpose fuzzer that adopts ML to seed mutation and greatly improves fuzzing performance. Then we present MTFuzz, a follow-up work of Neuzz by including diverse data into ML to generate effective seed mutations. In the end, we present K-Scheduler, a fuzzer-agnostic seed scheduling algorithm in data-driven approach. K-Scheduler leverages the graph data (i.e., inter-procedural control flow graph) and dynamic coverage data (i.e., code coverage bitmap) to construct a dynamic graph and schedule seeds by the graph centrality scores on that graph. It can significantly improve the fuzzing performance than the-state-of-art seed schedulers on various fuzzers widely-used in the industry.

Table of Contents

List of Figures

List of Tables

Dedication

To my family.

Chapter 1: Introduction

Security vulnerabilities can significantly impact our daily lives. They are abused by attackers to launch various attacks. The rampant global ransomware, often built with security vulnerabilities, has been costing billions of dollars every year since 2017 [1]. Security vulnerabilities also cause many confidential data breaches in government, military and industry that can result in profound consequences. Recent news shows that 400 million Twitter user accounts are being leaked and up for sale on the black market [2]. Through sophisticated exploits of multiple security vulnerabilities, attackers even directly hacked into financial systems such as banks or crypto exchange centers to steal millions of dollars [3].

To make the software more secure, researchers come up with many automated software testing techniques, such as fuzzing, static analysis, dynamic analysis, formal verification and symbolic execution [4, 5, 6, 7, 8]. Among them, fuzzing is a popular and effective technique that has been widely used in industry. It is a lightweight software testing technique and can scale to large-real world applications. Moreover, fuzzing can be easily deployed in industrial settings due to its simple design. Despite the wide application of fuzzing, many issues still constrain its performance. One of the fundamental limitations is the rule-based design. A rule-based fuzzer heavily relies on a set of static rules or heuristics and often fails to generalize on a diverse set of programs.

Unlike rule-based approach, data-driven approach is a better design that is more flexible, adaptive and effective on various input domains. A data-driven system makes decisions based on the analysis and reasoning of dynamic data rather than static rules/heuristics. Therefore, it enjoys better generalization than a rule-based counterpart. In this dissertation, we propose a novel fuzzing framework in a data-driven approach. Our data-driven fuzzer can learn useful knowledge from the massive amount of data collected during a fuzzing campaign, then use the learned knowledge to enable smart and effective fuzzing.

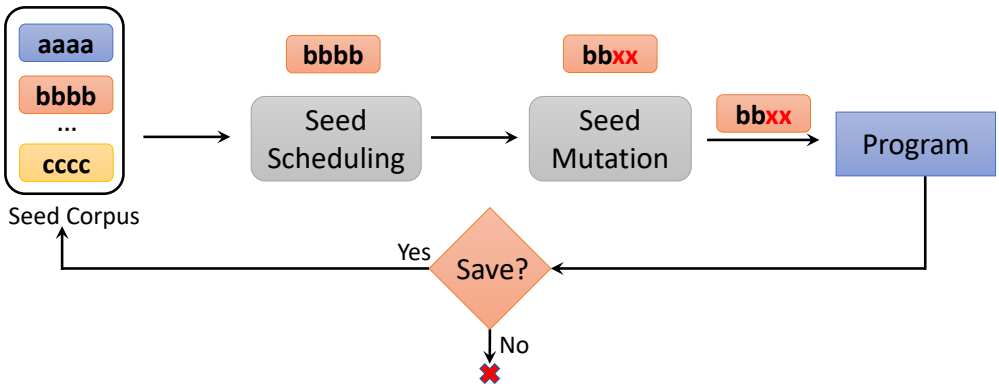


Figure 1.1: **A general workflow of fuzzing**

1.1 General Workflow of Fuzzing

Fuzzing has been widely used in industry. Tech giants such as Google, Meta and Mircosoft leverage this simple yet effective technique to test diverse software products such as kernel, device driver, application software even IoT devices and CPU [9]. Essentially, fuzzing is an *iterative* search that is aimed at finding inputs that can trigger vulnerabilities. The search consists of a large number of iterations where a new testcase is generated and tested. Figure 1.1 shows a general workflow of the fuzzing framework.

A fuzzing campaign starts with a seed corpus. The seed corpus comprises a set of testcases; each one is called a seed. The seed scheduling module first chooses a seed from the seed corpus. Then seed mutation applies perturbation on the selected seed to generate a new testcase. The newly-generated testcase is sent to the tested program for dynamic execution. Meanwhile, fuzzer monitors the execution of new testcase to see if there are any interesting behaviors (i.e., new code coverage, crash and hang). If found, fuzzer saves the new testcase into the seed corpus for further mutation; else, it simply discards that testcase. In the end, fuzzer iteratively repeat this process until

timeout. As shown in Figure 1.1, the entire workflow of fuzzing is a loop consisting of two major steps: seed scheduling and seed mutation. A real-world fuzzing campaign can have millions or even billions of such search iterations.

1.2 Limitation of Rule-based Fuzzing

Existing fuzzers heavily rely on rule-based design. A typical rule-based system hardcodes a set of static rules or heuristics. As a result, these fixed rules often fail on diverse input domains. A rule-based fuzzer inherently suffers from this generalization issue. For example, a rule-based fuzzer schedules seed by execution runtime. It favors seed with the smallest execution time at each scheduling round in order to boost fuzzing throughput and overall performance. Such fixed heuristic may show superior performance on programs that are sensitive to execution runtime, but fail on programs where the format of seed matters. Because a high-quality seed with a valid format might not have the smallest execution runtime.

Real-world programs can have drastically different properties and semantics. Although existing works incorporate many heuristics into the fuzzer design, such as file size, execution time, hit count of code block and random strategies, it is almost impossible to build a general strategy with fixed rules that is effective on diverse programs. Therefore, a general, flexible and adaptive solution is urgently needed in the fuzzing community.

1.3 Fuzzing in a Data-Driven Approach

Data-driven approach is a common solution to provide adaptivity and generalization across diverse input domains, widely used in statistics, machine learning and business analysis. It treats diverse input domains as dynamic data and makes decisions by the analysis and reasoning of data, rather than static rules. In this dissertation, we propose a general fuzzing framework in a data-driven approach. Given that fuzzing is an iterative search composed of a large number of iterations, data-driven fuzzer leverages the massive amount of *past* iterations and extracts useful knowledge to guide *future* iterations. More interestingly, data-driven approach can bridge the connection from

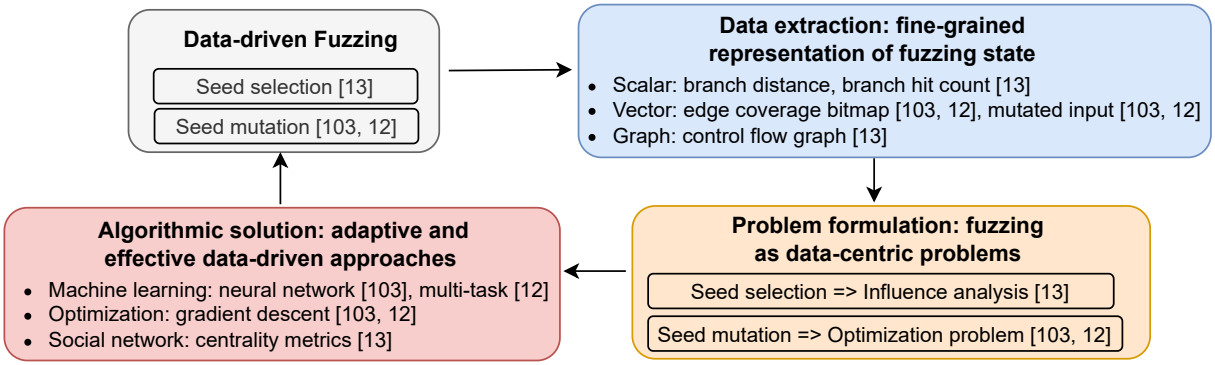


Figure 1.2: **An overview of data-driven fuzzing and summary of my PhD research**

fuzzing to other data-centric domains, such as optimization, machine learning and social network analysis, thus enabling sophisticated design in the general fuzzing framework.

Figure 1.2 shows an overview of data-driven fuzzing. It first extracts data in diverse forms that are fine-grained representations of the fuzzing state. Then with these data, we model fuzzing as various data-centric problems, such as seed selection as an influence analysis problem and seed mutation as an optimization problem. Then we come up with algorithmic solutions such as machine learning, optimization algorithms and social network analysis to solve these problems.

This dissertation presents three data-driven fuzzers that apply this methodology to the two main components (i.e., seed selection and seed mutation) in a general fuzzing framework.

NEUZZ. We build the first general-purpose ML-based fuzzer that can generate effective mutation with a neural network (NN) model [10, 11]. The goal of fuzzing is to search for inputs within a high-dimensional input space in order to increase code coverage. We formally formulate it as an optimization problem whose goal is to maximize the total code coverage. Existing works heavily rely on random mutation and hence are not efficient. We propose a novel gradient-guided mutation scheme to generate high-quality mutations following the gradient of the tested programs. The gradients are obtained from a NN model that approximates the program behaviors of the tested program. Through searching the gradient of the tested program, Neuzz greatly narrows down the search space and improves fuzzing efficiency.

MTFuzz. Recent works point out that code coverage widely used in fuzzer is a coarse-grained

metric and fails to reflect true fuzzing performance. More expressive and fine-grained metrics are needed to guide effective fuzzing. Therefore, we incorporate three coverage metrics into a data-driven fuzzer with a multi-task learning model [12]. Each coverage metric models a unique program property. The code coverage monitors the basic blocks that are dynamically exercised by testcases generated fuzzer. The context coverage captures the function-calling context of each covered basic block. The last metric approach level represents the distance between the visited basic block and unvisited basic block on the control flow graph of the tested program. We train a NN model to approximate multiple program properties in the multi-task learning setting. Then we extract the gradient to jointly affect these program properties and guide effect mutations. Our evaluation demonstrates that MTFuzz can significantly outperform a single task NN-based fuzzer.

K-Scheduler. We propose a general seed scheduling algorithm based on graph centrality [13]. Fuzzing is an iterative search problem aimed at covering more code regions, and seed scheduling in fuzzing determines the starting point for each search iteration. Search from a good starting point can discover many codes and search from a bad starting point can discover few codes. Existing works all use some fixed rules or heuristics and ignore graph information of the program i.e., control flow graph(CFG). We formally model fuzzing as a graph search problem that aims to discover the maximum number of nodes on CFG. And seed scheduling can be formulated as a graph search problem within the CFG to find a promising seed that can lead to the discovery of the maximum number of new nodes. We build a dynamic graph with CFG and online fuzzing data, then perform efficient graph centrality analysis to identify the most influential seeds on all the new nodes in CFG.

1.4 Roadmap

The rest of the dissertation is organized as follows. Chapter 2 presents NEUZZ for data-driven mutation guided by a NN model. Chapter 3 describes MTFuzz, which jointly learns multiple program properties in a multi-task model and generates a joint gradient to guide mutation. Chapter 4 introduces K-Scheduler that proposes a data-driven seed scheduling algorithm based on graph centrality scores. Finally, we conclude in Chapter 5.

Chapter 2: NEUZZ: Efficient Fuzzing with Neural Program Smoothing

Fuzzing has become the de facto standard technique for finding software vulnerabilities. However, even state-of-the-art fuzzers are not very efficient at finding hard-to-trigger software bugs. Most popular fuzzers use evolutionary guidance to generate inputs that can trigger different bugs. Such evolutionary algorithms, while fast and simple to implement, often get stuck in fruitless sequences of random mutations. Gradient-guided optimization presents a promising alternative to evolutionary guidance. Gradient-guided techniques have been shown to significantly outperform evolutionary algorithms at solving high-dimensional structured optimization problems in domains like machine learning by efficiently utilizing gradients or higher-order derivatives of the underlying function.

However, gradient-guided approaches are not directly applicable to fuzzing as real-world program behaviors contain many discontinuities, plateaus, and ridges where the gradient-based methods often get stuck. We observe that this problem can be addressed by creating a smooth surrogate function approximating the target program's discrete branching behavior. In this work, we propose a novel program smoothing technique using surrogate neural network models that can incrementally learn smooth approximations of a complex, real-world program's branching behaviors. We further demonstrate that such neural network models can be used together with gradient-guided input generation schemes to significantly increase the efficiency of the fuzzing process.

Our extensive evaluations demonstrate that NEUZZ significantly outperforms 10 state-of-the-art graybox fuzzers on 10 popular real-world programs both at finding new bugs and achieving higher edge coverage. NEUZZ found 31 previously unknown bugs (including two CVEs) that other fuzzers failed to find in 10 real-world programs and achieved 3X more edge coverage than all of the tested graybox fuzzers over 24 hour runs. Furthermore, NEUZZ also outperformed existing fuzzers on both LAVA-M and DARPA CGC bug datasets.

2.1 Introduction

Fuzzing has become the de facto standard technique for finding software vulnerabilities [4, 14]. The fuzzing process involves generating random test inputs and executing the target program with these inputs to trigger potential security vulnerabilities [15]. Due to its simplicity and low performance overhead, fuzzing has been very successful at finding different types of security vulnerabilities in many real-world programs [16, 17, 18, 19, 5, 20]. Despite their tremendous promise, popular fuzzers, especially for large programs, often tend to get stuck trying redundant test inputs and struggle to find security vulnerabilities hidden deep within program logic [21, 22, 23].

Conceptually, fuzzing is an optimization problem whose goal is to find program inputs that maximize the number of vulnerabilities found within a given amount of testing time [24]. However, as security vulnerabilities tend to be sparse and erratically distributed across a program, most fuzzers aim to test as much program code as they can by maximizing some form of code coverage (*e.g.*, edge coverage) to increase their chances of finding security vulnerabilities. Most popular fuzzers use evolutionary algorithms to solve the underlying optimization problem—generating new inputs that maximize code coverage [4, 5, 20, 25]. Evolutionary optimization starts from a set of seed inputs, applies random mutations to the seeds to generate new test inputs, executes the target program for these inputs, and only keeps the promising new inputs (*e.g.*, those that achieve new code coverage) as part of a corpus for further mutation. However, as the input corpus gets larger, the evolutionary process becomes increasingly less efficient at reaching new code locations.

One of the main limitations of evolutionary optimization algorithms is that they cannot leverage the structure (*i.e.*, gradients or other higher-order derivatives) of the underlying optimization problem. Gradient-guided optimization (*e.g.*, gradient descent) is a promising alternative approach that has been shown to significantly outperform evolutionary algorithms at solving high-dimensional structured optimization problems in diverse domains including aerodynamic computations and machine learning [26, 27, 28].

However, gradient-guided optimization algorithms cannot be directly applied to fuzzing real-

world programs as they often contain significant amounts of discontinuous behaviors (cases where the gradients cannot be computed accurately) due to widely different behaviors along different program branches [29, 30, 31, 32, 7]. We observe that this problem can be overcome by creating a smooth (*i.e.*, differentiable) surrogate function approximating the target program's branching behavior with respect to program inputs. Unfortunately, existing program smoothing techniques [30, 32] incur prohibitive performance overheads as they depend heavily on symbolic analysis that does not scale to large programs due to several fundamental limitations like path explosion, incomplete environment modeling, and large overheads of symbolic memory modeling [33, 34, 35, 36, 37, 38, 39].

In this work, we introduce a novel, efficient, and scalable program smoothing technique using feed-forward Neural Networks (NNs) that can incrementally learn smooth approximations of complex, real-world program branching behaviors, *i.e.*, predicting the control flow edges of the target program exercised by a particular given input. We further propose a gradient-guided search strategy that computes and leverages the gradient of the smooth approximation (*i.e.*, an NN model) to identify target mutation locations that can maximize the number of detected bugs in the target program. We demonstrate how the NN model can be refined by incrementally retraining the model on mispredicted program behaviors. We find that feed-forward NNs are a natural fit for our task because of (i) their demonstrated ability to approximate complex non-linear functions, as implied by the universal approximation theorem [40], and (ii) their support for efficient and accurate computation of gradients/higher-order derivatives [28].

We design and implement our technique as part of NEUZZ, a new learning-enabled fuzzer. We compare NEUZZ with 10 state-of-the art fuzzers on 10 real-world programs covering 6 different file formats, (*e.g.*, ELF, PDF, XML, ZIP, TTF, and JPEG) with an average of 47, 546 lines of code, the LAVA-M bug dataset [41], and the CGC dataset [42]. Our results show that NEUZZ consistently outperforms all the other fuzzers by a wide margin both in terms of detected bugs and achieved edge coverage. NEUZZ found 31 previously unknown bugs (including CVE-2018-19931 and CVE-2018-19932) in the tested programs that other fuzzers failed to find. Our tests on the DARPA

CGC dataset also confirmed that NEUZZ can outperform state-of-the-art fuzzers like Driller [21] at finding different bugs.

2.2 Optimization Basics

In this section, we first describe the basics of optimization and the benefits of gradient-guided optimization over evolutionary guidance for smooth functions. Finally, we demonstrate how fuzzing can be cast as an optimization problem.

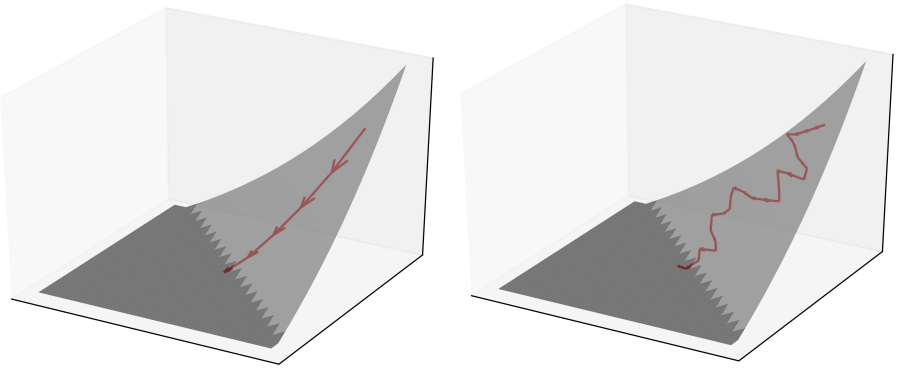
An optimization problem usually consists of three different components: a vector of parameters x, an objective function $F(x)$ to be minimized or maximized, and a set of constraint functions $C_i(x)$ each involving either inequality or equality that must be satisfied. The goal of the optimization process is to find a concrete value of the parameter vector x that maximizes/minimizes $F(x)$ while satisfying all constraint functions $C_i(x)$ as shown below.

$$\underset{x \in R^n}{max/min}\ F(x) \text{ subject to } \begin{cases} C_i(x) \geq 0, i \in N \\ C_i(x) = 0, i \in Q \end{cases} \tag{2.1}$$

Here R, N, and Q denote the sets of real numbers, the indices for inequality constraints, and the indices for equality constraints, respectively.

Function smoothness & optimization. Optimization algorithms usually operate in a loop beginning with an initial guess of the parameter vector x and gradually iterating to find better solutions. The key component of any optimization algorithm is the strategy it uses to move from one value of x to the next. Most strategies leverage the values of the objective function F, the constraint functions C_i, and, if available, the gradient/higher-order derivatives.

The ability and efficiency of different optimization algorithms to converge to the optimal solution heavily depend on the nature of the objective and constraint functions F and C_i. In general, smoother functions (*i.e.*, those with well-defined and computable derivatives) can be more efficiently optimized than functions with many discontinuities (*e.g.*, ridges or plateaus). Intuitively,



(a) **gradient descent** (b) **evolutionary algorithm**

Figure 2.1: **Gradient-guided optimization algorithms like gradient descent can be significantly more efficient than evolutionary algorithms that do not use any gradient information**

the smoother the objective/constraint functions are, the easier it is for the optimization algorithms to accurately compute gradients or higher-order derivatives and use them to systematically search the entire parameter space.

For the rest of this paper, we specifically focus on unconstrained optimization problems that do not have any constraint functions, *i.e.*, $C = \phi$, as they closely mimic fuzzing, our target domain. For unconstrained smooth optimization problems, gradient-guided approaches can significantly outperform evolutionary strategies at solving high-dimensional structured optimization problems [26, 27, 28]. This is because gradient-guided techniques effectively leverage gradients/higher-order derivatives to efficiently converge to the optimal solution as shown in Figure 2.1.

Convexity & gradient-guided optimization. For a common class of functions called convex functions, gradient-guided techniques are highly efficient and can always converge to the globally optimal solution [43]. Intuitively, a function is convex if a straight line connecting any two points on the graph of the function lies entirely above or on the graph. More formally, a function f is called convex if the following property is satisfied by all pairs of points x and y in its domain: $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y), \forall t \in [0,1]$.

However, in non-convex functions, gradient-guided approach may get stuck at locally optimal solutions where the objective function is greater (assuming that the goal is to maximize) than all nearby feasible points but there are other larger values present elsewhere in the entire range

of feasible parameter values. However, even for such cases, simple heuristics like restarting the gradient-guided methods from new randomly chosen starting points have been shown to be highly effective in practice [28, 43].

Fuzzing as unconstrained optimization. Fuzzing can be represented as an unconstrained optimization problem where the objective is to maximize the number of bugs/vulnerabilities found in the test program for a fixed number of test inputs. Therefore, the objective function can be thought of as $F_p(x)$, which returns 1 if input x triggers a bug/vulnerability when the target program p is executed with input x. However, such a function is too ill-behaved (*i.e.*, mostly containing flat plateaus and a few very sharp transitions) to be optimized efficiently.

Therefore, most graybox fuzzers instead try to maximize the amount of tested code (*e.g.*, maximize edge coverage) as a stand-in proxy metric [4, 5, 44, 45, 7]. Such an objective function can be represented as $F'_p(x)$ where F' returns the number of new control flow edges covered by the input x for program P. Note that F' is relatively easier to optimize than the original function F as the number of all possible program inputs exercising new control flow edges tend to be significantly higher than the inputs that trigger bugs/security vulnerabilities.

Most existing graybox fuzzers use evolutionary techniques [4, 5, 44, 45, 7] along with other domain-specific heuristics as their main optimization strategy. The key reason behind picking such algorithms over gradient-guided optimization is that most real-world programs contain many discontinuities due to significantly different behaviors along different program paths [46]. Such discontinuities may cause the gradient-guided optimization to get stuck at non-optimal solutions. In this paper, we propose a new technique using a neural network for smoothing the target programs to make them suitable for gradient-guided optimization and demonstrate how fuzzers might exploit such strategies to significantly boost their effectiveness.

2.3 Overview of Our Approach

Figure 2.2 presents a high level overview of our approach. We describe the key components in detail below.

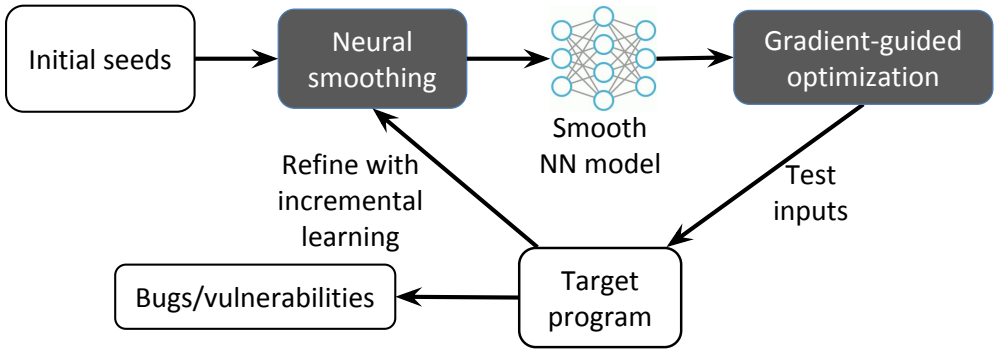
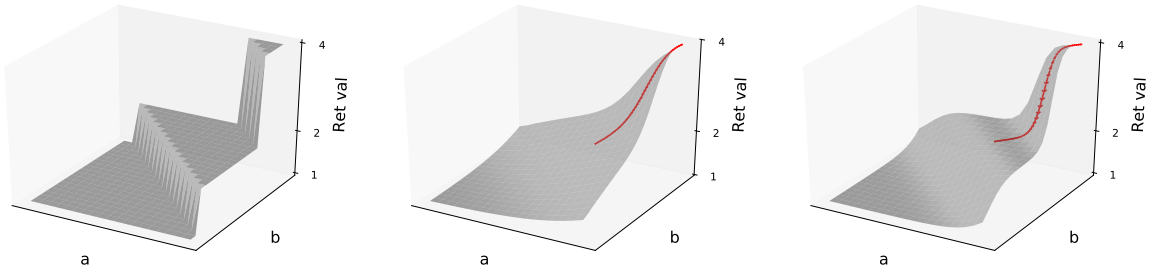


Figure 2.2: **An overview of our approach**

```
if(z < 1){
  return 1;
}
else if(z < 2){
  //vulnerability
  return 2;
}
else if(z < 4){
  return 4;
}
```



(a) Original (b) NN smoothing (c) NN smoothing + refining

Figure 2.3: **Simple code snippet demonstrating the benefits of neural smoothing for fuzzing**

Neural program smoothing. Approximating a program's discontinuous branching behavior smoothly is essential for accurately computing gradients or higher-order derivatives that are necessary for gradient-guided optimization. Without such smoothing, the gradient-guided optimization process may get stuck at different discontinuities/plateaus. The goal of the smoothing process is to create a smooth function that can mimic a program's branching behavior without introducing large errors (*i.e.*, it deviates minimally from the original program behavior). We use a feed-forward neural network (NN) for this purpose. As implied by the universal approximation theorem [40],

an NN is a great fit for approximating arbitrarily complex (potentially non-linear and non-convex) program behaviors. Moreover, NNs, by design, also support efficient gradient computation that is crucial for our purposes. We train the NN by either using existing test inputs or with the test input corpus generated by existing evolutionary fuzzers as shown in Figure 2.2.

Gradient-guided optimization. The smooth NN model, once trained, can be used to efficiently compute gradients and higher-order derivatives that can then be leveraged for faster convergence to the optimal solution. Different variants of gradient-guided algorithms like gradient descent, Newton's method, or quasi-Newton methods like the L-BFGS algorithm use gradients or higher-order derivatives for faster convergence [47, 48, 49]. Smooth NNs enable the fuzzing input generation process to potentially use all of these techniques. In this paper, we design, implement and evaluate a simple gradient-guided input generation scheme tailored for coverage-based fuzzing as described in detail in Section 2.4.3.

Incremental learning. Any types of existing test inputs (as long as they expose diverse behaviors in the target program) can be potentially used to train the NN model and bootstrap the fuzzing input generation process. In this paper, we train the NN by collecting a set of test inputs and the corresponding edge coverage information by running evolutionary fuzzers like AFL.

However, as the initial training data used for training the NN model may only cover a small part of the program space, we further refine the model through incremental training as new program behaviors are observed during fuzzing. The key challenge in incremental training is that if an NN is only trained on new data, it might completely forget the rules it learned from old data [50]. We avoid this problem by designing a new coverage-based filtration scheme that creates a condensed summary of both old and new data, allowing the NN to be trained efficiently on them.

A Motivating Example. We show a simple motivating example in 4.2 to demonstrate the key insight behind our approach. The simple `C` code snippet shown in Figure 4.2 demonstrates a general switch-like code pattern commonly found in many real-world programs. In particular, the example code computes a non-linear exponential function of the input (*i.e.*, `pow(3,a+b)`). It returns different values based on the output range of the computed function. Let us also assume that a

buggy code block is exercised if the function output range is in (1,2).

Consider the case where evolutionary fuzzers like AFL have managed to explore the branches in lines 2 and 9 but fail to explore branch in line 5. The key challenge here is to find values of `a` and `b` that will trigger the branch at line 5. Evolutionary fuzzers often struggle with such code as the odds of finding a solution through random mutation are very low. For example, Figure 2.3a shows the original function that the code snippet represents. There is a sharp jump in the function surface from $a+b=0$ to $a+b-\epsilon=0$ ($\epsilon \rightarrow +0$). To maximize the edge coverage during fuzzing, an evolutionary fuzzer can only resort to random mutations to the input as such techniques do not consider the shape of function surface. By contrast, our NN smoothing and gradient-guided mutations are designed to exploit the function surface shape as measured by the gradients.

We train an NN model on the program behaviors from the other two branches. The NN model smoothly approximates the program behaviors as shown in Figure 2.3b and 2.3c. We then use the NN model to perform more effective gradient-guided optimization to find the desired values of a and b and incrementally refine the model until the desired branch is found that exercises the target bug.

2.4 Methodology

We describe the different components of our scheme in detail below.

2.4.1 Program smoothing

Program smoothing is an essential step to make gradient-guided optimization techniques suitable for fuzzing real-world programs with discrete behavior. Without smoothing, gradient-guided optimization techniques are not very effective for optimizing non-smooth functions as they tend to get stuck at different discontinuities [29]. The smoothing process minimizes such irregularities and therefore makes the gradient-guided optimization significantly more effective on discontinuous functions.

In general, the smoothing of a discontinuous function f can be thought of as a convolution

operation between f and a smooth mask function g to produce a new smooth output function as shown below. Some examples of popular smoothing masks include different Gaussian and Sigmoid functions.

$$f'(x) = \int_{-\infty}^{+\infty} f(a)g(x-a)da \tag{2.2}$$

However, for many practical problems, the discontinuous function f may not have a closed-form representation and thus analytically computing the above-mentioned integral is not possible. In such cases, the discrete version $f'(x) = \sum_a f(a)g(x-a)$ is used and the convolution is computed numerically. For example, in image smoothing, often fixed-sized 2-D convolution kernels are used to perform such computation. However, in our setting, f is a computer program and therefore the corresponding convolution cannot be computed analytically.

Program smoothing techniques can be classified into two broad categories: blackbox and whitebox smoothing. The blackbox approach picks discrete samples from the input space of f and computes the convolution numerically using these samples. By contrast, the whitebox approach looks into the program statements/instructions and try to summarize their effects using symbolic analysis and abstract interpretation [30, 32]. The blackbox approaches may introduce large approximation errors while whitebox approaches incur prohibitive performance overhead, which makes them infeasible for real-world programs.

To avoid such problems, we use NNs to learn a smooth approximation of program behaviors in a graybox manner (*e.g.*, by collecting edge coverage data) as described below.

2.4.2 Neural program smoothing

In this paper, we propose a novel approach to program smoothing by using surrogate NN models to learn and iteratively refine smooth approximations of the target program based on the observed program behaviors. The surrogate neural networks can smoothly generalize to the observed program behaviors while also accurately modeling potentially non-linear and non-convex behaviors. The neural networks, once trained, can be used for efficiently computing gradients and higher-level

derivatives to guide the fuzzing input generation process as shown in Figure 4.2.

Why NNs? As implied by the universal approximation theorem [40], an NN is a great fit for approximating complex (potentially non-linear and non-convex) program behaviors. The advantages of using NNs for learning smooth program approximations are as follows: (i) NNs can accurately model complex non-linear program behaviors and can be trained efficiently. Prior works on model-based optimization have used simple linear and quadratic models [51, 52, 53, 54]. However, such models are not a good fit for modeling real-world software with highly non-linear and non-convex behaviors; (ii) NNs support efficient computation of their gradients and higher-order derivatives. Therefore, the gradient-guided algorithms can compute and use such information during fuzzing without any extra overhead; and (iii) NNs can generalize and learn to predict a program's behaviors for unseen inputs based on its behaviors on similar inputs. Therefore, NNs can potentially learn a smooth approximation of the entire program based on its behaviors for a small number of input samples.

NN Training. While NNs can be used to model different aspects of a program's behavior, in this paper we use them specifically for modeling the target program's branching behavior (*i.e.*, predicting control flow edges exercised by a given program input). One of the challenges in using neural nets to model branching behavior is the need to accept variably-sized input. Feedforward NNs, unlike real-world programs, typically accept fixed size input. Therefore, we set a maximum input size threshold and pad any smaller-sized inputs with null bytes during training. Note that supporting larger inputs is not a major concern as modern NNs can easily scale to millions of parameters. Therefore, for larger programs, we can simply increase the threshold size, if needed. However, we empirically find that relatively modest threshold values yield the best results and larger inputs do not increase modeling accuracy significantly.

Formally, let $f : \{\texttt{0x00}, \texttt{0x01}, ..., \texttt{0xff}\}^m \rightarrow \{0, 1\}^n$ denote the NN that takes program inputs as byte sequences with size m and outputs an edge bitmap with size n. Let θ denote the trainable weight parameters of f. Given a set of training samples (X, Y), where X is a set of input bytes and Y represents the corresponding edge coverage bitmap, the training task of the parametric

function $f(x, \theta) = y$ is to obtain the parameter $\hat{\theta}$ such that $\hat{\theta} = \arg\min_{\theta} \sum_{x \in X, y \in Y} L(y, f(x, \theta))$ where $L(y, f(x, \theta))$ defines the loss function between the output of the NN and the ground truth label $y \in Y$ in the training set. The training task is to find the weight parameters θ of the NN f to minimize the loss, which is defined using a distance metric. In particular, we use binary cross-entropy to compute the distance between the predicted bitmap and the true coverage bitmap. In particular, let y_i and $f_i(x, \theta)$ denote the i-th bit in the output bitmap of ground truth and f's prediction, respectively. Then, the binary cross-entropy between these two is defined as:

$$-\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot log(f_i(x, \theta) + (1 - y_i) \cdot log(1 - f_i(x, \theta)]$$

In this paper, we use feed-forward fully connected NNs to model the target program's branching behavior. The feed-forward architecture allows highly efficient computation of gradients and fast training [55].

Our smoothing technique is agnostic to the source of the training data and therefore the NN can be trained on any edge coverage data gathered from an existing input corpus. For our prototype implementation, we use input corpora generated by existing evolutionary fuzzers like AFL to train our initial model.

Training data preprocessing. Edge coverage exercised by the training data often tends to be biased, as it only contains labels for a small section of all edges in a program. For example, some edges might always be exercised together by all inputs in the training data. This type of correlation between a set of labels is known in machine learning as multicollinearity, which often prevents the model from converging to a small loss value [56]. To avoid such cases, we follow the common machine learning practice of dimensionality reduction by merging the edges that always appear together in the training data into one edge. Furthermore, we only consider the edges that have been activated at least once in the training data. These steps significantly reduce the number of labels to around $4,000$ from around $65,536$ on average. Note that we rerun the data preprocessing

step at every iteration of incremental learning and thus some merged labels may get split as their correlation may decrease as new edge data is discovered during fuzzing.

2.4.3 Gradient-guided optimization

Different gradient-guided optimization techniques like gradient descent, Newton's method, or quasi-Newton methods like L-BFGS can use gradient or higher-order derivatives for faster convergence [47, 48, 49]. Smooth NNs enable the fuzzing input generation process to potentially use any of these techniques by supporting efficient computation of gradient and higher-order derivatives. In this paper, we specifically design a simple gradient-guided search scheme that is robust to minor prediction errors to demonstrate the effectiveness of our approach. We leave the exploration of more sophisticated techniques as future work.

Before describing our mutation strategy, which is based on the NN's gradient, we first provide a formal definition of the gradient that indicates how much each input byte should be changed to affect the output of a final layer neuron in the NN (indicating changed edge coverage in the program) f [57]. Here each output neuron corresponds to a particular edge and computes a value between 0 and 1 summarizing the effect of the given input byte on a particular edge. The gradients of the output neurons of the NN f w.r.t. the inputs have been extensively used for adversarial input generation [58, 59] and visualizing/understanding DNNs [60, 57, 61]. Intuitively, in our setting, the goal of gradient-based guidance is to find inputs that will change the output of the final layer neurons corresponding to different edges from 0 to 1.

Given a parametric NN $y = f(\theta, x)$ as defined in Section 2.4.2, let y_i denote the output of i-th neuron in the final layer of f, which can also be written as $f_i(\theta, x)$. The gradient G of $f_i(\theta, x)$ with respect to input x can be defined as $G = \nabla_x f_i(\theta, x) = \partial y_i / \partial x$. Note that f's gradient w.r.t to θ can be easily computed as the NN training process requires iteratively computing this value to update θ. Therefore, G can also be easily calculated by simply replacing the computation of the gradient of θ to that of x. Note that the dimension of the gradient $\boldsymbol{G}$ is identical to that of the input x, which is a byte sequence in our case.

Algorithm 1 Gradient-guided mutation

Input: $seed \leftarrow$ initial seed
$iter \leftarrow$ number of iterations
$k \leftarrow$ parameter for picking top-k critical bytes for mutation
$g \leftarrow$ computed gradient of seed

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: $v \leftarrow clip(v, 0, 255)$
7: $gen\_mutate(seed, loc, v)$
8: **end for**
9: **for** $loc \in locations$ **do**
10: $v \leftarrow seed[loc] - m * sign(g[loc])$
11: $v \leftarrow clip(v, 0, 255)$
12: $gen\_mutate(seed, loc, v)$
13: **end for**
14: **end for**
15: **end for**

Gradient-guided optimization. Algorithm 1 shows the outline of our gradient-guided input generation process. The key idea is to identify the input bytes with highest gradient values and mutate them, as they indicate higher importance to the NN and thus have higher chances of causing major changes in the program behavior (*e.g.*, flipping branches).

Starting from a seed, we iteratively generate new test inputs. As shown in Algorithm 1, at each iteration, we first leverage the absolute value of the gradient to identify the input bytes that will cause the maximum change in the output neurons corresponding to the untaken edges. Next, we check the sign of the gradient for each of these bytes to decide the direction of the mutation (*e.g.*, increment or decrement their values) to maximize/minimize the objective function. Conceptually, our usage of gradient sign is similar to the adversarial input generation methods introduced in [58]. We also bound the mutation of each byte in its legal range (0-255). Lines 6 and 10 denote the use of `clip` function to implement such bounding.

We start the input generation process with a small mutation target (k in Algorithm 1) and

exponentially grow the number of target bytes to mutate to effectively cover the large input space.

2.4.4 Refinement with incremental learning

The efficiency of the gradient-guided input generation process depends heavily on how accurately the surrogate NN can model the target program's branching behavior. To achieve higher accuracy, we incrementally refine the NN model when divergent program behaviors are observed during the fuzzing process (*i.e.*, when the target program's behavior does not match the predicted behavior). We use incremental learning techniques to keep the NN model updated by learning from new data when new edges are triggered.

The main challenge behind NN refinement is preventing the NN model from abruptly forgetting the information it previously learned from old data while training on new data. Such forgetting is a well-known phenomenon in deep learning literature and has been thought to be a result of the stability-plasticity dilemma [62, 63]. To avoid such forgetting issues, an NN must change the weights enough to learn new tasks but not too much as to cause it to forget previously learned representations.

The simplest way to refine an NN is to add the new training data (*i.e.*, program branching behaviors) together with the old data and train the model from scratch again. However, as the number of data points grows, such retraining becomes harder to scale. Prior research has tried to solve this problem using mainly two broad approaches [64, 65, 66, 67, 68, 69, 70]. The first one tries to keep separate representations for the new and old models to minimize forgetting using distributed models, regularization, or creating an ensemble out of multiple models. The second approach maintains a summary of the old data and retrains the model on new data along with the summarized old data and therefore is more efficient than complete retraining. We refer the interested readers to the survey by Kemker et al. [71] for more details.

In this paper, we used edge-coverage-based filtering to only keep the old data that triggered new branches for retraining. As new training data becomes available, we identify the ones achieving new edge coverage, put them together with the filtered old training data, and retrain the NN. Such a

method effectively prevents the number of training data samples from drastically increasing over the number of retraining iterations. We find that our filtration scheme can easily support up to 50 iterations of retraining while still keeping the training time under several minutes.

2.5 Implementation

In this section, we discuss our implementation and how we fine-tune NEUZZ to achieve optimal performance. We have released our implementation through GitHub at `http://github.com/dongdongshe/neuzz`. All our measurements are performed on a system running Arch Linux 4.9.48 with an Nvidia GTX 1080 Ti GPU.

NN architecture. Our NN model is implemented in Keras-2.1.3 [72] with Tensorflow-1.4.1 [73] as a backend. The NN model consists of three fully-connected layers. The hidden layer uses ReLU as its activation function. We use sigmoid as the activation function for the output layer to predict whether a control flow edge is covered or not. The NN model is trained for 50 epochs (*i.e.*, 50 complete passes of the entire dataset) to achieve high test accuracy (around 95% on average). Since we use a simple feed-forward network, the training time for all 10 programs is less than 2 minutes. Even with pure CPU computation on an Intel i7-7700 running at 3.6GHz, the training time is under 20 minutes.

Training Data Collection. For each program tested, we run AFL-2.52b [4] on a single core machine for an hour to collect training data for the NN models. The average number of training inputs collected for 10 programs is around $2K$. The resulting corpus is further split into training and testing data with a 5:1 ratio, where the testing data is used to ensure that the models are not overfitting. We use `10KB` as the threshold file size for selecting our training data from the AFL input corpus (on average 90% of the files generated by AFL were under the threshold).

2.6 Evaluation

In this section, we discuss our implementation and how we fine-tune NEUZZ to achieve optimal performance. We have released our implementation through GitHub at `http://github.com/`

dongdongshe/neuzz. All our measurements are performed on a system running Arch Linux 4.9.48 with an Nvidia GTX 1080 Ti GPU.

NN architecture. Our NN model is implemented in Keras-2.1.3 [72] with Tensorflow-1.4.1 [73] as a backend. The NN model consists of three fully-connected layers. The hidden layer uses ReLU as its activation function. We use sigmoid as the activation function for the output layer to predict whether a control flow edge is covered or not. The NN model is trained for 50 epochs (*i.e.*, 50 complete passes of the entire dataset) to achieve high test accuracy (around 95% on average). Since we use a simple feed-forward network, the training time for all 10 programs is less than 2 minutes. Even with pure CPU computation on an Intel i7-7700 running at 3.6GHz, the training time is under 20 minutes.

Training Data Collection. For each program tested, we run AFL-2.52b [4] on a single core machine for an hour to collect training data for the NN models. The average number of training inputs collected for 10 programs is around $2K$. The resulting corpus is further split into training and testing data with a 5:1 ratio, where the testing data is used to ensure that the models are not overfitting. We use `10KB` as the threshold file size for selecting our training data from the AFL input corpus (on average 90% of the files generated by AFL were under the threshold).

2.6.1 Can NEUZZ find more bugs than existing fuzzers?

To answer this RQ, we evaluate NEUZZ *w.r.t.* other fuzzers in three settings: (i) Detecting real-world bugs. (ii) Detecting injected bugs in LAVA-M dataset [41]. (iii) Detecting CGC bugs. We describe the results in details.

(i) Detecting real-world bugs. We compare the total number of bugs and crashes found by NEUZZ and other fuzzers on 24-hour running time given the same seed corpus. There are five different types of bugs found by NEUZZ and other fuzzers: out-of-memory, memory leak, assertion crash, integer overflow, and heap overflow. To detect memory bugs that would not necessarily lead to a crash, we compile program binaries with AddressSanitizer [74]. We measure the unique memory bugs found by comparing the stack traces reported by AddressSanitizer. For crashes that do not

cause AddressSanitizer to generate a bug report, we examine the execution trace. The integer overflow bugs are found by manually analyzing the inputs that trigger an infinite loop. We further verify integer overflow bugs using undefined behavior sanitizer [75]. The results are summarized in Table 3.3.

NEUZZ finds all 5 types of bugs across 6 programs. AFL, AFLFast, and AFL-laf-intel find 3 types of bugs—they do not find any integer overflow bugs. The other fuzzers only uncover 2 types of bugs (*i.e.*, memory leak and assertion crash). AFL can a heap overflow bug on program `size`, while NEUZZ can find the same bug and another heap overflow bug on program `nm`. In total, NEUZZ finds 2× more bugs than the second best fuzzer. Moreover, the integer-overflow bug in `strip` and the heap-overflow bug in `nm`, *only found by* NEUZZ, have been assigned with CVE-2018-19932 and CVE-2018-19931, later fixed by the developers.

Table 2.1: **Number of real-world bugs found by 6 fuzzers. We only list the programs where the fuzzers find a bug.**

Programs	AFL	AFLFast	VUzzer	KleeFL	AFL-laf-intel	NEUZZ
Detected Bugs per Project						
readelf	4	5	5	3	4	16
nm	8	7	0	0	6	9
objdump	6	6	0	3	7	8
size	4	4	0	3	2	6
strip	7	5	2	5	7	20
libjpeg	0	0	0	0	0	1
Detected Bugs per Type						
out-of-memory	✓	✓	✗	✓	✓	✓
memory leak	✓	✓	✓	✓	✓	✓
assertion crash	✗	✓	✗	✗	✓	✓
interger overflow	✗	✗	✗	✗	✗	✓
heap overflow	✓	✗	✗	✗	✗	✓
Total	29	27	7	14	26	60

(ii) Detecting injected bugs in LAVA-M dataset. The LAVA dataset is created to evaluate the efficacy of fuzzers by providing a set of real-world programs injected with a large number of bugs [41]. LAVA-M is a subset of the LAVA dataset, consisting of 4 GNU coreutil programs

`base64`, `md5sum`, `uniq`, and `who` injected with 44, 57, 28, and 2136 bugs, respectively. All the bugs are guarded by four-byte magic number comparisons. The bugs get triggered only if the condition is satisfied. We compare NEUZZ's performance at finding these bugs to other state-of-the-art fuzzers, as shown in 2.2. Following conventional practice [7, 41], we use 5-hour time budget for the fuzzers' runtime.

Triggering a magic number condition in the LAVA dataset is a hard task for a coverage-guided fuzzer because the fuzzer has to generate the exact combination of 4 continuous bytes out of 256^4 possible cases. To solve this problem, we used a customized LLVM pass to instrument the magic byte checks like Steelix [45]. But unlike Steelix, we leverage the NN's gradient to guide the input generation process to find an input that satisfies the magic check. We run AFL for an hour to generate the training data and use it to train an NN whose gradients identify the possible critical bytes triggering the first byte-comparison of a magic-byte condition. Next, we perform a locally exhaustive search on each byte adjacent to the first critical byte to solve each of the remaining three byte-comparisons with 256 tries. Therefore, we need one NN gradient computation to find the byte locations that affect the magic checking and $4 \times 256 = 1024$ trials to trigger each bug. For program `md5sum`, following the latest suggestion of the LAVA-M's authors [76], we further reduce the seed into a single line, which significantly boosts the fuzzing performance.

As shown in Table 2.2, NEUZZ finds all the bugs in programs `base64`, `md5sum`, and `uniq`, and the highest number of bugs for program `who`. Note that LAVA-M authors left some bugs unlisted in all 4 programs, so the total number of bugs found by NEUZZ is actually higher than the number of listed bugs, as shown in the result.

NEUZZ has two key advantages over the other fuzzers. First, NEUZZ breaks the search space into multiple manageable steps: NEUZZ trains the underlying NN on AFL generated data, uses the computed gradient to reach the first critical byte, and performs a local search around the found critical region. Second, as opposed to VUzzer, which leverages magic numbers hard-coded in the target binary to construct program inputs, NEUZZ's gradient-based searching strategy do not rely on any hard-coded magic number. Thus, it can find all the bugs in program `md5sum`, which performs

some computations on the input bytes before the magic number checking causing VUzzer to fail. In comparison to Angora, the current state-of-the-art fuzzer for LAVA-M dataset, NEUZZ finds 3 more bugs in `md5sum`. Unlike Angora, NEUZZ uses NN gradients to trigger the complex magic number conditions more efficiently.

Table 2.2: **Bugs found by different fuzzers on LAVA-M datasets.**

	base64	md5sum	uniq	who
#Bugs	44	57	28	2,136
FUZZER	7	2	7	0
SES	9	0	0	18
VUzzer	17	1	27	50
Steelix	43	28	24	194
Angora	48	57	29	1,541
AFL-laf-intel	42	49	24	17
T-fuzz	43	49	26	63
NEUZZ	48	60	29	1,582

(iii) Detecting CGC bugs. The DARPA CGC dataset [77] consists of vulnerable programs used in the DARPA Cyber Grand Challenge. These programs are implemented as network services performing various tasks and aim to mirror real-world applications with known vulnerabilities. Every bug in the program is guarded by a number of sanity checks on the input. The dataset comes with a set of inputs as proof of vulnerabilities.

We evaluate NEUZZ, Driller, and AFL on 50 randomly chosen CGC binaries. As running each test binary for each fuzzer takes 6 hours to run on CPU/GPU and our limited GPU resources do not allow us to execute multiple instances in parallel, we randomly picked 50 programs to keep the total experiment time within reasonable bounds. Similar to LAVA-M, here we also run AFL for an hour to generate the training data and use it to train the NN. We provide the same random seed to all three fuzzers and let them run for six hours. NEUZZ uses the same customized LLVM pass used for the LAVA-M dataset to instrument magic checkings in CGC binaries.

The results (Table 2.3) show that NEUZZ uncovers 31 buggy binaries out of 50 binaries, while AFL and Driller find 21 and 25, respectively. The buggy binaries found by NEUZZ include all those

Table 2.3: **Bugs found by 3 fuzzers in 50 CGC binaries**

Fuzzers	AFL	Driller	NEUZZ
Bugs	21	25	31

found by Driller and AFL. NEUZZ further found bugs in 6 new binaries that both AFL and Driller fail to detect.

```
int cgc_ReceiveCommand(CommandStruct* command,
  int* more_command){
  ...
  if(cgc_strncmp(&buffer[1], "VISUALIZE",
    cgc_strlen("VISUALIZE")) == 0){
    command->command = VISUALIZE;
    //vulnerable code
  ...
```

Listing 2.1: **`cgc_ReceiveCommand` function in CROMU_00027**

We analyze an example program `CROMU_00027` (shown in Listing 2.6.1). This is an ASCII content server that takes a query from a client and serves the corresponding ASCII code. A null-pointer dereferencing bug is triggered after a user tries to set command as VISUALIZE. AFL failed to detect this bug within 6-hour time budget due to its inefficiency at guessing the magic string. Although Driller tries to satisfy such complex magic string checking by concolic execution, in this case it fails to find an input that satisfies the check. By contrast, NEUZZ can easily use the NN gradient to locate the critical bytes in the program input that affects the magic comparison and find inputs that satisfy the magic check.

2.6.2 Can NEUZZ achieve higher edge coverage than existing fuzzers?

To investigate this question, we compare the fuzzers on 24-hour fixed runtime budget. This evaluation shows not only the total number of new edges found by fuzzers but also the speed of new edge coverage versus time.

We collect the edge coverage information from AFL's edge coverage report. The results are summarized in Table 2.4. For all 10 real-world programs, NEUZZ significantly outperforms other

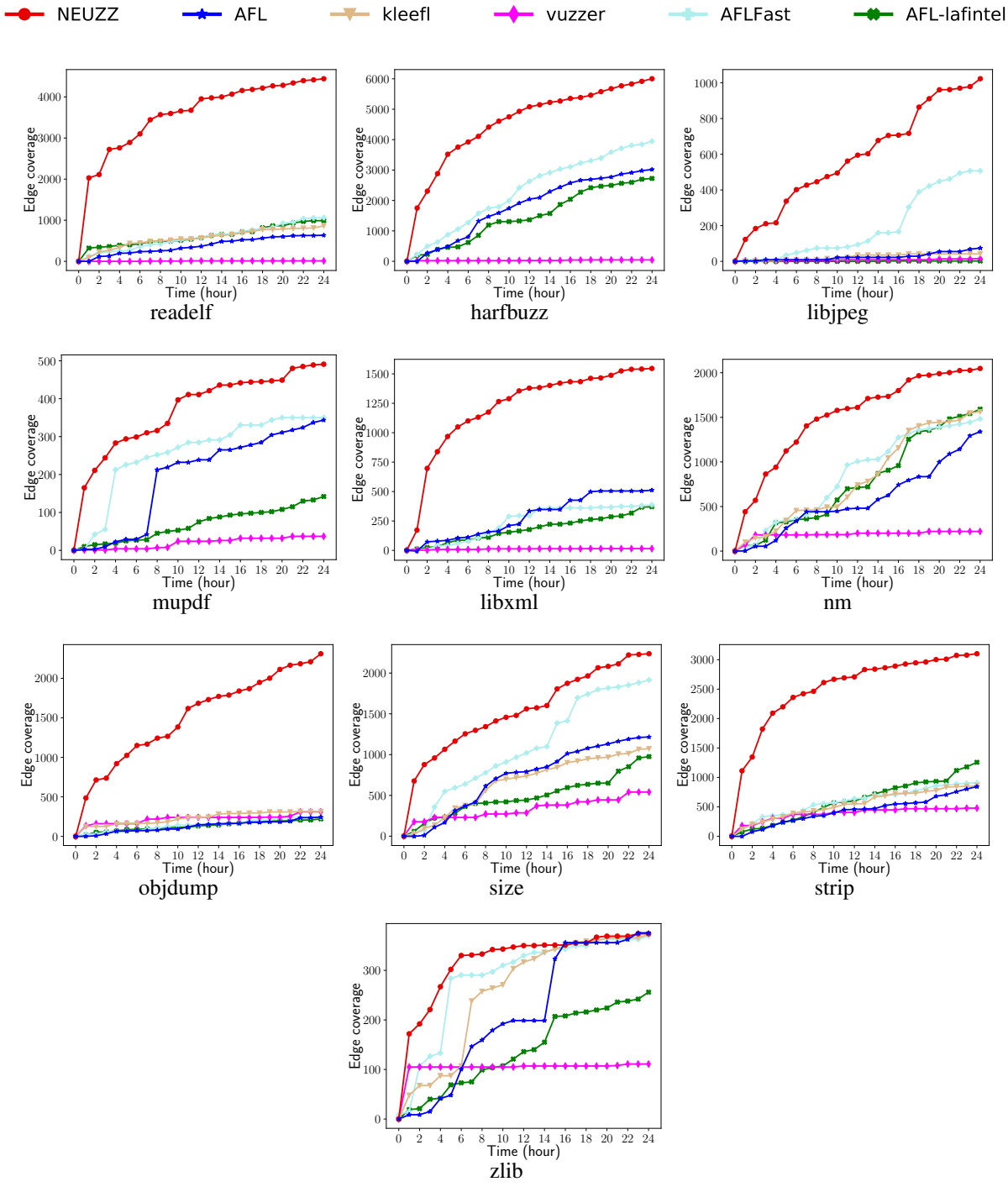


Figure 2.4: **The edge coverage of different fuzzers running for 24 hours.**

Table 2.4: **Comparing edge coverage of NEUZZ *w.r.t.* other fuzzers for 24 hours runs.**

Programs	**NEUZZ**	**AFL**	**AFLFast**	**VUzzer**	**KleeFL**	**AFL-laf-intel**
readelf -a	4,942	746	1,073	12	968	1,023
nm -C	2,056	1,418	1,503	221	1,614	1,445
objdump -D	2,318	257	263	307	328	221
size	2,262	1,236	1,924	541	1,091	976
strip	3,177	856	960	478	869	1,257
libjpeg	1,022	94	651	60	67	2
libxml	1,596	517	392	16	n/a†	370
mupdf	487	370	371	38	n/a	142
zlib	376	374	371	15	362	256
harfbuzz	6,081	3,255	4,021	111	n/a	2,724

†indicates cases where Klee failed to run due to external dependencies

fuzzers in terms of edge coverage. As shown in Fig 2.4, NEUZZ can achieve significantly more new edge coverage than other fuzzers within the first hour. On programs `strip`, `harfbuz` and `readelf`, NEUZZ can achieve more than $1,000$ new edge coverage *within an hour*. For programs `readelf` and `objdump`, the number of new edge coverage from NEUZZ's 1 hour running even beats the numbers of new edge coverage from all other fuzzers' 24 hours running. This shows the superior edge coverage ability of NEUZZ. For all 9 out of 10 programs, NEUZZ achieves 6×,1.5×,9×,1.8×,3.7×,1.9×,10×,1.3× and 3× edge coverage than baseline AFL, respectively, and 4.2×,1.3×,7×,1.2×,2.5×,1.5×,1.5×,1.3× and 3× edge coverage than the second highest number among all 6 fuzzers. For the smallest program `zlib`, which has less than 2k lines of code, NEUZZ achieves similar edge coverage with other fuzzers. We believe it reaches a saturation point when most of the possible edges for such a small program are already discovered after 24 hours fuzzing. The significant outperformance shows the effectiveness of NEUZZ in efficiently locating and mutating critical bytes using the gradient to cover new edges. NEUZZ also scales well in large systems. In fact, for programs with more than 10K lines (*e.g.*, `readelf`, `harfbuzz`, `mupdf` and `libxml`), NEUZZ achieves the highest edge coverage, where the taint-assisted fuzzer (*i.e.*, VUzzer) and symbolic execution assisted fuzzer (*i.e.*, KleeFL) either perform badly or does not scale.

The gradient-guided mutation strategy allows NEUZZ to explore diverse edges, while other evolutionary-based fuzzers often get stuck and repetitively check the same branch conditions. Also,

the minimal execution overhead of the NN smoothing technique helps NEUZZ to scale well for larger programs while other advanced evolutionary fuzzers incur high execution overhead due to the use of heavyweight program analysis techniques like taint-tracking or symbolic execution.

Among the evolutionary fuzzers, AFLFast, uses an optimized seed selection strategies that focuses more on rare edges and thus achieves higher coverage than AFL on 8 programs, especially in `libjpeg`, `size` and `harfbuzz`. VUzzer, on the other hand, achieves higher coverage than AFL, AFLFast, and AFL-laf-intel within the first hour on small programs (*e.g.*, `zlib`, `nm`, `objdump`, `size` and `strip`), but its lead stalls quickly and eventually is surpassed by other fuzzers. Meanwhile, VUzzer's performance degrades on larger programs like `readelf`, `harfbuzz`, `libxml`, and `mupdf`. We suspect that the imprecisions introduced by VUzzer's taint tracker causes it to perform poorly on large programs. KleeFL uses additional seeds generated by the symbolic execution engine Klee to guide AFL's exploration. Similar to VUzzer, for small programs (`nm`, `objdump`, and `strip`), KleeFL has good performance at the beginning, but its advantage of additional seeds from Klee fade away after several hours. Moreover, KleeFL is based on Klee that cannot scale to large programs with complex library code, a well-known limitation of symbolic execution. Thus, KleeFL does not have results on programs `libxml`, `mupdf` and `harfbuzz`. Unlike VUzzer and KleeFL, NEUZZ does not rely on any heavy program analysis techniques; NEUZZ uses the gradients computed from NNs to generate promising mutations even for larger programs. The efficient NN gradient computation process allow NEUZZ to scale better than VUzzer and KleeFL at identifying the critical bytes that affect different unseen program branches, achieving significantly more edge coverage.

AFL-laf-intel transforms complex magic number comparison into nested byte-comparison using an LLVM pass and then runs AFL on the transformed binaries. It achieves second-highest new edge coverage on program `strip`. However, the comparison transformations add additional instructions to common comparison operations and thus cause a potential edge explosion issue. The edge explosion greatly increases the rate of edge conflict and hurt the performance of evolutionary fuzzing. Also, these additional instructions cause extra execution overheads. As a result, programs

like `libjpeg` with frequent comparison operations suffer significant slowdown (*e.g.*, `libjpeg`), and AFL-laf-intel struggles to trigger new edges.

2.6.3 Can NEUZZ perform better than existing RNN-based fuzzers?

Existing recurrent neural network (RNN)-based fuzzers learn mutation patterns from past fuzzing experience to guide future mutations [78]. These models first learn mutation patterns (composed of critical bytes) from a large number of mutated inputs generated by AFL. Next, they use the mutation patterns to build a filter to AFL which only allows mutations on critical bytes to pass, vetoing all other non-critical byte mutations. We choose 4 programs studied by the previous work to evaluate the performance of NEUZZ compared to the RNN-based fuzzer for 1 million mutations. We train two NN models with the same training data, then let the two NN-based fuzzers run to generate 1 million mutations and compare the new code coverage achieved by the two methods. We report both the achieved edge coverage and training time, as shown in 2.5.

Table 2.5: **NEUZZ vs. RNN fuzzer *w.r.t.* baseline AFL**

Programs	**Edge Coverage**			**Training Time (sec)**		
	NEUZZ	RNN	AFL	NEUZZ	RNN	AFL
readelf -a	1,800	215	213	108	2,224	NA
libjpeg	89	21	28	56	1,028	NA
libxml	256	38	19	95	2,642	NA
mupdf	260	70	32	62	848	NA

For all the four programs, NEUZZ significantly outperforms the RNN-based fuzzer on 1M mutations. NEUZZ achieves 8.4×, 4.2×, 6.7×, and 3.7× more edge-coverage than the RNN-based fuzzer across the four programs respectively. In addition, the RNN-based fuzzer has, on average, 20× more training overhead than NEUZZ, because RNN models are significantly more complicated than feed-forward network models.

An additional comparison of the RNN-based fuzzer with AFL shows that the former achieves 2× more edge coverage on average than AFL on `libxml` and `mupdf` using the 1-hour corpus. We also observe that the RNN-based fuzzer vetoes around 50% of the mutations generated by AFL. Thus,

the new edge coverage of 1M mutations from RNN-based fuzzer can achieve the edge coverage of 2M mutations in vanilla AFL. This explains why the RNN-based fuzzer uncovers around 2× more new edges of AFL on some programs. If AFL gets stuck after 2M mutations, the RNN-based fuzzer would also get stuck after 1M filtered mutations. The key advantage of NEUZZ over the RNN-based fuzzer is that NEUZZ obtains critical locations using neural-network-based gradient-guided search, while the RNN fuzzer tries to model the task in an end-to-end manner. Our model can distinguish different contributing factors of critical bytes that the RNN model may miss as demonstrated by our experimental results. For mutation generation, we perform an exhaustive search for critical bytes determined by corresponding contributing factors, while the RNN-based fuzzer still relies on AFL's uniform random mutations.

NEUZZ, a fuzzer based on simple feed-forward network, significantly outperforms the RNN-based fuzzers by achieving 3.7× to 8.4× more edge coverage across different projects.

2.6.4 How do different model choices affect NEUZZ's performance?

NEUZZ's fuzzing performance heavily depends on the accuracy of the trained NN. As described in Section 2.5, we empirically find that an NN model with 1 hidden layer is expressive enough to model complex branching behavior of real-world programs. In this section, we conduct an ablation study by exploring different model settings for a 1 hidden layer architecture, *i.e.*, a linear model, an NN model without refinement, and an NN model with incremental refinement. We evaluate the effect of these models on NEUZZ's performance.

To compare the fuzzing performance, we generate 1M mutations for each version of NEUZZ on 4 programs. We implement the linear model by removing the non-linear activation functions used in the hidden layer and thus making the whole feed-forward network completely linear. The NN model is trained same seed corpus from AFL. Next, We generate 1M mutations from the passive learning model and measure the edge coverage achieved by these 1M mutations. Finally, we filter out the mutated inputs that exercise unseen edges from the 1 million mutations and add these selected inputs to original seed corpus to incrementally retrain another NN model and use it to generate

Table 2.6: **Edge coverage comparison of 1M mutations generated by NEUZZ using different machine learning models.**

Programs	Linear Model	NN Model	NN + Incremental
readelf -a	1,723	1,800	2,020
libjpeg	63	89	159
libxml	117	256	297
mupdf	93	260	329

further mutations. The results are summarized in Table 2.6. We can see that both NN models (with or without incremental learning) outperform the linear models for all 4 tested programs. This shows that the nonlinear NN models can approximate program behaviors better than a simple linear model. We also observe that incremental learning helps NNs to achieve significantly higher accuracy and therefore higher edge coverage.

NN models outperform linear models and incremental learning makes NNs even more accurate over time.

2.7 Conclusion

We present NEUZZ, an efficient learning-enabled fuzzer that uses a surrogate neural network to smoothly approximate a target program's branch behavior. We further demonstrate how gradient-guided techniques can be used to generate new test inputs that can uncover different bugs in the target program. Our extensive evaluations show that NEUZZ significantly outperforms other 10 state-of-the-art fuzzers both in the numbers of detected bugs and achieved edge coverage. Our results demonstrate the vast potential of leveraging different gradient-guided input generation techniques together with neural smoothing to significantly improve the effectiveness of the fuzzing process.

Chapter 3: MTFuzz: Fuzzing with a Multi-Task Neural Network

Fuzzing is a widely used technique for detecting software bugs and vulnerabilities. Most popular fuzzers generate new inputs using an evolutionary search to maximize code coverage. Essentially, these fuzzers start with a set of seed inputs, mutate them to generate new inputs, and identify the promising inputs using an evolutionary fitness function for further mutation.

Despite their success, evolutionary fuzzers tend to get stuck in long sequences of unproductive mutations. In recent years, machine learning (ML) based mutation strategies have reported promising results. However, the existing ML-based fuzzers are limited by the lack of quality and diversity of the training data. As the input space of the target programs is high dimensional and sparse, it is prohibitively expensive to collect many diverse samples demonstrating successful and unsuccessful mutations to train the model.

In this paper, we address these issues by using a Multi-Task Neural Network that can learn a compact embedding of the input space based on diverse training samples for multiple related tasks (i.e., predicting different types of coverage). The compact embedding can guide the mutation process by focusing most of the mutations on the parts of the embedding where the gradient is high. MTFUZZ uncovers 11 previously unseen bugs and achieves an average of 2× more edge coverage compared with 5 state-of-the-art fuzzer on 10 real-world programs.

3.1 Introduction

Coverage-guided graybox fuzzing is a widely used technique for detecting bugs and security vulnerabilities in real-world software [79, 14, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89]. The key idea behind a fuzzer is to execute the target program on a large number of automatically generated test inputs and monitor the corresponding executions for buggy behaviors. However, as the input spaces

of real-world programs are typically very large, unguided test input generation is not effective at finding bugs. Therefore, most popular graybox fuzzers use evolutionary search to generate new inputs; they mutate a set of seed inputs and retain only the most promising inputs (*i.e.*, inputs exercising new program behavior) for further mutations [79, 81, 82, 83, 90, 45, 91, 84, 92].

However, the effectiveness of traditional evolutionary fuzzers tends to decrease significantly over fuzzing time. They often get stuck in long sequences of unfruitful mutations, failing to generate inputs that explore new regions of the target program [93, 85, 94]. Several researchers have worked on designing different mutation strategies based on various program behaviors (e.g., focusing on rare branches, call context, etc.) [84, 93]. However, program behavior changes drastically, not only across different programs but also across different parts of the same program. Thus, finding a generic robust mutation strategy still remains an important open problem.

Recently, Machine Learning (ML) techniques have shown initial promise to guide the mutations [95, 85, 96]. These fuzzers typically use existing test inputs to train ML models and learn to identify promising mutation regions that improve coverage [85, 91, 96, 95]. Like any other supervised learning technique, the success of these models relies heavily on the number and diversity of training samples. However, collecting such training data for fuzzing that can demonstrate successful/unsuccessful mutations is prohibitively expensive due to two main reasons. First, successful mutations that increase coverage are often limited to very few, sparsely distributed input bytes, commonly known as hot bytes, in high-dimensional input space. Without knowing the distribution of hot bytes, it is extremely hard to generate successful mutations over the sparse, high-dimensional input space [85, 95]. Second, the training data must be diverse enough to expose the model to various program behaviors that lead to successful/unsuccessful mutations—this is also challenging as one would require a large number of test cases exploring different program semantics. Thus, the ML-based fuzzers suffer from both *sparsity* and *lack of diversity* of the target domain.

In this paper, we address these problems using Multi-Task Learning, a popular learning paradigm used in domains like computer vision to effectively learn common features shared across related tasks from limited training data. In this framework, different participating tasks allow an ML

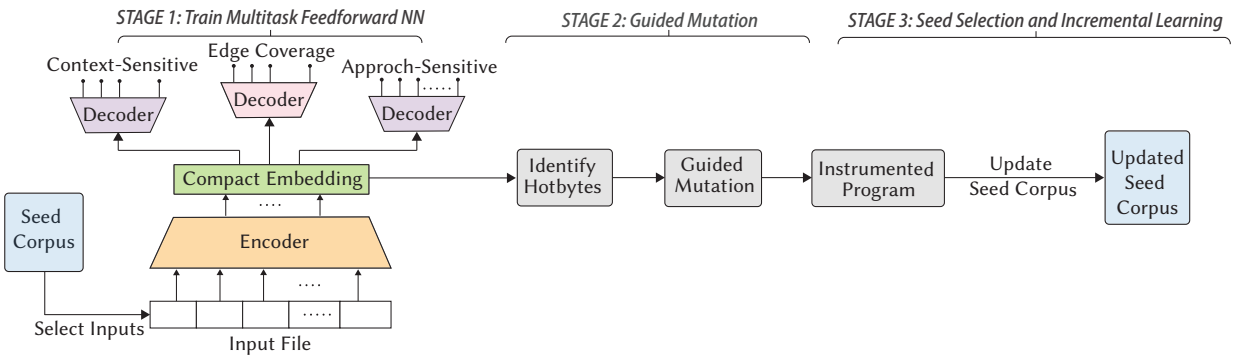


Figure 3.1: **Overview of MTFuzz**

model to effectively learn a compact and more generalized feature representation while ignoring task-specific noises. To jointly learn a compact embedding of the inputs, in our setting, we use different tasks for predicting the relationship between program inputs and different aspects of fuzzing-related program behavior (e.g., different types of edge coverage). Such an architecture addresses both the data sparsity and lack of diversity problem. The model can simultaneously learn from diverse program behaviors from different tasks as well as focus on learning the important features (hot bytes in our case) across all tasks. Each participating task will provide separate pieces of evidence for the relevance or irrelevance of the input features [97].

To this end, we design, implement, and evaluate MTFuzz, a Multi-task Neural Network (MTNN) based fuzzing framework. Given the same set of test inputs, MTFuzz learns to predict three different code coverage measures showing various aspects of dynamic program behavior:

1. edge coverage: which edges are explored by a test input [79, 85]?

2. approach-sensitive edge coverage: if an edge is not explored, how far off it is (i.e., approach level) from getting triggered [98, 99, 100, 101]?

3. context-sensitive edge coverage: from which call context an explored edge is called [93, 102]?

Note that our primary task, like most popular fuzzers, is to increase edge coverage. However, the use of call context and approach level provides additional information to boost edge coverage.

Architecturally, the underlying MTNN contains a group of hidden layers shared across the participating tasks, while still maintaining task-specific output layers. The last shared layer learns a *compact embedding* of the input space as shown in Figure 3.1. Such an embedding captures a generic compressed representation of the inputs while preserving the important features, *i.e.*, hot-byte distribution. We compute a saliency score [94] of each input byte by computing the gradients of the embedded representation *w.r.t.* the input bytes. Saliency scores are often used in computer vision models to identify the important features by analyzing the importance of that feature *w.r.t.* an embedded layer [57]. By contrast, in this paper, we use such saliency scores to guide the mutation process—focus the mutations on bytes with high saliency scores.

Our MTNN architecture also allows the compact embedding layer, once trained, to be transferred across different programs that operate on similar input formats. For example, compact-embedding learned with MTFUZZ for one `xml` parser may be transferred to other `xml` parsers. Our results (in RQ4) show that such transfer is quite effective and it reduces the cost to generate high quality data from scratch on new programs which can be quite expensive. Our tool is available at https://git.io/JUWkj and the artifacts available at doi.org/10.5281/zenodo.3903818.

We evaluate MTFUZZ on 10 real world programs against 5 state-of-the-art fuzzers. MTFUZZ covers at least 1000 more edges on 5 programs and several 100 more on the rest. MTFUZZ also finds a total of 71 real-world bugs (11 previously unseen) (see RQ1). When compared to learning each task individually, MTFUZZ offers significantly more edge coverage (see RQ2). Lastly, our results from transfer learning show that the compact-embedding of MTFUZZ can be transferred across parsers for xml and elf binaries.

3.2 Background: Multi-Task Networks

Multi-task Neural Networks (MTNN) are becoming increasingly popular in many different domains including optimization [103, 104], natural language processing [105, 106], and computer vision [107]. The key intuition behind MTNN is that it is useful for related tasks to be learned jointly so that each task can benefit from the relevant information available in other tasks [108, 109,

107, 110]. For example, if we learn to ride a unicycle, a bicycle, and a tricycle simultaneously, experiences gathered from one usually help us to learn the other tasks better [111]. In this paper, we use a popular MTNN architecture called hard parameter sharing [109], which contains two groups of layers (see Fig. 3.5): a set of initial layers shared among all the tasks, and several individual task-specific output layers. The shared layers enable a MTNN to find a *common feature representation* across all the tasks. The task-specific layers use the shared feature representation to generate predictions for the individual tasks [112, 107, 97].

MTNN Training. While MTNNs can be used in many different ML paradigms, in this paper we primarily focus on supervised learning. We assume that the training process has access to a training dataset $\mathcal{X} = \{x_1, x_2, ..., x_n\}$. The training data contains the ground truth output labels for each task. We train the MTNN on the training data using standard back-propagation to minimize a multi-task loss.

Multi-task Loss. An MTNN is trained using a multi-task loss function, $\mathcal{L}$. We assume that each individual task τ_i in the set of tasks $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_m\}$ has a corresponding loss function $\mathcal{L}_i$. The multi-task loss is computed as a weighted sum of each individual task loss. More formally, it is given by $\mathcal{L} = \sum_{i=1}^{m} \alpha_i \cdot \mathcal{L}_i$. Here, α_i represents the weight assigned to task i. The goal of training is to reduce the overall loss. In practice, the actual values of the weights are decided based on the relative importance of each task. Most existing works assign equal weights to the tasks [113, 114, 115].

The multi-task loss function forces the shared layer to learn a general input representation for all tasks offering two benefits:

1) Increased generalizibility. The overall risk of overfitting in multi-task models is reduced by an order of m (where m is the number of tasks) compared to single task models [116]. Intuitively, the more tasks an MTNN learns from, the more general the compact representation is in capturing features of all the tasks. This prevents the representation from overfitting to the task-specific features.

2) Reduced sparsity. The shared embedding layer in an MTNN can be designed to increase the

compactness of the learned input representation. Compared with original input layer, a shared embedding layer can achieve same expressiveness on a given set of tasks while requiring far fewer nodes. In such compact embedding, the important features across different tasks will be boosted with each task contributing its own set of relevant features [97].

3.3 Methodology

This section presents a brief overview of MTFUZZ that aims to maximize edge coverage with the aid of two additional coverage measures: context-sensitive edge coverage and approach-sensitive edge coverage using multi-task learning. 3.1 illustrates an end-to-end workflow of the proposed approach. The first stage trains an MTNN to produce a compact embedding of an otherwise sparse input space while preserving information about the hot bytes *i.e.*, the input bytes have the highest likelihood to impact code coverage (3.3.2). The second stage identifies these hot bytes and focuses on mutating them (3.3.3). Finally, in the third stage, the seed corpus is updated with the mutated inputs and retains only the most interesting new inputs (3.3.4).

3.3.1 Modeling Coverage as Multiple Tasks

The goal of any ML-based fuzzers, including MTFUZZ, is to learn a mapping between input space and code coverage. The most common coverage explored in the literature is edge coverage, which is an effective measure and quite easy to instrument. However, it is coarse-grained and misses many interesting program behavior (*e.g.*, explored call context) that are known to be important to fuzzing. One workaround is to model path coverage by tracking the program execution path per input. However, keeping track of all the explored paths can be computationally intractable since it can quickly lead to a state-space explosion on large programs [102]. As an alternative, in this work, we propose a middle ground: we model the edge coverage as the primary task of the MTNN, while choosing two other fine-granular coverage metrics (approach-sensitive edge coverage and context-sensitive edge coverage) as auxiliary tasks to provide useful additional context to edge coverage.

Edge Coverage: Primary Task.

Edge coverage measures how many unique control-flow edges are triggered by a test input as it interacts with the program. It has become the de-facto code coverage metric [79, 85, 84, 83] for fuzzing. We model edge coverage prediction as the *primary task* of our multi-task network, which takes a binary test case as input and predicts the edges that could be covered by the test case. For each input, we represent the edge coverage as an *edge bitmap*, where value per edge is set to `1` or `0` depending on whether the edge is exercised by the input or not.

In particular, in the control-flow-graph of a program, an edge connects two basic blocks (denoted by `prev_block` and `cur_block`) [79]. A unique *edge_id* is obtained as: *hash*(`prev_block`, `cur_block`). For each *edge_id*, there is a bit allocated in the bitmap. For every input, the edge_ids in the corresponding edge bitmap are set to `1` or `0`, depending on whether or not those edges were triggered.

Approach-Sensitive Edge Coverage: Auxiliary Task 1.

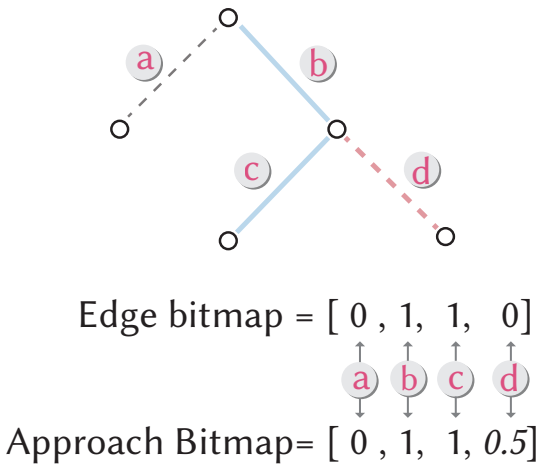


Figure 3.2: **Approach Bitmap vs. Edge Bitmap. The edge 'd' has a visited parent edge 'b' and is thus marked 0.5 in the approach bitmap.**

For an edge that is not exercised by an input, we measure how far off the edge is from getting triggered. Such a measure provides additional contextual information of an edge. For example, if two test inputs failed to trigger an edge, however one input reached "closer" to the unexplored edge than the other, traditional edge coverage would treat both inputs the same. However, using

a proximity measure, we can discern between the two inputs and mutate the closer input so that it can reach the unexplored edge. To achieve this, approach-sensitive edge coverage extends edge coverage by offering a distance measure that computes the distance between an unreached edge and the nearest edge triggered by an input. This is a popular measure in the search-based software engineering literature [98, 100, 99], where instead of assigning a binary value (0 or 1), as in edge bitmap, *approach level* assigns a numeric value between 0 and 1 to represent the edges [117]; if an edge is triggered, it is assigned `1`. However, if the edge is *not* triggered, but one of its parents are triggered, then the non-triggered edge is assigned a value of β (we use $\beta = 0.5$). If neither the edge nor any of its parents are triggered, it is assigned 0. This is illustrated in Fig. 3.2. Note that, for a given edge, we refrain from using additional ancestors farther up the control-flow graph to limit the computational burden. The approach sensitive coverage is represented in an *approach bitmap*, where for every unique $edge\_id$, we set an approach level value, as shown in Fig. 3.2. We model this metric in our Multi-task Neural Network as an auxiliary task, where the task takes binary test cases as inputs and learn to predict the corresponding approach-level bitmaps.

Context-sensitive Edge Coverage: Auxiliary Task 2.

Edge coverage cannot distinguish between two different test inputs triggering the same edge, but via completely different internal states (*e.g.*, through the same function called from different sites in the program). This distinction is valuable since reaching an edge via a new internal state (*e.g.*, through a new function call site) may trigger a vulnerability hidden deep within the program logic. Augmenting edge coverage with context information regarding internal states of the program may help alleviate this problem [93].

Consider the example in Fig. 3.3. Here, for an input `[1, 0]`, the first call to function `foo()` appears at site `line 12` and it triggers the if condition (on `line 2`); the second call to `foo()` appears on site `line 14` and it triggers the else condition (on `line 5`). As far as edge coverage is concerned, both the edges of the function `foo()` (on lines 2 and 5) have been explored and any additional inputs will remain uninteresting. However, if we provide a new input say `[0, 8]`, we

```
1  void foo(char* addr, int a) {
2      if(a > 0){
3          strncpy(addr, "I might overflow.", a+4);
4          return;
5      }else
6          return;
7  }
8  int main(int argc, char** input)
9  {
10     char buf[8];
11     ...
12     foo(buf, input[0]);
13     ...
14     foo(buf, input[1]);
15     ...
16 }
```

Figure 3.3: **An example C-code to demonstrate the usefulness of using context-sensitive measures. Measures such as edge coverage will fail to detect a possible bug in `strncpy(·)`**

would first trigger `line 5` of `foo` when it is called from `line 12`. Then we trigger `line 2` of `foo` from `line 14` and further cause a buffer overflow at `line 3` because a 12 bytes string is written into a 8 bytes destination buffer `buf`. Moveover, the input `[0, 8]` will not be saved by edge coverage fuzzer since it triggers no new edges. Frequently called functions (like `strcmp`) may be quite susceptible such crashes [102].

Input	Call Ctx	Edge Coverage
[1, 0]	(L12, L2, L3)	(L2, L3)
	(L14, L5, L6)	(L5, L6)
[0, 8]	(L12, L5, L6)	(L5, L6)
	(L14, L2, L3)	(L2, L3)

Figure 3.4: **The tuple in edge coverage does not differentiate between the clean input and the buggy input, while of context-sensitive edge coverage (labeled 'Call Ctx') does.**

In order to overcome this challenge, Chen *et al.* [93] propose keeping track of the *call stack* in addition to the edge coverage by maintaining tuple: $(call\_stack, prev\_block, cur\_block)$. Fig. 3.4 shows the additional information provided by context-sensitive edge coverage over edge coverage.

Here, we see an example where a buggy input `[0,8]` has the exact same edge coverage as the clean input `[1,0]`. However, the call context information can differentiate these two inputs based on the call stacks at `line 12` and `14`.

We model context-sensitive edge coverage in our framework as an auxiliary task. We first assign a unique id to every call. Next, at run time, when we encounter a call at an edge ($edge\_id$), we first compute a $hash$ to record all the functions on current call stack as: $call\_stack = call\_id_1 \oplus ... \oplus call\_id_n$,

where $call\_id\_i$ represents the i-th function on current call stack and $\oplus$ denotes XOR operation. Next, we compute the *context sensitive edge id* as: $call\_trace\_id = call\_stack \oplus edge\_id$.

Thus we obtain a unique $call\_trace\_id$ for every function called from different contexts (*i.e.*, call sites). We then create a bit-map of all the $call_trace_id$s. Unlike existing implementations of context-sensitive edge coverage [93, 102], we assign an additional id to each call instruction while maintaining the original $edge\_id$ intact. Thus, the total number of elements in our bit map reduces to sum of $call_trace_id$s and $edge_id$s rather than a product of $call_trace_id$s and $edge_id$s. An advantage of our design is that we minimize the bitmap size. In practice, existing methods requires around 7× larger bitmap size than just edge coverage [93]; our implementation only requires around 1.3× bitmap size of edge coverage. The smaller bitmap size can avoid edge explosion and improve performance.

In our multi-tasking framework, the context-sensitive edge coverage "task" is trained to predict the mapping between the inputs and the corresponding $call_trace_id$s bitmaps. This can enable us to learn the difference between two inputs in a more granular fashion. For example, an ML model can learn that under certain circumstances, the second input byte (`input[1]` in Fig. 3.3) can cause crashes. This information cannot be learned by training to predict for edge coverage alone since both inputs will have the same edge coverage (as shown in Fig. 3.4).

3.3.2 Stage-I: Multi-Task Training

This phase builds a multi-task neural network (MTNN) that can predict different types of edge coverage given a test input. The trained model is designed to produce a more general and compact embedding of the input space focusing only on those input bytes that are most relevant to all the tasks. This compact representation will be reused by the subsequent stages of the program to identify the most important bytes in the input (*i.e.*, the hot-bytes) and guide mutations on those bytes.

Architecture.

Fig. 3.5 shows the architecture of the MTNN. The model contains an *encoder* (shared among all the tasks) and three task-specific *decoders*. The model takes existing test input bytes as input and outputs task-specific bitmap. Each input byte corresponds to one input node, and each bitmap value corresponds to an output node.

- **Encoder.** Comprises of one input layer followed by three progressively narrower intermediate layers. The total number of nodes in the input layer is equal to the total number of bytes in the largest input in the seed corpus. All shorter inputs are padded with `0x00` for consistency. The last layer of the encoder is a compact representation of the input to be used by all the tasks (green in Fig. 3.5).

- **Decoders.** There are three task-specific decoders (shown in lilac in Fig. 3.5). Each task specific decoder consists of three intermediate layers that grow progressively wider. The last layer of each of the decoder is the output layer. For edge coverage, there is one node in the output layer for each unique $edge\_id$, likewise for context-sensitive edge coverage there is one output node for each $call\_trace\_id$, and for approach-sensitive edge coverage there is one output node for each unique $edge\_id$ but they take continuous values (see Fig. 3.2).

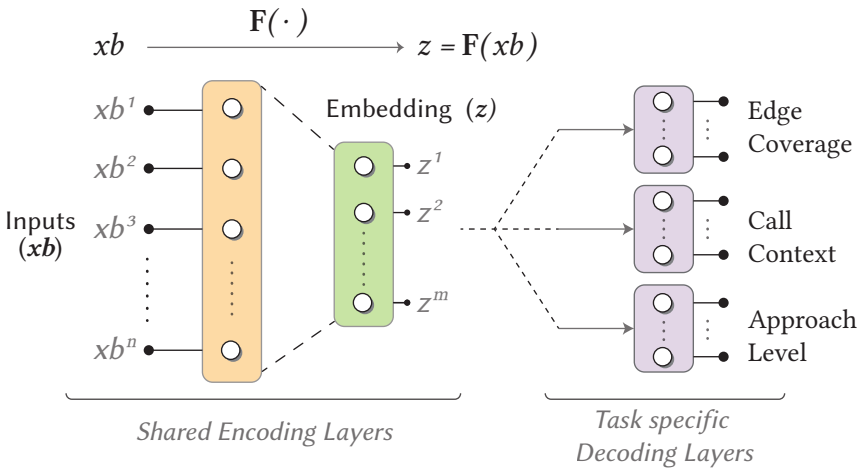


Figure 3.5: **The MTNN architecture representing the n-dimensional input layer $\mathbf{xb^i} \in \mathbf{xb}$; m-dimensional compact embedding layer $\mathbf{z^j} \in \mathbf{z}$, s.t. $\mathbf{m} < \mathbf{n}$, with a function $\mathbf{F}(\cdot)$ to map input xb and the embedding layer xb; three task-specific layers .**

Loss Functions.

The loss function of a MTNN is a weighted sum of the task-specific loss functions. Among our three tasks, edge coverage and context-sensitive edge coverage are modeled as classification tasks and approach-sensitive edge coverage is modeled as a regression task. Their loss functions are designed accordingly.

Loss function for approach-sensitive edge coverage. Approach-level measures how close an input was from an edge that was not triggered. This distance is measured using a *continuous value* between 0 and 1. Therefore, this is a regression problem and we use *mean squared error loss*, given by:

$$\mathcal{L}_{\tau_{\text{approach}}} = \text{MSE} = \frac{1}{n} \sum_{i=1\ldots n} (Y_i - \hat{Y}_i)^2. \tag{3.1}$$

Where Y_i is the prediction and $\hat{Y}_i$ is the ground truth.

Loss functions for edge coverage and context-sensitive edge coverage. The outputs of both these tasks are binary values where `1` means an input triggered the *edge_id* or the *call_trace_id* and `0` otherwise. We find that while some *edge_id*s or *call_trace_id*s are invoked very rarely, resulting in imbalanced classes. This usually happens when an input triggers a previously unseen (rare) edges.

Due to this imbalance, training with an off-the-shelf loss functions such as cross entropy is ill suited as it causes a lot of false negative errors often missing these rare edges.

To address this issue, we introduce a parameter called *penalty* (denoted by β) to penalize these false negatives. The penalty is the ratio of the number of times an edge is *not* invoked over the number of times it is invoked. That is,

$$\text{Penalty} = \beta_\tau = \frac{\text{\# times an edge\_id (or call\_trace\_id) is } \textit{not} \text{ invoked}}{\text{\# times an edge\_id (or call\_trace\_id) is invoked}}$$

Here, β_τ represents the penalty for every applicable task $\tau \in \mathcal{T}$ and it is dynamically evaluated as fuzzing progresses. Using β_τ we define an *adaptive loss* for classification tasks in our MTNN as:

$$\mathcal{L}_{\tau_{\text{ec/ctx}}} = -\sum_{\text{edge}} (\beta_\tau \cdot p \cdot log(\hat{p}) + (1-p) \cdot log(1-\hat{p})) \tag{3.2}$$

In Eq. 3.2, $\mathcal{L}_{\tau_{\text{ec/ctx}}}$ results in two separate loss functions for edge coverage and context-sensitive edge coverage. The penalty (β_τ) is used to penalize false-positive and false-negative errors. $\beta_\tau > 1$ penalizes $p \cdot log(\hat{p})$, representing false negatives; $\beta_\tau = 1$ penalizes both false positives and false negatives equally; and $\beta_\tau < 1$ penalizes $(1-p) \cdot log(1-\hat{p})$, representing false positives. With this, we compute the total loss for our multi-task NN model with K tasks:

$$\mathcal{L}_{total} = -\sum_{i=1}^{K} \alpha_i \mathcal{L}_i \tag{3.3}$$

This is the weighted sum of the adaptive loss $\mathcal{L}_i$ for each individual task. Here, α_i presents the weight assigned to task i.

3.3.3 Stage-II: Gradient guided mutation

This phase uses the trained MTNN to generate new inputs that can maximize the code coverage. This is achieved by focusing mutation on the byte locations in the input that can most influence the branching behavior of the program (*hot-bytes*). Task-specific decoders can't be used to infer *hot-bytes* because they tend to offer a localized view of the hot-byte distribution limited to the program regions previously explored by that task.

We use the compact embedding layer of the MTNN (shown in green in Fig. 3.5) to infer the distribution of the hot-bytes. The compact embedding layer is well suited for this because it

(a) captures the most semantically meaningful features (*i.e.*, bytes) in the input in a compact and an interpretable manner; and (b) learns to ignore task specific noise patterns [118] paying more attention to the important bytes that apply to all tasks [119, 120, 121].

Formally, we can represent the input (shown in orange in Fig. 3.5) as a byte vector $xb = \{xb^1, xb^2, ..., xb^n\} \in [0, 255]^n$, where xb^i is the i^{th} byte in xb and n represents the input dimensions (*i.e.*, number of bytes). Then, after the MTNN has been trained, we obtain the compact embedding layer $\mathbf{z} = \{z^1, z^2, ..., z^m\}$ consisting of m nodes. Let $f_i(\mathbf{xb})$ denote the function that embeds the input $\mathbf{xb}$ into the i-th node z^i in the embedding layer. Then, we may express the compact embedding as follows:

$$\begin{bmatrix} z^1 \\ z^2 \\ \vdots \\ z^m \end{bmatrix} = \begin{bmatrix} f_1(\mathbf{xb}) \\ f_2(\mathbf{xb}) \\ \vdots \\ f_m(\mathbf{xb}) \end{bmatrix} = \begin{bmatrix} f_1(xb^1, \cdots, xb^n) \\ f_2(xb^1, \cdots, xb^n) \\ \vdots \\ f_q(xb^1, \cdots, xb^n) \end{bmatrix} \tag{3.4}$$

An important property of Eq. 3.4 is that, for every byte that changes in $\mathbf{xb} = \{xb^1, ..., xb^n\}$, we obtain corresponding changes to every node in $\mathbf{z} = \{z^1, ..., z^m\}$. The extent of the change is determined by how influential each of the n bytes in the input $\mathbf{xb}$ are to all the tasks in the MTNN model. Changes to the *hot-bytes*, which are more influential, will result in *larger* changes to $\mathbf{z}$. This property may be used to discover the *hot-bytes*.

To determine how influential each of the bytes in $\mathbf{xb}$ are, we compute the partial derivatives of the nodes in compact layer with respect to all the input bytes. The partial derivative of the j-th node in the embedding layer $\mathbf{z}$ with respect to the i-th byte in the input is given by:

$$\nabla_{\mathbf{xb}}\, \mathbf{z} = \frac{\partial f_j(\mathbf{x})}{xb^i} = \frac{\partial z^j}{\partial xb^i} \tag{3.5}$$

Since there are n bytes in the input layer $\mathbf{xb}$ and m nodes in the embedding layer, computing the

partial derivatives results in the following *Jacobian Matrix*:

$$\mathbb{J}_{m\times n} = \begin{bmatrix} \frac{\partial z^1}{\partial xb^1} & \cdots & \frac{\partial z^1}{\partial xb^n} \\ \vdots & \ddots & \vdots \\ \frac{\partial z^m}{\partial xb^1} & \cdots & \frac{\partial z^m}{\partial xb^n} \end{bmatrix} \tag{3.6}$$

The above Jacobian matrix has a dimensionality of $m \times n$. Here,

1. Every row in the jacobian matrix corresponds to a node in the embedding layer $z^i \in z$.

2. Every column in the jacobian matrix corresponds to an input byte $xb^i \in z$.

In order to infer the importance of each byte $xb^i \in xb$, we compute the sum of every column in Eq. 3.6.

$$\mathrm{S}(\mathbf{xb}) = \begin{bmatrix} \sum_i \left|\frac{\partial z^i}{xb^1}\right| & \cdots & \sum_i \left|\frac{\partial z^i}{xb^n}\right| \end{bmatrix} \tag{3.7}$$

Here, $\mathrm{S}(\mathbf{xb})$ is a vector of length n. The i-th element in $\mathrm{S}(\mathbf{xb})$ represents the sum of all the gradients of the compact-embedding nodes ($\mathbf{z}$) *w.r.t.* the i-th input byte xb^i. Note that,

- The i^{th} element in $\mathrm{S}(\mathbf{xb})$ measures the *average influence* the i^{th} input byte (xb^i) has on *all* the nodes in the compact layer.

- The numeric value of each of the n elements in $\mathrm{S}(\mathbf{xb})$ determines the hotness of each of the bytes. The larger the value, the more likely it is that the byte is potentially a *hot-byte*.

Using the *saliency map* ($\mathrm{S}(\mathbf{xb})$) from Eq. 3.7 we can now mutate the existing inputs to generate new ones. To do this, we identify the $top-k$ bytes with the largest saliency values as shown below:

$$H(k) = arg\left(top_k\left(\mathrm{S}(\mathbf{xb})\right)\right) \tag{3.8}$$

Here, $H(k)$ represents the byte-locations of the $top-k$ *hot-bytes* that will be mutated by our algorithm. The specifics of the mutation scheme is presented in our pseudocode shown in Fig. 3.7.

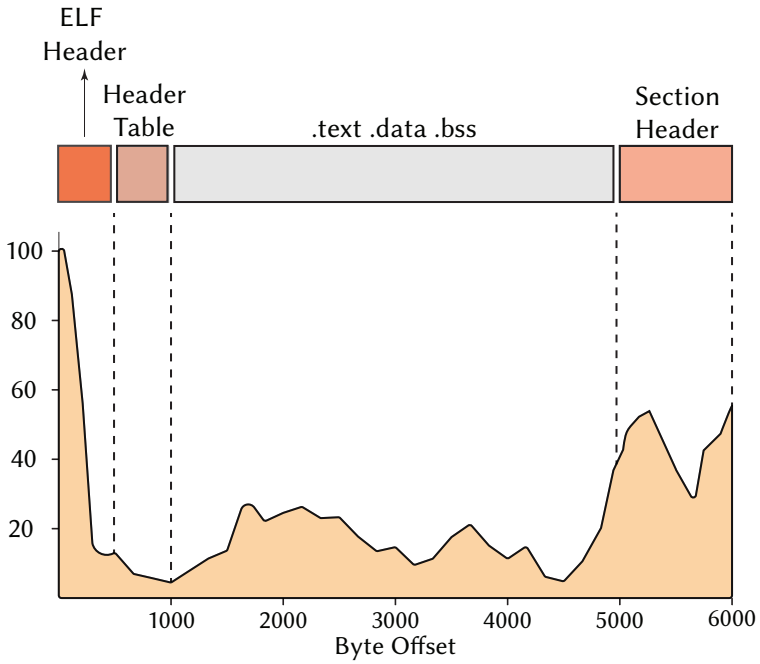


Figure 3.6: **Hot-byte distribution for readelf**

Algorithm 1 Gradient-guided mutation

Input: seed ← initial seed
iter ← number of iterations
k ← Picking the top-k critical bytes for mutation
s ← computed saliency of the input

```
1:  for i = 1 to iter do
2:      locations ← top(g, k_i)
3:      for m = 1 to 255 do
4:          for loc ∈ locations do
5:              v ← seed[loc] + m * sign(g[loc])
6:              v ← clip(v, 0, 255)
7:              gen_mutate(seed, loc, v)
8:          for loc ∈ locations do
9:              v ← seed[loc] − m * sign(g[loc])
10:             v ← clip(v, 0, 255)
11:             gen_mutate(seed, loc, v)
```

Figure 3.7: **Pseudocode of saliency guided mutation.**

To demonstrate that this gradient guided mutation is effective at mutating the *hot-byte* location, we inspect the extent of mutation at each byte location is mutated in ELF binaries (see Fig. 3.6). We

find that the key regions such as *ELF header* and *Section header* are mutated to the greatest extent. Intuitively, this make sense because these regions play a major role in triggering unique program behaviors as compared to other regions.

3.3.4 Stage-III: Seed Selection and Incremental Learning

In this step, MTFUZZ samples some of the mutated inputs from the previous stage to retrain the model. Sampling inputs is crucial as the choice of inputs can significantly affect the fuzzing performance. Also, as fuzzing progresses, the pool of available inputs keeps growing in size. Without some form of sampling strategy, training the NN and computing gradients would take prohibitively long.

To this end, we propose an importance sampling [122] strategy where inputs are sampled such that they reach some *important region* of the control-flow graph instead of randomly sampling from available input. In particular, our sampling strategy first retains all inputs that invoke previously unseen edges. Then, we sort all the seen edges by their rarity. The rarity of an edge is computed by counting how many inputs trigger that specified edge. Finally, we select the top T-rarest edges and include at least one input triggering each of these rare edges. We reason that, by selecting the inputs that invoke the rare edges, we may explore deeper regions of the program on subsequent mutations. In order to limit the number of inputs sampled, we introduce a sampling budget K that determines how many inputs will be selected per iteration.

Using these sampled inputs, we retrained the model periodically to refine its behavior—as more data is becoming available about new execution behavior, retraining makes sure the model has knowledge about them and make more informed predictions.

3.4 Implementation

Our MTNN model is implemented in Keras-2.2.3 with Tensorflow-1.8.0 as a backend [123, 72]. The MTNN is based on a feed-forward model, composed of one shared encoder and three independent decoders. The encoder compresses an input file into a 512 compact feature vector and

feeds it into three following decoders to perform different task predictions. For encoder, we use three hidden layers with dimensions 2048, 1024 and 512. For each decoder, we use one final output layer to perform corresponding task prediction. The dimension of final output layer is determined by different programs. We use ReLU as activation function for all hidden layers. We use sigmoid as the activation function for the output layer. For task specific weights, set each task to equal weight ($\alpha_\tau = 1$ in Eq. 3.3). The MTNN model is trained for 100 epochs achieving a test accuracy of around 95% on average. We use Adam optimizer with a learning rate of 0.001. As for other hyperparameters, we choose k=1024 for $top-K$ hot-bytes. For seed selection budget T in Stage-III (§3.3.4), we use $T = 750$ input samples where each input reaches atleast one rare edge. We note that all these parameters can be tuned in our replication package.

To obtain the various coverage measures, we implement a custom LLVM pass. Specifically, we instrument the first instruction of each basic block of tested programs to track edge transition between them. We also instrument each call instructions to record the calling context of tested programs at runtime. Additionally, we instrument each branch instructions to measure the distance from branching points to their corresponding descendants. As for magic constraints, we intercept operands of each `CMP` instruction and use direct-copy to satisfy these constraints.

3.5 Evaluation

Study Subjects. We evaluate MTFuzz on 10 real-world programs, as shown in Table. 3.1. To demonstrate the performance of MTFuzz, we compare the edge coverage and number of bugs detected by MTFuzz with 5 state-of-the-art fuzzers listed in Table. 3.2. Each state-of-the-art fuzzer was run for 24 hours. The training, retraining, and fuzzing times are included in the total 24 runs for each fuzzer. Training time for MTFuzz is shown in Table. 3.1. MTFuzz and Neuzz both use the same initial seeds for the the approaches and the same fuzzing backend for consistency. We ensure that all other experimental settings were also identical across all studied fuzzers.

Experimental Setup. All our measurements are performed on a system running Ubuntu 18.04 with Intel Xeon E5-2623 CPU and an Nvidia GTX 1080 Ti GPU. For each program tested, we run

Table 3.1: **Test programs used in our study**

Programs		**# Lines**	**MTFuzz train (s)**	**Initial coverage**
Class	Name			
	readelf -a	21,647	703	3,132
binutils-2.30	nm -C	53,457	202	3,031
ELF	objdump -D	72,955	703	3,939
Parser	size	52,991	203	1,868
	strip	56,330	402	3,991
TTF	harfbuzz-1.7.6	9,853	803	5,786
JPEG	libjpeg-9c	8,857	1403	1,609
PDF	mupdf-1.12.0	123,562	403	4,641
XML	libxml2-2.9.7	73,920	903	6,372
ZIP	zlib-1.2.11	1,893	107	1,438

Table 3.2: **State-of-the art fuzzers used in our.**

Fuzzer	**Technical Description**
AFL [79]	evolutionary search
AFLFast [124]	evolutionary + markov-model-based search
FairFuzz [84]	evolutionary + byte masking
Angora [93]	evolutionary + dynamic-taint-guided + coordinate descent + type inference
Neuzz [85]	Neural smoothing guided fuzzing

AFL-2.52b [79] on a single core machine for an hour to collect training data. The average number of training inputs collected for 10 programs is around $2K$. We use `10KB` as the threshold file size for selecting our training data from the AFL input corpus (on average 90% of the files generated by AFL were under the threshold).

3.6 Experimental Results

We evaluate MTFUZZ with the following research questions:

- **RQ1: Performance.** How does MTFUZZ perform in comparison with other state-of-the-art fuzzers?
- **RQ2: Contributions of Auxiliary Tasks.** How much does each auxiliary task contribute to the overall performance of MTFUZZ?
- **RQ3: Impact of Design Choices.** How do various design choices affect the performance of MTFUZZ?
- **RQ4: Transferrability.** How transferable is MTFUZZ?

RQ1: Performance

We compare MTFUZZ wiht other fuzzers (from Table. 3.2) in terms of the number of real-world and synthetic bugs detected (RQ1-A and RQ1-B), and edge coverage (RQ1-C).

RQ1-A. How many real world bugs are discovered by MTFUZZ *compared to other fuzzers?*

Evaluation. To evaluate the number of bugs discovered by a fuzzer, we first instrument the program binaries with `AddressSanitizer` [74] and `UnderfinedBehaviorSanitizer` [125]. Such instrumentation is necessary to detect bugs beyond crashes. Next, we run each of the fuzzers for 24 hours (all fuzzers use the same seed corpus) and gather the test inputs generated by each of the fuzzers. We run each of these test inputs on the instrumented binaries and count the number of bugs found in each setting. Finally, we use the stack trace of bug reports generated by two sanitizers to

categorize the found bugs. Note, if multiple test inputs trigger the same bug, we only consider it once. Table 3.3 reports the results.

Observations. We find that:

1. MTFuzz finds a **total of 71 bugs**, the most among other five fuzzers in 7 real world programs. In the remaining three programs, no bugs were detected by any fuzzer after 24 hours.

2. Among these, **11 bugs were previously unreported**.

Among the other fuzzers, Neuzz (another ML-based fuzzer) is the second-best fuzzer, finding 60 bugs. Angora finds 58. We observe that the 11 new bugs predominantly belonged to 4 types: memory leak, heap overflow, integer overflow, and out-of-memory. Interestingly, MTFuzz discovered a potentially serious heap overflow vulnerability in *mupdf* that was not found by any other fuzzer so far (see 3.8).

A mupdf function `ensure_solid_xref` allocates memory (line 10) for each object of a pdf file and fills content to these memory chunks (line 14). Prior to that, at `line 6`, it tries to obtain the total number of objects by reading a field value `xref->num_objects` which is controlled by program input: a pdf file. MTFuzz leverages gradient to identify the hot bytes which control `xref->num_objects` and sets it to a *negative* value. As a result, `num` maintains its initial value 1 as line 6 if check fails. Thus, at `line 10`, the function allocates memory space for a single object as *num* = 1. However, in `line 14`, it tries to fill more than one object to `new_sub->table` and causes a heap overflow. This bug results in a crash and potential DoS if `mupdf` is used in a web server.

RQ1-B. How many synthetic bugs in LAVA-M dataset are discovered by MTFuzz *compared to other fuzzers?*

Evaluation. LAVA-M is a synthetic bug benchmark where bugs are injected into four GNU `coreutil` programs [41]. Each bug in LAVA-M dataset is guarded by a magic number condition. When the magic verification is passed, the corresponding bug will be triggered. Following conventional practice, we run MTFuzz and other fuzzers from Table. 3.2 on LAVA-M dataset for

Table 3.3: **Real-world bugs found after 24 hours by various fuzzers. MTFUZZ finds the most number of bugs, *i.e.*, 71 (11 unseen) comprised of 4 heap-overflows, 3 Memory leaks, 2 integer overflows, and 2 out-of-memory bugs.**

Program	AFLFast	AFL	FairFuzz	Angora	Neuzz	MTFUZZ
readelf	5	4	5	16	16	17
nm	7	8	8	10	9	12
objdump	6	6	8	5	8	9
size	4	4	5	7	6	10
strip	5	7	9	20	20	21
libjpeg	0	0	0	0	1	1
mupdf	0	0	0	0	0	1
Total	27	29	35	58	60	71

Table 3.4: **Synthetic bugs in LAVA-M dataset found after 5 hours.**

Program	#Bugs	Angora	Neuzz	MTFUZZ
base64	44	48	48	48
md5sum	57	57	60	60
uniq	28	29	29	29
who	2136	1541	1582	1833

a total of 5 hours. We measure the number of bugs triggered by each of the state-of-the-art fuzzers. The result is tabulated in Table. 3.4.

Observations. MTFUZZ discovered the most number of bugs on all 4 programs after 5 hours run. The better performance is attributed to the direct-copy module. To find a bug in LAVA-M dataset, fuzzers need to generate an input which satisfies the magic number condition. MTFUZZ's direct-copy module is very effective to solve these magic number verification since it can intercept operands of each `CMP` instruction at runtime and insert the magic number back into generated inputs.

RQ1-C. How much edge coverage does MTFUZZ *achieve compared to other fuzzers?*

Evaluation. To measure edge coverage, we run each of the fuzzers for 24 hours (all fuzzers use the same seed corpus). We periodically collect the edge coverage information of all the test inputs for each fuzzer using AFL's coverage report toolkit `afl-showmap` [79]. AFL provides coverage instrumentation scheme in two mainstream compilers `GCC` and `Clang`. While some authors prefer to use `afl-gcc` [85, 84, 124], some others use `afl-clang-fast`[93, 126]. The underlying

```
1  // mupdf-1.12.0-source/pdf/pdf-xref.c:174
2  static void ensure_solid_xref(...){
3      ...
4      int num = 1;
5      // xref->num_objects is manipulated by attacker
6      if (num < xref->num_objects)
7          num = xref->num_objects;
8      ...
9      // allocate memory for num objects
10     new_sub->table = fz_calloc(ctx, num, sizeof(object));
11     ...
12     // fill content to num objects
13     for(i = 0; i < sub->len; i++)
14         new_sub->table[i] = sub->table[i];
15     ...
```

Figure 3.8: **Heap overflow bug in `mupdf`. The red line shows the bug.**

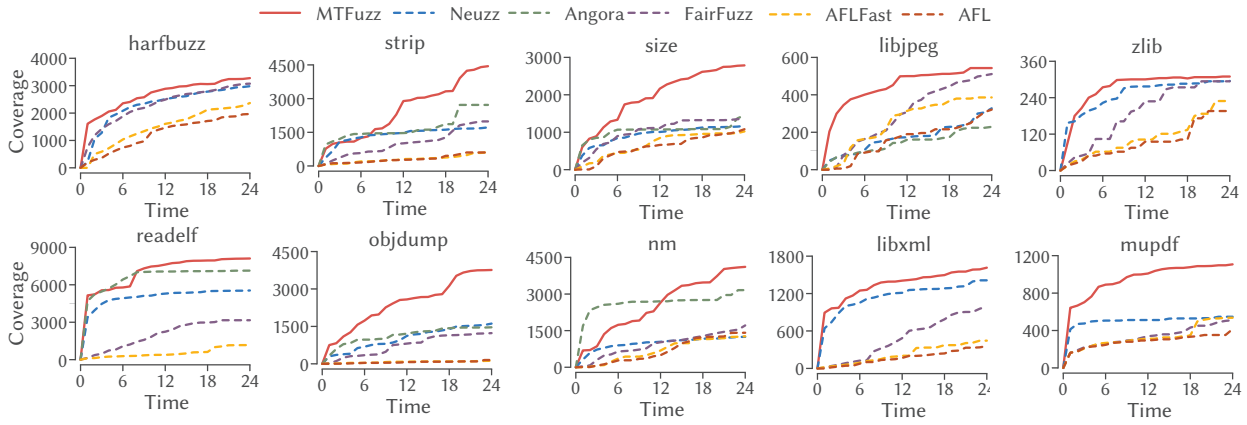


Figure 3.9: **Edge coverage over 24 hours of fuzzing by MTFuzz and other state-of-the-art fuzzers.**

compilers can have different program optimizations which affects how edge coverage is measured. Therefore, in order to offer a fair comparison with previous studies, we measure edge coverage on binaries compiled with with both `afl-gcc` and `afl-clang-fast`. In the rest of the paper, we report results on programs compiled with `afl-clang-fast`. We observed similar findings with `afl-gcc`.

Observations. The results for edge coverage after 24 hours of fuzzing are tabulated in Table. 3.5. The edge-coverage gained over time is shown in Fig. 3.9. Overall, MTFuzz achieves noticeably

Table 3.5: **The *average* edge coverage of MTFUZZ compared with other fuzzers after 24 hours runs for 5 repetitions. Parenthesized numbers represent the standard deviation.**

(a) **Program binaries compiled with `afl-clang-fast`**

Program	MTFUZZ	Neuzz	Angora	FairFuzz	AFL	AFLFast
readelf	8,109	5,953	7,757	3,407	701	1,232
	(286)	*(141)*	*(248)*	*(1005)*	*(87)*	*(437)*
nm	4,112	1,245	3,145	1,694	1,416	1,277
	(161)	*(32)*	*(2033)*	*(518)*	*(144)*	*(18)*
objdump	3,762	1,642	1,465	1,225	163	134
	(359)	*(77)*	*(148)*	*(316)*	*(25)*	*(15)*
size	2,786	1,170	1,586	1,350	1,082	1,023
	(69)	*(45)*	*(204)*	*(47)*	*(86)*	*(117)*
strip	4,406	1,653	2,682	1,920	596	609
	(234)	*(110)*	*(682)*	*(591)*	*(201)*	*(183)*
libjpeg	543	328	201	504	327	393
	(19)	*(34)*	*(42)*	*(101)*	*(99)*	*(21)*
libxml	1,615	1,419	·	956	358	442
	(28)	*(76)*	·	*(313)*	*(71)*	*(41)*
mupdf	1,107	533	·	503	419	536
	(26)	*(76)*	·	*(76)*	*(28)*	*(30)*
zlib	298	297	·	294	196	229
	(38)	*(44)*	·	*(95)*	*(41)*	*(53)*
harfbuzz	3,276	3,000	·	3,060	1,992	2,365
	(140)	*(503)*	·	*(233)*	*(121)*	*(147)*

· indicates cases where Angora failed to run due to the external library issue.

(b) **Program binaries compiled with `alf-gcc`**

Programs	MTFUZZ	Neuzz	Angora	FairFuzz	AFL	AFLFast
readelf	6,701	4,769	6,514	3,423	1,072	1,314
nm	4,457	1,456	2,892	1,603	1,496	1,270
objdump	5,024	2,017	1,783	1,526	247	187
size	3,728	1,737	2,107	1,954	1,426	1,446
strip	6,013	2,726	3,112	3,055	764	757
libjpeg	1,189	719	499	977	671	850
libxml	1,576	1,357	·	1,021	395	388
mupdf	1,107	533	·	503	419	536
zlib	298	297	·	294	196	229
harfbuzz	6,325	5,629	·	5,613	2,616	3,692

more edge coverage than all other baseline fuzzers. Consider the performance gains obtained over the following families of fuzzers:

◦ *Evolutionary fuzzers*: MTFUZZ outperforms all the three evolutionary fuzzers studied here.

MTFuzz outperforms Angora on the 6 programs which Angora supports and achieves up to 2297 more edges in `objdump`. Note, Angora can't run on some programs due to the external library issue on its taint analysis engine [93, 127]. When compared to both FairFuzz and AFLFast, MTFuzz covers significantly more edges, *e.g.*, 4702 more than FairFuzz in `readelf` and over 28.1× edges compared to AFLFast on `objdump`.

∘ *Machine learning based fuzzers*: In comparison with the state-of-the-art ML based fuzzer, Neuzz [85], we observed that MTFuzz achieves much greater edge coverage in all 10 programs studied here. We notice improvements of **2000** more edges in `readelf` and **2500** more edges in `nm` and `strip`.

MTFuzz found 71 real-world bugs (11 were previously unknown) and also reach on average 1,277 and up to 2,867 more edges compared to Neuzz, the second-best fuzzer, on 10 programs.

RQ2: Contributions of Auxiliary Tasks

MTFuzz is comprised of an underlying multi-task neural network (MTNN) that contains one primary task (edge coverage) and two auxiliary tasks namely, context-sensitive edge coverage and approach-sensitive edge coverage. A natural question that arises is—*How much does each auxiliary task contributes to the overall performance?*

Evaluation. To answer this question, We study what would happen to the edge coverage when one of the auxiliary tasks is excluded from the MTFuzz. For this, we build four variants of MTFuzz:

1. *(EC)*: A single-task NN with only the primary task to predict edge coverage.
2. *(EC, Call Ctx)*: An MTNN with edge coverage as the primary task and context-sensitive edge coverage as the auxiliary task.
3. *(EC, Approach)*: An MTNN with edge coverage as the primary task and approach-sensitive edge coverage as the auxiliary task.
4. MTFuzz: Our proposed model with edge coverage as the primary task and two auxiliary tasks context-sensitive edge coverage and approach-sensitive edge coverage.

Table 3.6: **Edge coverage after 1 hour. The most improvement in edge coverage is observed when including both the auxiliary tasks are trained together as a multi-task learner.**

Programs	EC	EC, Call Stack	EC, Approach	MTFuzz
readelf	4,012	4,172	4,044	4,799
nm	546	532	412	577
objdump	605	632	624	672
size	350	404	500	502
strip	744	787	902	954
harfbuzz	593	661	752	884
libjpeg	190	135	182	223
mupdf	252	193	257	269
libxml2	525	649	677	699
zlib	56	33	59	67

To rule out other confounders, we ensure that each setting shares the same hyper-parameters and the same initial seed corpus. Also, we ensure that all subsequent steps in fuzzing remain the same across each experiment. With these settings, we run each of the above multi-task models on all our programs from Table. 3.1 for 1 hour to record the edge coverage for each of these MTNN models.

Observations. Our results are tabulated in Table. 3.6. We make the following noteworthy observations:

1. Fuzzer that uses an MTNN trained on edge coverage as the primary task and context-sensitive edge coverage as the only auxiliary task tends to perform only marginally better than a single task NN based on edge coverage. In some cases, *e.g.*, in Table. 3.6 we notice about 25% more edges. However, in some other cases, for example `libjpeg`, we noticed that the coverage reduces by almost 31%.

2. The above trend is also observable for using edge coverage with approach-sensitive edge coverage as the auxiliary. For example, in `libjpeg`, the edge coverage is lower than the single-task model that uses only edge coverage.

3. However, MTFuzz, which uses both context-sensitive edge coverage and approach-sensitive edge coverage as auxiliary tasks to edge coverage, performs noticeably better than all other models with up to **800 more edges covered** (≈**20%**) in the case of `readelf`.

The aforementioned behavior is expected because each auxiliary task provides very specific

albeit somewhat partial context to edge coverage. Context-sensitive edge coverage only provides context to triggered edges, while approach-sensitive edge coverage only reasons about non-triggered edges (see §3.3.1 for details). Used in isolation, a partial context does not have much to offer. However, while working together as auxiliary tasks along with the primary task, it provides a better context to edge coverage resulting in overall increased edge coverage (see the last column of Table. 3.6).

MTFUZZ benefits from *both* the auxiliary tasks. Using context-sensitive edge coverage and edge coverage along with the primary task (of predicting edge coverage) is most beneficial. We achieve up to **20%** more edge coverage.

RQ3. Impact of Design Choices

While building MTFUZZ, we made few key design choices such as using a task-specific adaptive loss (§3.3.2) to improve the quality of the multi-task neural network (MTNN) model and a novel seed selection strategy based on importance sampling (see §3.3.4). Here we assess how helpful these design choices are.

RQ3-A. What are the benefits of using adaptive loss?

MTNN model predicting for edge coverage and for context-sensitive edge coverage tends to experience severely imbalanced class labels. Consider the instance when a certain input triggers an edge for the first time. This is an input of much interest because it represents a new behaviour. The MTNN model must learn what lead to this behaviour. However, in the training sample, there exists only one positive sample for this new edge in the entire corpus. An MTNN that is trained with an off-the-shelf loss functions is likely to misclassify these edges resulting in a false negative error. Such false negatives are particularly damaging because a large number of new edge discoveries go undetected affecting the overall model performance. To counter this, we defined an adaptive loss in §3.3.2; here we measure how much it improves the MTNN's performance.

Evaluation. To evaluate the effect of class imbalance, we measure *recall* which is high when the overall false negatives (FN) are low. While attempting to minimize FNs the model must not make

too many false positive (FP) errors. Although false positives are not as damaging as false negatives, we must attempt to keep them low. We therefore also keep track of the F1-scores which quantify the trade-off between false positives and false negatives. We train MTFUZZ with two different losses (*i.e.*, with our adaptive loss and with the default cross-entropy loss) on 10 programs for 100 epochs and record the final recall and F-1 scores.

Observations. The result are shown in Table. 3.7. We observe that adaptive loss results in MTNNs with an average of **90**% recall score on 10 programs, while the default loss model only achieves on average 75% recall score. Generally, we notice **improvements greater than 15**% over default loss functions. The low recall for default loss function indicates that it is susceptible to making a lot of false negative predictions. However, our adaptive loss function is much better at reducing false negative predictions. Also, the adaptive loss model achieves on average F-1 score of 72%, while unweighted loss model achieves an average of 70%. This is encouraging because even after significantly reducing the number of false negatives, we maintain the overall performance of the MTNN.

Weighted loss improves MTFUZZ's recall by more than 15%.

RQ3-B. How does seed selection help?

Evaluation. Here, we evaluate our seed selection strategy (§3.3.4) by comparing it to a random selection strategy. Specifically, we run two variants of MTFUZZ, one with importance sampling for seed selection and the other with a random seed selection. All other components of the tool such as MTNN model, hyperparameters, random seed, etc. are kept constant. We measure the edge coverage obtained by both the strategies on 10 programs after fuzzing for one hour. Table. 3.7 shows the results.

Observations. When compared to a random seed selection strategy. Importance sampling outperforms random seed selection in all 10 programs offering average improvements of 1.66× more edges covered than random seed selection—for `readelf`, it covers around 2000 more edges. This makes intuitive sense because, the goal of importance sampling was to retain the newly generated inputs that invoke certain rare edges. By populating the corpus with such rare and novel inputs, the

Table 3.7: **Impact of design choices. Adaptive loss (§3.3.2) increases Recall by ~ 15% while maintaining similar F1-scores. Seed selection based on importance sampling (§3.3.4) demonstrates notable gains in overall edge coverage.**

Programs	Adaptive		Default		Seed Selection	
	Recall(%)	F1(%)	Recall(%)	F1(%)	Our Approach	Random
readelf	88	68	74	66	4,799	2,893
nm	89	62	69	62	577	269
objdump	89	72	65	71	672	437
size	94	81	78	78	502	312
strip	89	73	80	72	954	545
harfbuzz	92	67	80	71	884	558
libjpeg	88	68	65	65	223	124
mupdf	92	84	90	84	269	160
libxml2	90	70	76	69	699	431
zlib	86	70	70	65	67	57

number of newly explored edges would increase over time, resulting in increase edge coverage (see Table. 3.7).

Importance sampling helps MTFUZZ achieve on average 1.66× edge coverage compared with random seed selection.

RQ4. Transferability

In this section, we explore the extent to which MTFUZZ can be generalized across different programs operating on the same inputs (*e.g.*, two ELF fuzzers). Among such programs, we study if we can transfer inputs generated by fuzzing from one program to trigger edge coverage in another program (RQ4-A) and if it is possible to transfer the shared embedding layers between programs (RQ4-B).

RQ4-A. Can inputs generated for one program be transferred to other programs operating on the same domain?

MTFUZZ mutates the hot-bytes in the inputs to generate additional test inputs. These hot-bytes are specific to the underlying structure of the inputs. Therefore, inputs that have been mutated on these hot-bytes should be able to elicit new edge coverage for any program that parses the same input.

Table 3.8: **Generalizability of MTFuzz across different programs parsing the same file types (`ELF` and `XML`). The numbers shown represent new edge coverage.**

File Type	Source → Target	Inputs + Embedding	Inputs only (RQ4-A)		
		MTFuzz (RQ4-B)	MTFuzz	Neuzz	AFL
ELF	nm→ nm*	668	668	315	67
	size → nm	312	294	193	32
	readelf → nm	185	112	68	13
	size → size*	598	598	372	46
	readelf → size	218	151	87	19
	nm→ size	328	236	186	17
	readelf → readelf*	5,153	5,153	3,650	339
	size → readelf	3,146	1,848	1,687	327
	nm → readelf	3,329	2,575	1,597	262
XML	xmlwf → xmlwf*	629	629	343	45
	libxml2 → xmlwf	312	304	187	19
	libxml2 → libxml2*	891	891	643	73
	xmlwf → libxml2	381	298	72	65

* indicates baseline setting without transfer learning

Evaluation. To answer this question, we explore 5 different programs that operate on 2 file types: (1) `readelf`, `size`, and `nm` operating on ELF files, and (2) `libxml` and `xmlwf` [128] operating on XML files. For all the programs that operate on the same file format:

1. We pick a source program (say S = P_i) and use MTFuzz to fuzz the source program for 1 hour to generate new test inputs.

2. Next, for every other target program T = $P_{j \neq i}$, we use the test inputs from the previous step to measure the coverage. Note that we *do not* deploy the fuzzer on the target program, we merely measure the code coverage.

3. For comparison, we use Neuzz (another ML-based fuzzer) and AFL to fuzz the source program S to generate the test inputs for the target program.

Observation. We observe from Table. 3.8 that inputs generated by MTFuzz produce much higher edge coverage on the target program compared to seeds generated by Neuzz or AFL. In general, we notice on average 10× more edge coverage than AFL and 2× more edge coverage than Neuzz. Here, AFL performs the worse, since it generates seeds very specific to the source program.

NEUZZ, a machine learning based fuzzer, performs better than AFL since it attempts to learn some representation of the input, but it falls short of MTFUZZ which learns the most general input representation.

RQ4-B. Can the shared layer be transferred between programs?

We hypothesize that since MTFUZZ can learn a general compact representation of the input, it should, in theory, allow for these compact representations to be *transferred* across programs that share the same input, *e.g.*, across programs that process `ELF` binaries.

Evaluation: To verify this, we do the following:

1. We pick a source program (say S = P_i) and use MTFUZZ to fuzz the source program for 1 hour to generate new tests inputs.

2. For every target program T = $P_{j \neq i}$, we transfer the shared embedding layer along with the test inputs from the source program to fuzz the target program.

3. Note that the key distinction here is, unlike RQ4-A, here we *fuzz* the target program with MTFUZZ using the shared layers and the seed from the source program to bootstrap fuzzing.

Observation. We achieve significantly more edge coverage by transferring both the seeds and the shared embedding layers from the source to target program (Table. 3.8). On average, we obtain 2× more edge coverage on all 10 programs. Specifically, transferring the shared embedding layers and the seeds from `nm` to `readelf` results in covering **2× more** edges compared to Neuzz and over **15× more** edges compared to AFL. Transferring offers better edge coverage compared to fuzzing the target program with AFL.

MTFUZZ's compact embedding can be transferred across programs that operate on similar input formats. We achieve up to 14 times edge coverage for XML files (with an average of 2 times edge coverage across all programs) compared to other state-of-the-art fuzzers.

3.7 Limitation

(a) *Initialization*: For the fuzzers studied here, it is required to provide initial set of seed inputs. To ensure a fair comparison, we use the same set of seed inputs for all the fuzzers.
(b) *Target programs*: We selected diverse target programs from a wide variety of software systems. One still has to be careful when generalizing to other programs not studied here. We ensure that all the target programs used in this study have been used previously; we do not claim that our results generalize beyond these programs.
(c) *Other fuzzers*: When comparing MTFUZZ with other state-of-the-art fuzzers, we use those fuzzers that are reported to work on the programs tested here. Our baseline fuzzer Neuzz [85] has reported to outperform many other fuzzers on the same studied programs. Since we are outperforming Neuzz, it is reasonable to expect that we will outperform the other fuzzers as well.

3.8 Conclusion

This paper presents MTFUZZ, a multi-task neural-network fuzzing framework. MTFUZZ learns from multiple code coverage measures to reduce a sparse and high-dimensional input space to a compact representation. This compact representation is used to guide the fuzzer toward unexplored regions of the source code. Further, this compact representation can be transferred across programs that operate on the same input format. Our findings suggest MTFUZZ can improve edge coverage significantly while discovering several previously unseen bugs.

Chapter 4: K-Scheduler: Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis

Seed scheduling, the order in which seeds are selected, can greatly affect the performance of a fuzzer. Existing approaches schedule seeds based on their historical mutation data, but ignore the structure of the underlying Control Flow Graph (CFG). Examining the CFG can help seed scheduling by revealing the potential edge coverage gain from mutating a seed.

An ideal strategy will schedule seeds based on a count of all reachable and feasible edges from a seed through mutations, but computing feasibility along all edges is prohibitively expensive. Therefore, a seed scheduling strategy must approximate this count. We observe that an approximate count should have 3 properties —(i) it should increase if there are more edges reachable from a seed; (ii) it should decrease if mutation history information suggests an edge is hard to reach or is located far away from currently visited edges; and (iii) it should be efficient to compute over large CFGs.

We observe that centrality measures from graph analysis naturally provide these three properties and therefore can efficiently approximate the likelihood of reaching unvisited edges by mutating a seed. We therefore build a graph called the edge horizon graph that connects seeds to their closest unvisited nodes and compute the seed node's centrality to measure the potential edge coverage gain from mutating a seed.

We implement our approach in `K-Scheduler` and compare with many popular seed scheduling strategies. We find that `K-Scheduler` increases feature coverage by 25.89% compared to Entropic and edge coverage by 4.21% compared to the next-best AFL-based seed scheduler, in arithmetic mean on 12 Google FuzzBench programs. It also finds 3 more previously-unknown bugs than the next-best AFL-based seed scheduler.

4.1 Introduction

Fuzzing is a popular security testing technique that has found numerous vulnerabilities in real-world programs [129, 130, 7, 6, 131, 132, 133, 134, 135, 136, 137, 134]. Fuzzers automatically search through the input space of a program for specific inputs that result in potentially exploitable buggy behaviors. However, the input spaces of most real-world programs are too large to explore exhaustively. Therefore, most existing fuzzers follow an edge-coverage-guided evolutionary approach for guiding the input generation process to ensure that the generated inputs explore different control flow edges of the target program [4, 20, 138]. Starting from a seed input corpus, a coverage-guided fuzzer repeatedly selects a seed from the corpus, mutates it, and adds only those mutated inputs back to the corpus that generate new edge coverage. The performance of such fuzzers have been shown to heavily depend on seed scheduling, the order in which the seeds are selected for mutation [139].

The main challenge in seed scheduling is to identify which seeds in a corpus, when mutated, are more likely to explore many new edges. Performing more mutations on such promising seeds can achieve higher edge coverage. Most prior work on seed scheduling identifies and prioritizes the promising seeds based on the historical distribution of edge/path coverage across prior mutations of the seeds. For example, a fuzzer can prioritize the seeds whose mutations, in the past, resulted in a higher path coverage [140] or triggered rarer edges [84]. However, these existing approaches ignore the structure of the underlying Control Flow Graph (CFG). For example, consider a seed `s1` whose execution path is close to many unvisited edges and a seed `s2` whose execution path is close to only one unvisited edge. Existing coverage-guided fuzzers might schedule seed `S2` before `S1` based on historical patterns. However, examining the structure of the CFG will reveal that `S1` is indeed more promising than `S2` as mutating it can potentially result in exploration of many unvisited edges that are close to the `S1`'s execution path.

The naive strategy of scheduling seeds simply based on the counts of all potentially reachable edges in the CFG for each seed is unlikely to be effective. Such a naive approach assumes that all CFG edges are equally likely to be reachable through mutations which does not hold true for most real-world programs. In fact, some shallow edges tend to be reachable by a large number of mutated

inputs while other deep edges are only reached by a few, if any at all (as many branches might be infeasible) [141]. An ideal strategy would schedule seeds based on the count of all reachable and feasible edges from a seed by mutations. The seeds with higher edge counts will be mutated more. However, computing the feasibility along all edges is impractical as it will incur prohibitive computational cost.

Therefore, a seed scheduling strategy must approximate the feasible edge count. We observe that such an approximation should have 3 properties. First, the approximate count should increase if there are many edges reachable from a seed. Second, the count should decrease if mutation history information suggests that an edge is hard to reach or is located far away from currently visited edges. Empirical evidence from prior work has shown that reaching child nodes through input mutations is typically harder than reaching parent nodes [141] because the number of inputs that can reach a child, for a given path, is strictly less than or equal to the number of inputs that can reach the parent. Third, the approximate count must be efficient to compute for large CFGs as real-world CFGs can be quite large (e.g., inter-procedural CFGs might contain thousands of nodes).

Our key observation is that *centrality* measures from *graph influence analysis* naturally provide the aforementioned properties while measuring a node's influence on the graph. Influence analysis is often used to identify a graph's (e.g., a social network's) most influential nodes and graph centrality measures each node's influence on other nodes with three properties as described below. First, centrality measures additively scale up a node's influence proportional to the number of edges that are reachable from the node. Each sequence of edges of the same length is treated equally independent of its order. Second, centrality measures can easily incorporate external contribution (e.g., based on past mutation history) to a node's influence and can decay contributions from farther away nodes to the node's influence. Contributions decay multiplicatively with the increase in distance (i.e., more intermediate nodes) to reduce contributions from longer paths. Finally, centrality can be efficiently approximated on large graphs using iterative methods [142].

In this paper, we introduce a new approach for seed scheduling based on centrality analysis of the seeds on the CFG. We prioritize scheduling seeds with the largest centrality, i.e., approximate

counts of unvisited but potentially reachable CFG edges from a seed through mutations. To measure a seed's influence with centrality, we modify the CFG to construct an *edge horizon graph* containing the eponymous *horizon* nodes. The horizon nodes form the boundary between the visited and unvisited regions of the CFG for a given fuzzing corpus.

Since horizon nodes delineate between the visited and unvisited regions of the CFG, we first classify CFG nodes as visited or unvisited based on the coverage of a fuzzer's current corpus. We then define horizon nodes as unvisited nodes with a visited parent node. These nodes are crucial to fuzzing because a fuzzer must first visit a horizon node before going further into the unvisited region of the CFG. The centrality of horizon nodes reachable by mutations on a seed therefore measures the seed's ability to discover new edge coverage. Hence, we introduce one node corresponding to each seed and connect the nodes to their corresponding horizon nodes. We do not keep any visited node in the edge horizon graph to avoid inflating a seed's centrality score with contributions from already visited nodes.

To compute centrality over the edge horizon graph, we use Katz centrality because it provides all the three desired approximation properties described earlier in this section and can operate on directed graphs like CFGs. We also use historical mutation data to bias the influence of horizon nodes to a value between 0 and 1 where values closer to 0 mean the node is harder to reach by mutations. The bias value estimates the hardness to reach a node by counting how many mutations reach a node's parents but fail to reach the node itself.

Using the centrality scores for all seeds, a fuzzer can prioritize the seed with the highest centrality. We also periodically re-compute the edge horizon graph and centrality scores during a fuzzing campaign.

We implement our centrality-analysis-based seed scheduling technique as part of `K-Scheduler` (K stands for Katz centrality). Our evaluation shows that `K-Scheduler` increases feature coverage by 25.89% compared to Entropic and edge coverage by 4.21% compared to the next-best AFL-based seed scheduler, in arithmetic mean on 12 Google FuzzBench programs. It also finds 3 more previously-unknown bugs than the next-best AFL-based seed scheduler.

We also conduct preliminary experiments to show the utility of `K-Scheduler` in non-fuzzing seed scheduling settings such as concolic execution and measure the impact of `K-Scheduler`'s design choices.

4.2 Graph Influence Analysis Background

4.2.1 Centrality Measures for Influence Analysis

Identifying a graph's most influential nodes is a common and important task in graph analysis. Many different centrality measures exist in the literature to estimate a node's influence [143]. For example, degree centrality measures a node's influence by counting its direct neighbors. This technique can identify a node with local influence over its neighbors. Eigenvector centrality, in contrast, can identify nodes with global influence over the entire graph. However, eigenvector centrality can fail to produce useful scores on directed graphs [144, 145]. Because program CFGs are directed graphs and we want to measure the global influence of a node to reach other nodes in a graph, we use Katz centrality, a variant of eigenvector centrality for directed graphs. We believe that Pagerank centrality, another eigenvector centrality variant, is not suitable for our setting because it dilutes node influence by the number of its direct neighbors. Such artificial dilutions will undesirably decrease a node's influence in a program's CFG. We conduct experiments to experimentally support this claim in Section 4.6.

For directed graphs like a program CFG, a node's neighbors can be defined by incoming or outgoing edges. Therefore, centrality measures are classified as out-degree if they use outgoing edges or in-degree if they use incoming edges during the computation. Their actual usage depend on the target domain. For example, academic citation graphs use in-degree centrality measures because influential papers are highly cited. In our setting, we use out-degree Katz centrality because we want to measure a node's ability to reach as many unvisited CFG edges (with respect to the current fuzzing corpus) as possible. We describe the details of the out-degree Katz centrality measure below.

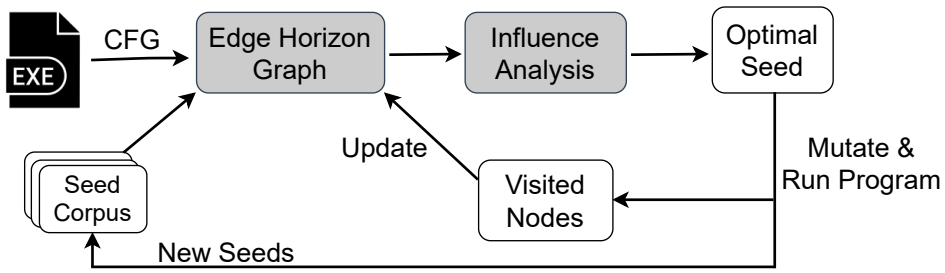


Figure 4.1: **Fuzzer workflow with `K-Scheduler`.**

4.2.2 Katz Centrality

Let A denote an n by n adjacency matrix of a graph with n nodes. If there is an edge connecting node i to node j, element $A_{ij} = 1$. Otherwise, $A_{ij} = 0$. Let $\mathbf{c}$ denote the Katz centrality vector of size n. The element corresponding to node i, c_i, is defined as follows,

$$c_i = \alpha \sum_{j=1}^{n} A_{ij} c_j + \beta_i \tag{4.1}$$

where $\alpha \in [0, 1]$ and β_i is the i-th element of $\boldsymbol{\beta}$, a vector of size n consisting of non-negative elements. Conceptually, the left equation term captures that node centrality additively depends on its neighbors centrality and assigns each neighbor equal weight. Because the sum operator is commutative, the centrality score is independent of the order in which nodes are reached. The right term $\boldsymbol{\beta}$ captures the minimum centrality of a node, which we will later use in Section 4.4 to bias the centrality of horizon nodes based on historical mutation data. The α term represents the decay factor, so that long paths are weighted less than short paths as we show in Section 4.4.

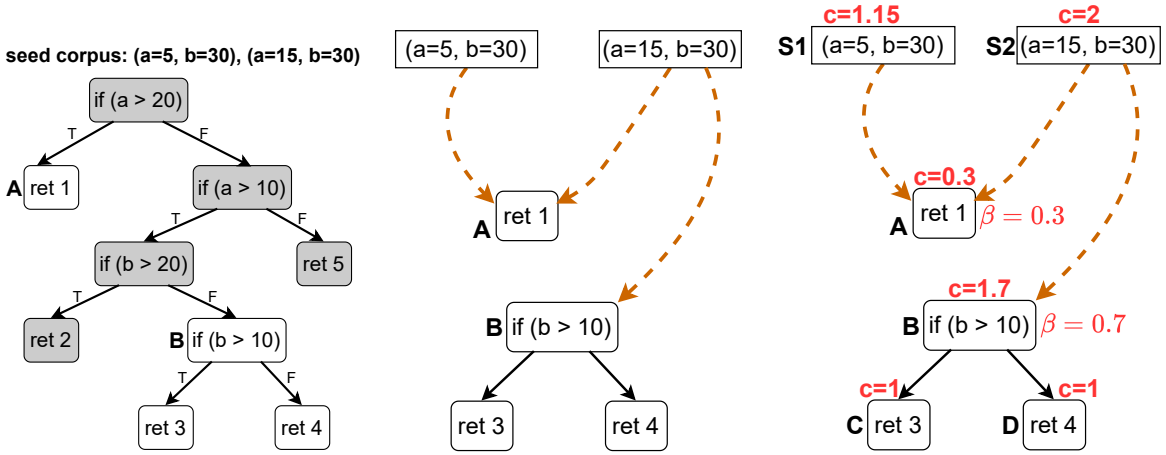
In matrix form, equation 4.1 can be written as

$$\mathbf{c} = \alpha A\mathbf{c} + \boldsymbol{\beta} \tag{4.2}$$

To compute $\mathbf{c}$, the Katz centrality vector, one can solve the linear system so that

$$\mathbf{c} = (I - \alpha A)^{-1}\boldsymbol{\beta} \tag{4.3}$$

```
if(a > 20){
  return 1;
}
else if(a > 10){
  if (b > 20)
    return 2;
  else if (b > 10)
    return 3;
  else
    return 4;
}
else
  return 5;
```



(a) Program CFG. (b) Edge horizon graph (c) Computing Katz Centrality

Figure 4.2: **This figure shows how `K-Scheduler` is used for seed scheduling on a small program. Given the code example on the left, Figure 4.2a shows the corresponding CFG, colored as *gray* if a node is visited and white if unvisited based on the fuzzer corpus. Figure 4.2b shows the edge horizon graph. Figure 4.2c displays node Katz centrality scores computed by iterative power method illustrated in Table 4.1. A fuzzer will prioritize seed** $(\mathtt{a} = 15, \mathtt{b} = 30)$ **because it has the highest centrality score.**

However, computing the matrix inverse in Equation 4.3 is prohibitively expensive with $O(n^3)$ complexity for large graphs. In practice, an iterative approach called the power method is used to approximate $\mathbf{c}$ based on Equation 4.2. After initially setting $\mathbf{c}(0) = \beta$, the power method computes the t-th iteration with the following formula,

$$\mathbf{c}(t) = \alpha A\mathbf{c}(t-1) + \boldsymbol{\beta} \tag{4.4}$$

where $\mathbf{c}(t)$ denotes the t-th iteration. Each iteration increases the power of matrix A which cor-

responds to considering neighbors farther away. Hence, Katz centrality measures global node influence over the entire graph. Each iteration also reduces the contribution of farther away nodes to a node's influence as we describe in Section 4.4. The power method converges to the centrality vector in Equation 4.3 with $O(n)$ complexity under some reasonable assumptions about the graph topology [145] such as α having to be less than the multiplicative inverse of the largest eigenvalue. We refer the reader to [144, 145] for more technical details.

4.3 Overview

Workflow. Figure 4.1 depicts the workflow of `K-Scheduler`. Given a program, seed corpus, and a target program's inter-procedural CFG, we modify the CFG to produce an edge horizon graph composed of only seed, horizon, and non-horizon unvisited nodes. We then use Katz centrality to perform centrality analysis on the edge horizon graph. A fuzzer prioritizes the seed with the highest centrality score. As a fuzzer's mutations reach previously unvisited nodes, we delete these newly visited nodes and re-compute Katz centrality on the updated edge horizon graph.

Motivating Example. Figure 4.2 shows a motivating example to explain our approach. The sample program (shown on the left) returns different values based on user input stored in variables `a` and `b`. Intuitively, we want to pick the seed node that can reach as many unvisited CFG edges as possible. In this case, this corresponds to seed node (`a` = 15, `b` = 30). To do this, our approach `K-Scheduler` takes two steps.

Edge Horizon Graph. First, we modify the CFG to build the edge horizon graph. We classify nodes in the program's CFG as visited or unvisited based on the coverage of a fuzzer's current corpus. Figure 4.2a shows a classification of program's CFG nodes, where nodes in *gray* are visited and nodes in white are unvisited. We next identify horizon nodes, which border the visited and unvisited CFG. In Figure 4.2a, the horizon nodes are nodes `A` and `B` since they are unvisited nodes with a visited parent node. We then insert seed nodes into the CFG and connect them to any horizon node whose parent is visited along the seed's execution path. For example, seed (`a` = 5, `b` = 30) takes both $False$ sides of the branch and hence its horizon node is node `A`. We connect this seed

Table 4.1: **Katz centrality computation by the iterative power method for the edge horizon graph in Figure 4.2c. Each row corresponds to a node's centrality value and each column indicates the current iteration. The power method converges in 3 steps on this simple graph. Assume $\alpha = 0.5$ and $\beta = \mathbf{c}(0)$.**

	t=0	t=1	t=2	t=3
c_a	0.3	0.3	0.3	0.3
c_b	0.7	1.7	1.7	1.7
c_c	1	1	1	1
c_d	1	1	1	1
c_{s1}	1	1.15	1.15	1.15
c_{s2}	1	1.5	2	2

node to horizon node `A`. Finally, we delete all visited nodes in the CFG. Figure 4.2b shows the resulting edge horizon graph.

Katz centrality. Second, we compute Katz centrality over the edge horizon graph. We use the β parameter in the centrality computation to estimate the hardness to reach a node by mutations. For this example, we assume that out of 100 mutations, 70 reached the parent of horizon node `A`, so its $\beta = 1 - \frac{70}{100} = 0.3$ and 30 reached the parent of horizon node B, so its $\beta = 1 - \frac{30}{100} = 0.7$. This shows that horizon node `A` is harder to reach by mutations because a fuzzer failed to reach it with 70% of its mutations. The remaining nodes default to $\beta = 1$ as described in Section 4.4. Katz centrality also decays the contribution from further away nodes when computing a node's centrality with an α parameter. For this example, we assume $\alpha = 0.5$. **Detailed Katz centrality computation.** To see how Katz centrality is computed by the power method from Section 4.2, we show $\mathbf{c}(t = 0), \mathbf{c}(t = 1), \ldots$ until it converges when $t = 3$ in Table 4.1, where the rows indicate the centrality score for a node and the columns indicate time. To explain the intuition behind Katz centrality, we walk through the iteration for a single seed node `s2` to explain the computation. Initially, $c_{s2}(0) = 1$. Using Equation 4.4 from Section 4.2, $c_{s2}(1) = \alpha(c_a(0) + c_b(0)) + \beta_{s2} = 0.5 * (0.3 + 0.7) + 1 = 1.5$. Then, the next iteration is $c_{s2}(2) = \alpha(c_a(1) + c_b(1)) + \beta_{s2} = 0.5 * (0.3 + 1.7) + 1 = 2$ and $c_{s2}(3) = c_{s2}(2)$ due to convergence. This computation illustrates how Katz centrality decays contributions from further away nodes. The number of edges reachable from `s2` is 4 but its Katz centrality score is 2

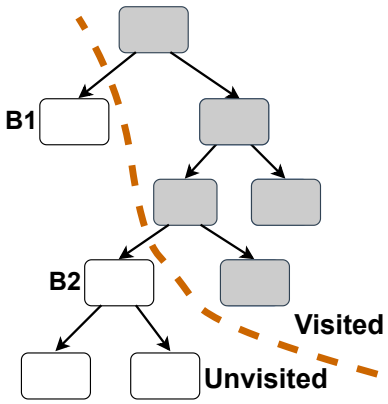


Figure 4.3: **A target program's CFG with visited nodes colored in *gray* and unvisited nodes colored white. The *dashed-brown* line shows the boundary between the visited and unvisited regions of the CFG. Horizon nodes `B1` and `B2` sit at the border and are defined as unvisited nodes with a visited parent node.**

due to this decay. Moreover, the computation reflects that Katz centrality increases if there are more edges reachable from a node. Compared to `s2`, `s1` can only reach 1 edge and hence its centrality of 1.15 is lower. Based on the results of Katz centrality, a fuzzer will prioritize seed (`a` = 15, `b` = 30) because it has the highest centrality score among seed nodes.

4.4 Methodology

In this section, we detail our approach to seed selection with influence analysis. We first describe how we build an edge horizon graph from a program's CFG and then how we compute Katz centrality on the edge horizon graph. Lastly, we describe how our approach can be integrated into a coverage-guided fuzzer.

4.4.1 Edge Horizon Graph Construction

We construct the target program's directed inter-procedural control-flow graph $CFG = (N, E)$, where N is the set of nodes representing the basic blocks and E is the set of edges capturing control-flow transitions through branches, jumps, etc. In the rest of the paper, for clarity, we use CFG to refer to the inter-procedural CFG unless otherwise noted. Directly computing centrality over the original CFG is not useful for seed selection because the graph lacks any reference to seed nodes. Hence, we modify the CFG to construct an edge horizon graph that contains seed nodes. We

Algorithm 2 Edge Horizon Graph Construction.

Input: $G \leftarrow$ Inter-procedural CFG
$S \leftarrow$ Seed corpus
$P \leftarrow$ Program

```
/* Classify Nodes as Visited/Unvisited */
V, U = {}, {}
for s ∈ S do
    visited_nodes = GetCoverage(P, s)
    V = V ∪ visited_nodes                          ▷ Union visited_nodes with V
end for
U = G.nodes \ V                                    ▷ Compute the complement set of V

/* Identify Horizon Nodes */
H = {}
for u ∈ U do
    for p ∈ u.parents do
        if p ∈ V then
            H = H ∪ u                              ▷ Union u with H
        end if
    end for
end for

/* Insert Seed Nodes */
for s ∈ S do
    seed_node = G.AddNode(s)
    visited_nodes = GetCoverage(P, s)
    for v ∈ visited_nodes do
        for n ∈ v.children do
            if n ∈ H then
                G.AddEdge(seed_node, n)
            end if
        end for
    end for
end for

for v ∈ V do
    G.RemoveNode(v)                                ▷ Remove visited nodes
end for
G.RemoveLoops()                                    ▷ Convert G to directed acyclic graph
```

can then compute a seed's centrality for seed selection. At a high level, we classify original CFG nodes as visited or unvisited and connect newly-inserted seed nodes to their corresponding horizon nodes, which are unvisited nodes with a visited parent node. Such connections ensure that a seed's centrality measures its ability to discover new edge coverage. We also delete visited nodes from the CFG to avoid their contributions increasing a seed's centrality score. Lastly, we convert the CFG to a directed acyclic graph to mitigate the undesirable effects of loops on centrality. We present the algorithm for constructing the edge horizon graph in Algorithm 2 and discuss each step in more detail below.

Classifying Nodes as Visited or Unvisited. We first classify all CFG nodes as visited or unvisited based on the coverage of a fuzzer's current corpus. A CFG node is visited if it is reached by the execution path of any seed in the corpus, or elsewise unvisited. We denote the set of visited nodes as V and the set of unvisited nodes as U. More formally,

$$V = \{n | n \in N, visited(n) = 1\} \tag{4.5}$$

$$U = \{n | n \in N, visited(n) = 0\} \tag{4.6}$$

Lines 1 to 6 in Algorithm 2 detail the classification process. Figure 4.3 colors visited nodes in gray and unvisited nodes in white based on the fuzzer's current corpus.

Identifying Horizon Nodes. We define a horizon node in terms of the prior graph partition of V and U, the visited and unvisited nodes as shown below.

$$H = \{u | (v, u) \in E, v \in V, u \in U\} \tag{4.7}$$

In other words, a horizon node is an unvisited node with a visited parent node. Conceptually, horizon nodes border the unvisited and visited region between V and U. Figure 4.3 shows how horizon nodes `B1` and `B2` border the unvisited and visited regions of the CFG. Algorithm 2 computes this set of horizon nodes in lines 8-13. Horizon nodes are *crucial* for fuzzing because a fuzzer must first reach a horizon node to increase edge coverage. This property can be seen in Figure 4.3 where a fuzzer must first reach horizon node `B1` or `B2` to discover new edge coverage. Therefore, a horizon

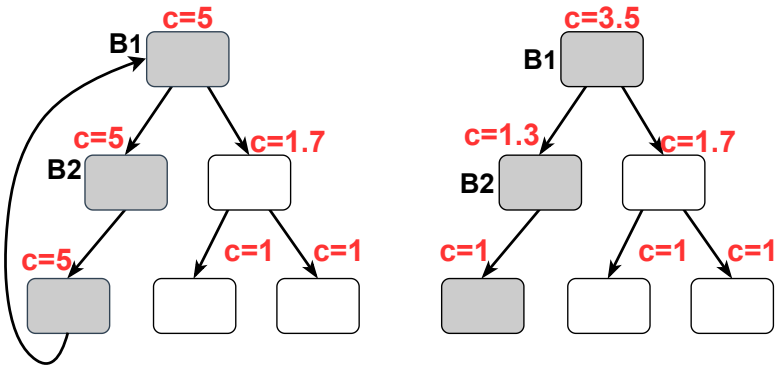
node's centrality measures the number of edges that can potentially be reached by mutations after visiting a horizon node.

Not all horizon nodes, however, have equal centrality. Some horizon nodes can increase edge coverage more than others. As shown in Figure 4.3, horizon node `B2` reaches more edges in U than horizon node `B1`. Hence, a fuzzer should prioritize seeds close to horizon node `B2` because `B2` can reach more edges in the unvisited CFG.

Inserting Seed Nodes. For each seed, we insert one node into the edge horizon graph and connect this seed node to a horizon node if the horizon node's parent is visited along the seed's execution path. Lines 15 to 22 from Algorithm 2 specify how seed nodes are connected to horizon nodes and Figure 4.2b visualizes the connection between seed nodes and horizon nodes. Connecting seed nodes to their corresponding horizon nodes ensures that a seed node's centrality is the sum of its horizon nodes centrality (i.e., Equation 4.1). Therefore, a seed's centrality measures its ability to discover new edge coverage through mutations.

Deleting Visited Nodes. We delete visited nodes from the edge horizon graph because we do not want a seed's centrality score to include contributions from already visited nodes. Note that we preserve the connectivity of the CFG when deleting visited nodes. For example, given a graph `A` $\rightarrow$ `B` $\rightarrow$ `C`, if B was visited, we preserve the connectivity by adding an edge producing `A` $\rightarrow$ `C` . Although this deletion changes the distance between nodes (i.e., $A \rightarrow C$ now has distance 1), it does preserve the connectivity, which is the most critical when measuring centrality.

Mitigating the effect of loops on centrality. Loops in a CFG can hurt the utility of a seed's centrality score for seed selection. Figure 4.4a shows a level loop where node `B1` and its child node `B2` are assigned equal scores by a centrality analysis. However, nodes that initiate a loop should have more centrality than nodes in the loop body. In this case, the node that initiates the loop `B1` should have higher centrality because it can reach more edges. To solve this problem, we convert the CFG to a directed acyclic graph by removing loops in the CFG. Such loops originate in program constructs such as `while` or `for` statements as well as connections between caller and callee nodes (i.e., caller to callee edge and callee to caller backedge can form a loop).



(a) Original CFG (b) Transformed CFG

Figure 4.4: **Figure 4.4a shows that node `B1` has the same centrality as node `B2` as an artifact of the loop. However, `B1` should have higher centrality than `B2` because it can reach more edges. To resolve this, we remove loops from the CFG and Figure 4.4b shows the graph after this transformation.**

4.4.2 Influence Analysis

To compute a seed's centrality, we could count the number of potentially reachable edges from a seed node in the edge horizon graph. However, this count assumes that all edges independent of distance are equally reachable and feasible which does not hold true for most real-world programs [146, 141]. Ideally, we want to count all feasible and reachable edges from a seed through mutations, but this is impractical to compute as it requires computing feasibility along all edges. Instead we use Katz centrality to approximate this count. Katz centrality provides three properties that make it a natural fit to approximate this count. First, it increases its approximation additively if more edges can be reached from a seed node independent of the order as described in Section 4.2. Second, Katz centrality decreases its approximation if mutation frequency information suggests an edge is hard to reach or if edges are far away. Third, Katz centrality is efficient to compute with the power method as discussed in Section 4.2.

Below, we explain how we set the mutation frequency information mechanism in Katz centrality and why Katz centrality multiplicatively decays contributions from further-away edges.

Using historical mutation data as a bias. We observe that β is a generic way of biasing a node's centrality based on external information. We therefore use β to lower a node's centrality if a node

appears harder to reach by mutations. We set each element of β to range from 0 to 1, where values closer to 0 mean the node is harder to reach through mutations. To measure this hardness, we use historical mutation data. We initialize $\beta = 1$ if there are no mutations and iteratively refine it as a fuzzer generates mutations. We use the following equation for node i,

$$\beta_i = 1 - \frac{R_i}{T} \tag{4.8}$$

where R_i measures the number of mutations that reach node i's parents and T measures the total number of mutations for all seeds.

Lastly, to set α, which ranges from 0 to 1, from Equation 4.9, we observe that setting $\alpha = 0$ means all nodes in the edge horizon graph will have the same centrality. This would not be useful for seed selection because we could not distinguish which seed node was more likely to discover new edge coverage with its centrality score. In contrast, setting $\alpha = 1$ treats closer and further-away edges with equal contribution, which fails to reflect program behavior. In practice, we set $\alpha = 0.5$ based on our experiments as described in Section 4.6.

Decaying contributions from longer paths. Katz centrality multiplicatively decays the contribution from further away edges when computing a node's centrality . This decay corresponds to a well-known program behavior where further away edges are harder to reach by mutations [141]. To see how Katz centrality reduces the contribution from further-away edges toward a node's centrality, consider Equation 4.9 which shows the 2nd iteration of the power method from Section 4.2.

$$\mathbf{c}(2) = ((\alpha)^0 I + (\alpha)^1 A + (\alpha)^2 A^2)\beta \tag{4.9}$$

Notice how the parameter α, which ranges between 0 and 1, multiplicatively decays the contribution from higher matrix powers. As discussed in Section 4.2, higher matrix powers consider edges farther away. Thus, this equation shows Katz centrality reduces the contribution from further away edges with multiplicative decay.

4.4.3 Seed Scheduling

Algorithm 3 shows how to integrate `K-Scheduler` into a coverage-guided fuzzer. `K-Scheduler` first builds the edge horizon graph as shown in Algorithm 2 and computes the Katz centrality over it to measure each seed's centrality. A fuzzer then uses these scores for seed scheduling which consists of selecting a seed and allocating a corresponding mutation budget. Because popular fuzzers such as AFL and LibFuzzer differ greatly in these two components, we abstract them out in lines 10 and 11 and specify how to integrate our generic technique into them in Section 4.5. Finally, `K-Scheduler` re-computes the edge horizon graph and its Katz centrality when the fuzzer discovers new edge coverage or a fixed time has elapsed. Periodically updating centrality (i.e. via β) ensures that `K-Scheduler` provides useful guidance even when a fuzzer fails to find new edge coverage.

Algorithm 3 Fuzzer integration with `K-Scheduler`.

Input: $G \leftarrow$ Inter-procedural CFG
$S \leftarrow$ Seed corpus
$P \leftarrow$ Program

```
1:  stats = {}                                          ▷ Store mutation statistics
2:  has_new = False                                     ▷ Indicate new edge coverage
3:  t = CreateTimer(k)                                  ▷ Build horizon graph every k seconds
4:  while fuzzer is running do
5:      if has_new = True or stats = ∅ or t.timeout() then
6:          H = GetHorizonNodes(G, S, P)
7:          Beta = ComputeBeta(H, stats)
8:          G_horizon = GetHorizonGraph(G, S, P)
9:          C_katz = KatzCentrality(G_horizon, Beta)
10:         t.reset()                                   ▷ Reset timer t
11:     end if
12:     seed = ChooseSeed(S, C_katz)
13:     energy = ComputeEnergy(seed, C_katz)
14:     has_new = Mutate(seed, energy)                  ▷ Fuzz seed with energy
15:     stats.update()
16: end while
```

4.5 Implementation

`K-Scheduler` consists of two components. First, to build the edge horizon graph, we construct the target program's inter-procedural CFG. We initially compile the program with `wllvm` [147] and use the LLVM's (version 11.0.1) `opt` tool to extract each function's intra-procedural CFG. In Python 3.7, we then merge each intra-procedural CFG together based on caller-callee relations to produce the inter-procedural CFG. We also implement all pieces from Algorithm 2 such as loop removal in Python. To classify CFG nodes as visited, we re-use a fuzzer's edge coverage information to identify visited basic blocks. Second, to compute Katz centrality, we use the power method provided by networkit [148], a large-scale graph computing library.

We now describe how we integrate `K-Scheduler` into LibFuzzer [20] and AFL [4] to show our technique is generic and widely applicable. We run `K-Scheduler` as a standalone process that communicates with a fuzzer to set the fuzzer's seed ranking based on centrality and identify the mapping between a seed node and its corresponding horizon nodes. We measure how much overhead `K-Scheduler` adds to the fuzzing process in Section 4.6.

Libfuzzer Integration. Libfuzzer [20] computes an energy for each seed in the form of a probability and flips a coin with bias corresponding to the seed's energy to determine whether a seed should be selected for mutation. Higher energy probabilities indicate a seed will be chosen more frequently. To integrate into Libfuzzer, we follow the same integration as Entropic, a state-of-the-art seed scheduler for Libfuzzer, and set each seed's energy to its Katz centrality score normalized by the total centrality scores for all seeds.

AFL Integration. Unlike Libfuzzer's probabilistic seed selection, AFL generally selects every seed for mutation. A seed's energy also determines its corresponding mutation budget. To integrate into AFL, we set each seed's energy directly to its Katz centrality score.

4.6 Evaluation

Our evaluation aims to answer the following questions.

1. **Comparison against seed schedulers:** How does `K-Scheduler` compare against other seed scheduling strategies?

2. **Bug Finding:** Does `K-Scheduler` improve a fuzzer's ability to find bugs?

3. **Runtime Overhead:** What is the performance overhead of `K-Scheduler`?

4. **Impact of Design Choices:** How do `K-Scheduler`'s various design choices contribute to its performance?

5. **Non-evolutionary fuzzing settings:** Does `K-Scheduler` show promise for seed scheduling in non-evolutionary fuzzing settings?

4.6.1 Experimental Setup

Baseline Seed Scheduling Strategies

We compare against popular seed scheduling strategies from industry and the academic community. These strategies are generally integrated into AFL or Libfuzzer. Directly comparing a seed scheduling strategy that uses AFL with another seed scheduling strategy that uses Libfuzzer can be misleading since the underlying fuzzers may cause the performance difference instead of the underlying seed scheduling strategy. Therefore, to be fair, we integrate `K-Scheduler` into both Libfuzzer and AFL separately and make comparisons about seed scheduling strategies when the underlying fuzzer is the same. Note this integration also demonstrates that `K-Scheduler` is generic and widely applicable.

For `K-Scheduler`'s comparison against Libfuzzer-based seed schedulers, we compare `K-Scheduler` against Entropic, a state-of-the-art seed scheduler in Libfuzzer [149]. To ensure a fair comparsion, we follow the same integration with Libfuzzer as Entropic. We also compare against Libfuzzer's default seed scheduler as a baseline and refer to it as Default. We use Libfuzzer and Entropic from LLVM 11.0.1 in our comparison. For `K-Scheduler`'s comparison against AFL-based seed schedulers, we compare against strategies that prioritize seeds if they take paths rarely observed (RarePath), reach rarely observed edges (RareEdge) or discover new paths (NewPath). We also

compare against a strategy that prioritizes seeds based on security-sensitive coverage (SecCov). To compare against RarePath, RareEdge, NewPath, and SecCov we use AFLFast [5], FairFuzz [84], EcoFuzz [140], and TortoiseFuzz [134] respectively. Since these fuzzers all modify AFL, we integrate `K-Scheduler` into AFL using their same modifications for a fair comparison. Moreover, we set each fuzzer to use the same mutation strategy to a enable a fair comparison. Hence, we disabled FairFuzz's custom mutation strategy. We also compare against AFL's default seed scheduling strategy as a baseline and refer to it as Default.

Benchmark Programs

In our seed scheduler comparison, we use the Google FuzzBench benchmark, a commonly used dataset to evaluate fuzzing performance on real-world programs. At the time of this writing, the benchmark consists of 40+ programs, so we decide to evaluate over a subset of them. We pick 12 diverse real-world programs from the benchmark that includes cryptographic and database programs as well as parsers as shown in Table 4.3. We plan to evaluate against the entire benchmark in the future. We also use the default seed corpus and configuration provided by the benchmark to enable a fair comparison. Note that Google FuzzBench configures all AFL-based fuzzers to use havoc mode by default [150], since AFL havoc mode has been shown to significantly outperform AFL deterministic mode [151].

For our bug-finding experiments, we select 12 real-world parsing programs commonly used to evaluate fuzzer's bug finding performance [5, 84, 140]. The 12 programs cover 8 file formats: `ELF`, `ZIP`, `PNG`, `JPEG`, `TIFF`, `TAR`, `TEXT` and `XML`. The list of programs and their details can be found in Table 4.6. Since these programs do not come with a default seed corpus, we make a corpus with small valid files.

Environmental Setup

We run all our evaluations on 4 64-bit machines running Ubuntu 20.04 with Intel Xeon E5-2623 CPUs (96 cores in total). We follow standard operating procedure in fuzzing evaluations [5, 149, 84] and bound each fuzzer to 1 CPU core. Because our current implementation runs `K-Scheduler`

in a separate process, we assign fuzzers using `K-Scheduler` 2 cores, one for the fuzzer and one for the `K-Scheduler`.

4.6.2 RQ1: Seed scheduling comparison

For `K-Scheduler`'s comparison against Libfuzzer-based seed schedulers, we follow the original evaluation of Entropic [149] and use the same two metrics for comparison: edge coverage and feature coverage. Edge coverage measures how many branches were reached along an input's execution path, whereas feature coverage includes this information as well as branch hit count. For example, edge coverage would not distinguish coverage between two inputs that visit the same branch a different number of times, but feature coverage would distinguish them.

We run `K-Scheduler`, Default (i.e., Libfuzzer's default seed scheduler), and Entropic on the 12 Google FuzzBench programs for 24 hours. We repeat each 24 hour run ten times for statistical power. In arithmetic mean over these 10 runs, Table 4.2 and Table 4.3 summarize the edge and feature coverage results for 1 hour and 24 hours, respectively. Within 1 hour, `K-Scheduler` improves upon next-best seed scheduling strategy Entropic by 20.11% in median and 31.75% in arithmetic mean over the 12 FuzzBench programs in feature coverage. For the 24 hour runs, `K-Scheduler` achieves 20.66% in median and 25.89% in arithmetic mean more feature coverage than Entropic. We attribute the increased improvement of `K-Scheduler` over Entropic within the first hour to `K-Scheduler`'s scheduling of promising seeds more frequently given a limited fuzzing budget (i.e., fuzzer only schedules a limited number of seeds). However, as the fuzzing budget increases to 24 hours, Entropic will eventually also schedule those promising seeds more frequently, which narrows the performance difference between `K-Scheduler`. Moreover, with a significance level of 0.05, our feature coverage over Entropic results are statistically significant for all programs for 24 hour runs and all programs except `zlib` for the 1 hour runs. Our results show that using the CFG structure for seed scheduling can improve fuzzing performance.

For `K-Scheduler`'s comparison against AFL-based seed schedulers, we only use edge coverage as a metric for comparison because AFL does not report feature coverage. We run `K-Scheduler`, Default (i.e., AFL's default seed scheduler), RarePath, RareEdge, NewPath, and

Table 4.2: **Arithmetic mean feature and edge coverage of Libfuzzer-based seed schedulers on 12 FuzzBench programs for 1 hour over 10 runs. We mark the highest number in bold.**

Programs	K-Scheduler		Entropic		Default	
	feature	edge	feature	edge	feature	edge
freetype	**51,184**	**10,886**	46,698	10,691	40,040	9,446
libxml2	**39,240**	**7,661**	24,167	6,128	25,914	6,296
lcms	**2,886**	**1,497**	1,707	1,004	1,392	874
harfbuzz	**35,017**	**9,112**	23,349	7,551	23,455	7,588
libjpeg	**10,974**	**2,553**	7,424	2,193	7,510	2,208
libpng	**5,001**	**1,501**	4,604	1,469	4,525	1,476
openssl	**14,520**	**4,622**	12,830	4,294	13,029	4,327
openthread	**6,525**	**3,318**	5,397	3,044	5,150	2,947
re2	**31,292**	**6,275**	28,877	6,147	29,941	6,207
sqlite	**73,532**	**13,299**	44,198	12,189	52,060	12,735
vorbis	**9,106**	**2,136**	7,632	2,010	5,710	1,823
zlib	**2,711**	**790**	2,572	784	2,408	782
Arithmetic mean coverage gain			31.75%	12.51%	37.37%	15.72%
Median coverage gain			20.11%	8.32%	34.54%	13.91%

Table 4.3: **Arithmetic mean feature and edge coverage of Libfuzzer-based seed schedulers on 12 FuzzBench programs for 24 hours over 10 runs. We mark the highest number in bold.**

Programs	K-Scheduler		Entropic		Default	
	feature	edge	feature	edge	feature	edge
freetype	71,717	13,754	**75,370**	**14,120**	67,510	12,870
libxml2	**54,081**	**9,869**	36,958	7,038	39,247	7,310
lcms	**6,345**	**2,541**	4,425	2,082	3,413	1,784
harfbuzz	**48,105**	**10,358**	32,799	8,808	33,499	8,912
libjpeg	**15,861**	**3,033**	11,755	2,646	11,220	2,574
libpng	**5,312**	**1,535**	5,002	1,501	4,992	1,501
openssl	**16,644**	**4,971**	15,137	4,731	15,173	4,738
openthread	**11,405**	**4,965**	6,435	3,276	6,123	3,196
re2	**33,797**	**6,482**	32,401	6,347	32,725	6,367
sqlite	**92,493**	**15,540**	75,723	14,351	83,228	14,710
vorbis	**10,417**	**2,247**	9,906	2,208	8,873	2,115
zlib	**3,215**	**801**	2,698	790	2,510	787
Arithmetic mean coverage gain			25.89%	13.69%	31.43%	16.34%
Median coverage gain			20.66%	6.68%	22.75%	6.54%

SecurityCov on the same 12 Google FuzzBench programs for 24 hours, repeated ten times. In arithmetic mean over these 10 runs, Table 4.4 and Table 4.5 summarize the edge coverage results for 1

Table 4.4: **Arithmetic mean edge coverage of AFL-based seed schedulers on 12 FuzzBench programs for 1 hour over 10 runs.**

	K-Sched	**Default**	**RarePath**	**RareEdge**	**NewPath**	**SecCov**
Fuzzer	AFL	AFL	AflFast	FairFuzz	EcoFuzz	TortoiseFuzz
freetype	**12,077**	11,001	10,707	11,319	8,925	10,532
libxml2	**8,120**	5,793	5,836	7,247	5,841	5,476
lcms	1,882	**1,989**	1,540	1,343	1,117	1,327
harfbuzz	**9,169**	8,864	9,022	8,767	7,629	8,773
libjpeg	**2,391**	2,354	2,374	2,140	1,739	2,073
libpng	1,470	**1,488**	1,460	1,430	1,428	1,456
openssl	**4,560**	4,485	4,399	4,381	4,252	4,336
openthread	**5,245**	5,063	5,064	5,047	5,047	5,012
re2	**5,792**	5,612	5,533	5,335	5,484	5,252
sqlite	9,865	10,038	9,890	**10,065**	9,722	9,627
vorbis	**2,048**	2,006	1,946	1,933	1,761	1,914
zlib	**761**	758	752	746	745	752
Arithmetic mean gain		4.80%	7.95%	8.39%	20.01%	13.03%
Median gain		1.87%	3.62%	5.27%	11.77%	6.07%

hour and 24 hours respectively. Similar to the comparison against Libfuzzer-based seed schedulers, we observe a higher improvement of `K-Scheduler` over the other seed scheduling strategies within the first hour. `K-Scheduler` outperforms the next best seed scheduling strategy (RarePath) by 7.95% in arithmetic mean and 3.62% in median over the 12 FuzzBench programs. For the 24 hour runs, `K-Scheduler` achieves 4.21% in arithmetic mean and 1.91% in median more coverage than RarePath. We note that the improvement of `K-Scheduler` against AFL-based seed schedulers is not as significant as `K-Scheduler`'s comparison against Libfuzzer-based seed schedulers. We believe `K-Scheduler`'s diminished performance difference occurs because the underlying fuzzer, AFL, iterates over the seed queue multiple times during the 24 hours fuzzing campaign and therefore will schedule nearly all seeds frequently, reducing the effect of seed selection.

The coverage plots over time also highlight the promise of `K-Scheduler`. Figure 4.5 and 4.6 show that `K-Scheduler` generally maintains its performance advantage during the lifetime of the fuzzing campaign. The consistency of `K-Scheduler`'s gain across many different seed schedulers show the promise of scheduling seeds based on CFG information. Moreover, it suggests `K-Scheduler` can be helpful independent of a fuzzer as we later explore.

Table 4.5: **Arithmetic mean edge coverage of AFL-based seed schedulers on 12 FuzzBench programs for 24 hours over 10 runs.**

	K-Sched	**Default**	**RarePath**	**RareEdge**	**NewPath**	**SecCov**
Fuzzer	AFL	AFL	AflFast	FairFuzz	EcoFuzz	TortoiseFuzz
freetype	**14,188**	13,508	13,646	13,486	11,965	13,206
libxml2	**10,936**	9,295	8,546	10,241	8,964	9,147
lcms	**2,325**	2,247	2,160	2,190	1,892	2,162
harfbuzz	**10,061**	9,980	10,019	9,804	9,946	9,882
libjpeg	**2,678**	2,513	2,601	2,497	2,309	2,413
libpng	**1,536**	**1,536**	1,535	1,524	1,528	1,528
openssl	**4,863**	4,805	4,761	4,788	4,732	4,685
openthread	**5,766**	5,704	5,646	5,666	5,527	5,636
re2	**5,887**	5,875	5,790	5,536	5,774	5,758
sqlite	12,081	**12,360**	12,019	10,648	12,199	11,810
vorbis	**2,215**	2,195	2,202	2,100	2,171	2,184
zlib	**780**	**780**	775	778	777	769
Arithmetic mean gain	2.89%	4.21%	4.81%	7.63%	5.11%	
Median gain	1.00%	1.91%	5.34%	2.38%	2.30%	

Result 1: `K-Scheduler` increases feature coverage by 25.89% compared to Entropic and edge coverage by 4.21% compared to the next-best AFL-based seed scheduler (RarePath), in arithmetic mean on 12 Google FuzzBench programs.

4.6.3 RQ2: Bug Finding

In order to detect memory corruption bugs that do not necessarily lead to a crash, we compile program binaries with Address and Undefined Behavior Sanitizers. We then ran `K-Scheduler`, Default (i.e., AFL's default seed scheduler), RarePath, RareEdge, and NewPath on 12 real-world parsing programs for 24 hours, a total of 10 times. We could not run the Libfuzzer-based seed schedulers because the 12 parsing programs are not equipped with a Libfuzzer-compatible fuzzing harness (i.e., LLVMFuzzerTestOneInput is undefined).

In our 24 hour runs, we found real-world bugs in binutils. Table 4.7 shows the bug count for each seed scheduling strategy in terms of integer overflow, out of memory and memory leak bugs, in arithmetic mean over the 10 runs. We count bugs with the following procedure based on prior work [130, 7, 10]. We first use AFL-CMin to reduce the number of crashing inputs. We then

Table 4.6: **Tested Programs in Bug Finding Experiments.**

Subjects	Version	Format	# lines
xmllint	libxml2-2.9.7	XML	72,630
miniunz	zlib-1.2.11	ZIP	1,895
readpng	libpng-1.6.37	PNG	3,205
djpeg	libjpeg-9d	JPEG	9,204
size	binutils-2.36.1	ELF	51,203
readelf -a	binutils-2.36.1	ELF	29,954
nm -C	binutils-2.36.1	ELF	52,763
objdump -D	binutils-2.36.1	ELF	78,610
strip	binutils-2.36.1	ELF	59,680
tiff2pdf	tiff-4.3.0	TIFF	20,387
bsdtar -xf	libarchive-3.5.1	TAR	45,031
infotocap	ncurses-6.2	TEXT	23,145

Table 4.7: **Overview of bugs discovered in our AFL-based seed scheduling experiments categorized by type.**

	K-Sched	**Default**	**RarePath**	**RareEdge**	**NewPath**	**SecCov**
Fuzzer	AFL	AFL	AflFast	FairFuzz	EcoFuzz	Tortoise†
out-of-memory	21	14	19	17	18	21
memory leak	24	20	21	19	20	22
integer overflow	3	2	3	3	2	2
Total	48	36	43	39	40	45

† Tortoise denotes TortoiseFuzz.

further deduplicate the crashing inputs by filtering them by unique stack traces. We lastly triage the remaining crashing inputs by manually reviewing their stack traces and corresponding source code. Our results show that `K-Scheduler` finds 3 more bugs than the next best seed scheduling strategy SecCov (i.e., TortoiseFuzz), which optimizes for bug-finding.

Result 2: `K-Scheduler` discovers 3 more bugs than the next best seed-scheduling strategy.

4.6.4 RQ3: Runtime Overhead

In this experiment, we measure the overhead that `K-Scheduler` adds to a fuzzer. The runtime overhead can be classified into two components: a fuzzer maintenance (i.e., record hit count of edges and compute seeds' energy) and a fuzzer invoking `K-Scheduler` (i.e., construct edge horizon graph and perform Katz centrality analysis) for seed scheduling. To measure these overheads,

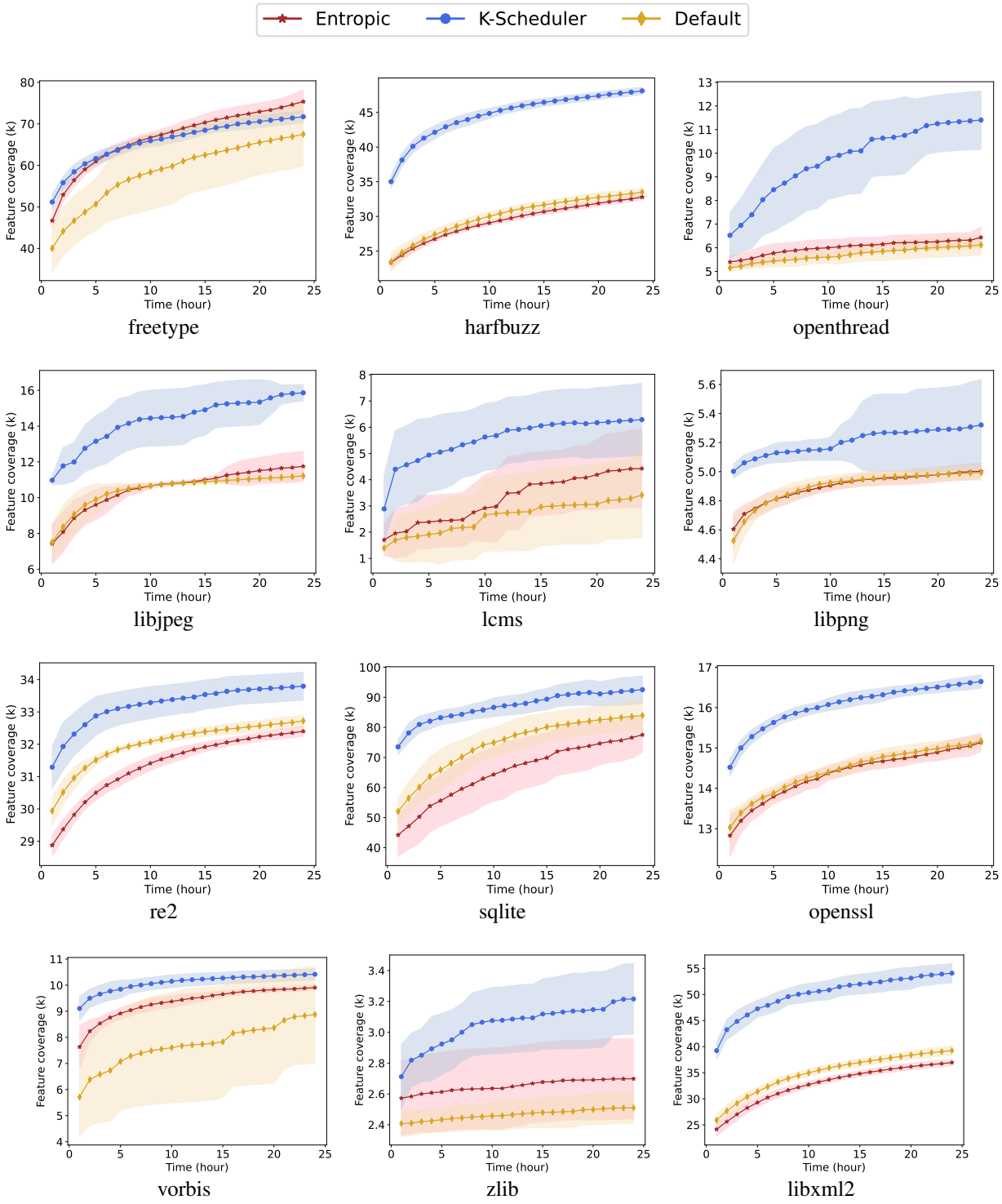


Figure 4.5: **The arithmetic mean feature coverage of Libfuzzer-based seed schedulers running for 24 hours and one standard deviation error bars over 10 runs. Default refers to the default seed scheduler in Libfuzzer.**

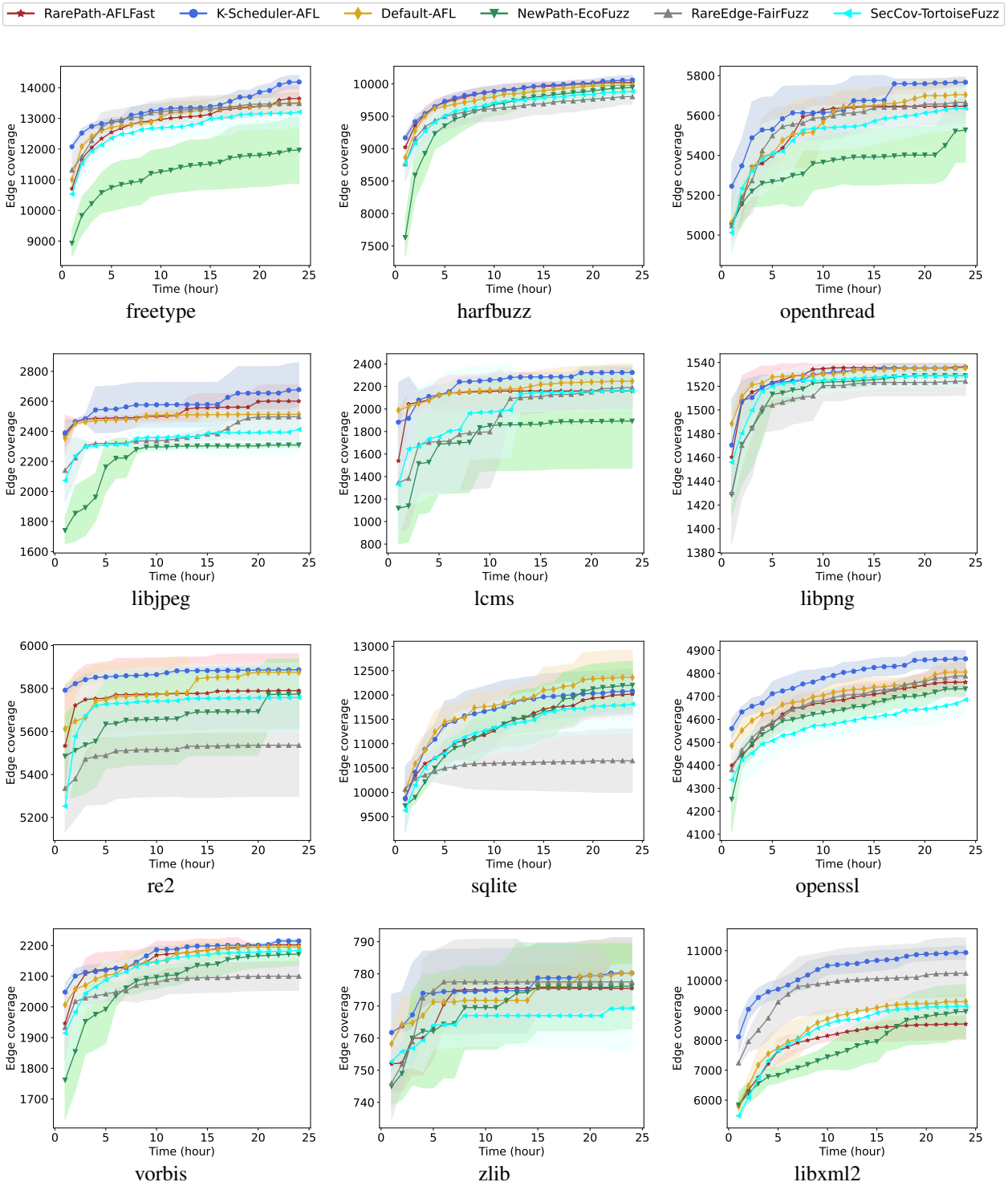


Figure 4.6: **The arithmetic mean edge coverage of of AFL-based seed schedulers running for 24 hours and one standard deviation error bars over 10 runs. Default refers to the default seed scheduler in AFL.**

we run our modified versions (see Section 4.5) of AFL and Libfuzzer against all 12 FuzzBench programs for 24 hours, recording the total time they spend in maintenance and separately the total time spent in computing Katz centrality over the edge horizon graph in the standalone process. We repeat this experiment 10 times to minimize variance. Table 4.8 summarizes the runtime overhead added to AFL's and LibFuzzer fuzzing processes in terms of fuzzer maintenance and graph centrality analysis.

The overhead of fuzzer maintenance is 0.28% for AFL and 1.74% for Libfuzzer, in arithmetic mean over the 12 FuzzBench programs. The graph analysis overhead is minimal, adding 0.15% in arithmetic mean over the 12 FuzzBench programs. We believe these small graph analysis overheads exist because Katz centrality can be efficiently computed with the power method (Section 4.2) and the edge horizon graph is cached and updated instead of being constructed from scratch each time. For clarity, we did not report graph analysis overheads for AFL and Libfuzzer separately because they use the same standalone process, so the overheads were nearly indistinguishable. Moreover, the difference in overheads per-program is explained by the variance in the target program's CFG size (i.e., number of nodes).

Result 3: `K-Scheduler` adds at most 1% overhead from graph analysis and at most 2% overhead for fuzzer maintenance.

4.6.5 RQ4: Impact of Design Choices

We conduct experiments to measure the performance effect of five design choices: (i) centrality measure, (ii) β parameterization, (iii) visited node deletion, (iv) loop removal, and (v) α parameterization. For each design choice experiment, we run `K-Scheduler` with Libfuzzer on the 12 Google FuzzBench programs for 1 hour, repeated 10 times, and compare their feature coverage. We run for 1 hour because the first hour of a fuzzing run often discovers more coverage than later hours and hence our results better measure the effect of the design choices. We also choose feature coverage because it provides more fine-grained information about a fuzzer's behavior than edge coverage. We describe each design choice experiment in more detail below.

Table 4.8: **Runtime overhead from `K-Scheduler` in Libfuzzer and AFL-based seed scheduling.**

Programs	Nodes #	Graph Analysis	Fuzzer Maintenance	
			LibFuzzer	AFL
freetype	38,352	0.20%	1.71%	0.23%
libxml2	96,732	0.22%	2.53%	0.39%
lcms	13,081	0.06%	0.92%	0.08%
harfbuzz	21,066	0.11%	2.25%	0.17%
libjpeg	16,508	0.04%	0.79%	0.06%
libpng	7,215	0.02%	0.53%	0.03%
openssl	57,729	0.25%	2.43%	0.67%
openthread	27,263	0.09%	1.48%	0.24%
re2	12,020	0.03%	1.39%	0.26%
sqlite	70,703	0.75%	3.12%	0.41%
vorbis	9,494	0.04%	0.80%	0.55%
zlib	1,882	0.02%	2.96%	0.29%
Arithmetic mean	31,004	0.15%	1.74%	0.28%
Median	18,787	0.08%	1.60%	0.25%

Centrality measure

We measure the effect of the centrality measure on seed scheduling in this experiment by varying the centrality measure used in `K-Scheduler`. We compare Eigenvector, Degree, Katz and PageRank centrality measures. Table 4.9 shows the feature coverage results. Enabling Katz centrality improves the feature coverage by 16.54%, 23.69%, and 19.17% in arithmetic mean over the 12 FuzzBench programs, relative to Pagerank, Eigenvector, and Degree centrality, respectively. These results experimentally justify our claim from Section 4.2 that Katz centrality is most desirable for seed scheduling. However, these results also show that for some programs, other forms of centrality are a better fit such as the superior performance of Pagerank on `re2` and Degree on `vorbis`.

β parameterization

In Section 4.4, we describe how we set β based on historical mutation data. In this comparison, we see the effect of this technique by comparing `K-Scheduler` with uniform β against `K-Scheduler` with non-uniform β. Table 4.10 shows the feature coverage results. The non-

Table 4.9: **Arithmetic mean feature coverage of `K-Scheduler` with different centrality metrics.**

Programs	Katz	Pagerank	Eigenvector	Degree
freetype	**51,184**	44,394	40,723	38,332
libxml2	**39,240**	29,575	28,473	28,014
lcms	**2,886**	2,071	1,557	2,054
harfbuzz	**35,017**	28,563	26,253	27,485
libjpeg	**10,974**	9,250	10,454	8,713
libpng	**5,001**	4,804	4,505	4,923
openssl	**14,520**	13,035	13,385	13,555
openthread	**6,525**	5,201	5,380	5,298
re2	31,292	**32,309**	29,648	29,595
sqlite	**73,532**	68,328	65,538	63,997
vorbis	9,106	8,129	7,470	**9,363**
zlib	**2,711**	2,410	2,323	2,404
Arithmetic mean coverage gain	16.54%	23.69%	19.17%	
Median coverage gain	13.89%	18.99%	19.03%	

Table 4.10: **Arithmetic mean feature coverage from analyzing the effect of non-uniform β.**

Programs	Non-uniform β	Uniform β
freetype	**51,184**	40,396
libxml2	**39,240**	31,733
lcms	**2,886**	1,506
harfbuzz	**35,017**	29,380
libjpeg	**10,974**	8,834
libpng	**5,001**	4,761
openssl	**14,520**	12,542
openthread	**6,525**	5,271
re2	**31,292**	28,263
sqlite	**73,532**	64,893
vorbis	**9,106**	7,679
zlib	**2,711**	2,305
Arithmetic mean coverage gain	24.19%	
Median coverage gain	18.88%	

uniform β technique increases feature coverage by 24.19% in arithmetic mean over the 12 FuzzBench programs. These results show the utility of biasing β.

Table 4.11: **Arithmetic mean feature coverage from analyzing the effect of α.**

Programs	**0.5**	**0.25**	**0.75**	**1**
freetype	**51,184**	38,369	41,777	40,723
libxml2	**39,240**	28,644	29,992	28,473
lcms	**2,886**	1,313	1,552	1,557
harfbuzz	**35,017**	27,250	28,276	26,253
libjpeg	**10,974**	9,542	10,336	10,454
libpng	**5,001**	4,913	4,929	4,505
openssl	**14,520**	13,420	13,302	13,385
openthread	**6,525**	6,216	5,597	5,380
re2	31,292	29,590	**31,885**	29,648
sqlite	**73,532**	64,175	68,550	65,538
vorbis	**9,106**	8,092	8,066	7,470
zlib	**2,711**	2,378	2,282	2,323
Arithmetic mean coverage gain		24.53%	19.47%	23.69%
Median coverage gain		14.29%	14.74%	18.99%

Table 4.12: **Arithmetic mean feature coverage from analyzing the effect of loop removal.**

Programs	**loop removal**	**no loop removal**
freetype	**51,184**	38,646
libxml2	**39,240**	28,737
lcms	**2,886**	1,455
harfbuzz	**35,017**	28,849
libjpeg	**10,974**	10,142
libpng	**5,001**	4,846
openssl	**14,520**	13,300
openthread	**6,525**	5,430
re2	31,292	**31,609**
sqlite	**73,532**	64,560
vorbis	9,106	**9,350**
zlib	**2,711**	2,247
Arithmetic mean coverage gain		21.70%
Median coverage gain		17.03%

Visited node deletion

In Section 4.4, we describe why we remove visited nodes from the edge horizon graph. In this comparison, we experimentally justify this choice. We compare `K-Scheduler` with visited node deletions from the edge horizon graph against `K-Scheduler` with no deletions from the

Table 4.13: **Arithmetic mean feature coverage from analyzing the effect of deleting visited nodes.**

Programs	Original	Deleted
freetype	**51,184**	39,892
libxml2	**39,240**	28,973
lcms	**2,886**	1,493
harfbuzz	**35,017**	24,667
libjpeg	**10,974**	9,715
libpng	**5,001**	4,827
openssl	**14,520**	13,121
openthread	**6,525**	5,712
re2	**31,292**	29,408
sqlite	**73,532**	61,609
vorbis	**9,106**	8,020
zlib	**2,711**	2,470
Arithmetic mean coverage gain		24.13%
Median coverage gain		13.89%

edge horizon graph. Table 4.13 shows the feature coverage results. The deleted edge horizon graph improves feature coverage by 24.13% in arithmetic mean over the 12 FuzzBench programs. Therefore, this result justifies our deletion of visited nodes.

Loop Removal

In Section 4.4, we introduce our loop removal transform as a technique to mitigate the effects of loops on computing centrality. In this experiment, we measure this effect by comparing `K-Scheduler` with and without the loop removal transform. Table 4.12 shows that the loop removal transform improves edge coverage by 21.70% in arithmetic mean over the 12 FuzzBench programs, justifying our loop removal transform.

α parameterization

In this design choice experiment, we study how the choice of α affects the `K-Scheduler`'s performance. Table 4.11 summarizes our findings. As described in Section 4.4, $\alpha = 1$ treats far and close paths with equal contribution to centrality and its experimental results are worse compared to distinguishing them, showing the utility of the multipicative decay effect. We note that $\alpha = 1$

is equivalent to Eigenvector centrality as seen by comparing the relevant column from Table 4.9. Given $\alpha = 0.5$ performs best in arithmetic mean over the 12 FuzzBench programs, we pick it in our current implementation.

Result 4: Our results empirically support `K-Scheduler`'s design choices.

4.6.6 RQ5: Utility for non-evolutionary input generation

In this experiment, we show the promise of `K-Scheduler` in non-fuzzing settings, we integrate `K-Scheduler` into concolic execution seed scheduling. Concolic execution is known to incur high overhead [146, 152] during path constraint collection and solving. Hence, in concolic execution, scheduling promising seeds is crucial to its performance [153, 154]. To perform this experiment, we use the concolic executor from QSYM's latest version [146]. QSYM, a hybrid fuzzer, consists of three components, a concolic executor, a fuzzer, and a coordinator that schedules seeds for the concolic executor. Since our goal is to show the utility of `K-Scheduler` for concolic execution seed scheduling, we disabled QSYM's fuzzer and only modified its coordinator's seed scheduling algorithm to use `K-Scheduler`. We did not modify QSYM's concolic executor logic. We evaluate on the 3 programs (`size`, `libarchive` and `tcpdump`). Note we did not run on SymCC because SymCC and QSYM have the same concolic execution scheduler [152], so comparing against one is sufficient. We run `K-Scheduler` against the default seed scheduler in QSYM on the 3 real world programs for 24 hours and compare the total edge coverage. In arithmetic mean over the 10 runs, Table 4.14 shows that `K-Scheduler` improves edge coverage by 35.76%, in arithmetic mean over the 3 programs. Hence, this shows the potential promise `K-Scheduler` for seed scheduling in non-evolutionary fuzzing settings. However, we note that our results are preliminary and are inconclusive. We leave a detailed evaluation to future work.

Result 5: `K-Scheduler` increases edge coverage by 35.76%, in arithmetic mean over 3 programs, compared to QSYM's default seed scheduling strategy.

Table 4.14: **Edge coverage of concolic-execution-based seed scheduling on 3 real-world programs for 24 hours over 5 runs.**

Scheduling	`K-Scheduler`	Default
libarchive	**3,886**	3,230
size	**3,068**	2,602
tcpdump	**3,552**	2,101
Arithmetic mean coverage gain		35.76%
Median coverage gain		20.31%

4.7 Related Work

4.8 conclusion

In this paper, we introduce a new approach to seed scheduling based on centrality analysis of seeds on the CFG. Centrality measures have several desirable properties that make them a natural fit for the seed scheduling problem. We implement our approach in `K-Scheduler` and show its effectiveness in seed scheduling: increasing feature coverage by 25.89% compared to Entropic and edge coverage by 4.21% compared to the next-best AFL-based seed scheduler, in arithmetic mean on 12 Google FuzzBench programs.

Chapter 5: Related Work

5.1 Learning-based fuzzing.

Recently, there has been increasing interest in using machine learning techniques for improving fuzzers [91, 78, 155, 156, 157, 158, 159]. However, existing learning-based fuzzers model fuzzing as an end-to-end ML problem, *i.e.*, they learn ML models to directly predict input patterns that can achieve higher code coverage. By contrast, we first use NNs to smoothly approximate the program branching behavior and then leverage gradient-guided input generation technique to achieve higher coverage. Therefore, our approach is more tolerant to learning errors by ML models than the end-to-end approaches. In this paper, we empirically demonstrate that our strategy outperforms end-to-end modeling both in terms of finding bugs and achieving higher edge coverage [78].

5.2 Taint-based fuzzing.

Several evolutionary fuzzers have tried to use taint information to identify promising mutating locations [160, 161, 162, 44, 45, 7]. For example, TaintScope [160] is designed to identify input bytes that affects system/library calls and focus on mutating these bytes. Similarly, Dowser [161] and BORG [162] specifically use taint information to target detection of buffer boundary violations and buffer over-read vulnerabilities respectively. By contrast, Vuzzer [44] captures magic constants through static analysis and mutates existing values to these constants. Steelix [45] instruments binaries to collect additional taint information about comparing instructions. Finally, Angora [7] uses dynamic taint tracking to identify promising mutation locations and perform coordinate descent to guide mutations on these locations.

However, all these taint-tracking-based approaches are fundamentally limited by the fact that dynamic taint analysis incurs very high overhead while static taint analysis suffers from a high rate of false positives. Our experimental results demonstrate that NEUZZ easily outperforms existing state-of-the-art taint-based fuzzers by using neural networks to identify promising locations for

mutation.

Several fuzzers and test input generators [31, 163, 7] have tried to use different forms of gradient-guided optimization algorithms directly on the target programs. However, without program smoothing, such techniques tend to struggle and get stuck at the discontinuities.

5.3 Symbolic/concolic execution.

Symbolic and concolic execution [33, 35, 34, 164, 22] use Satisfiability Modulo Theory (SMT) solvers to solve path constraints and find interesting test inputs. Several projects have also tried to combining fuzzing with such approaches [165, 166, 21]. Unfortunately, these approaches struggle to scale in practice due to several fundamental limitations of symbolic analysis including path explosion, incomplete environment modeling, large overheads of symbolic memory modeling, etc. [36].

Concurrent to our work, NEUEX [167] made symbolic execution more efficient by learning the dependencies between intermediate variables of a program using NNs and used gradient-guided neural constraint solving together with traditional SMT solvers. By contrast, in this paper, we focus on using NNs to make fuzzing more efficient as it is by far the most popular technique for finding security-critical bugs in large, real-world programs.

5.4 Seed Scheduling

While prior work has proposed a wide range of techniques to improve fuzzing such as symbolic execution [8, 168, 169, 170, 146, 141, 171, 172], dynamic taint analysis [160, 7, 173, 174, 44] and machine learning [175, 10, 176], in this paper we focus on improving the seed scheduling component in a fuzzer. We describe prior work that has focused on improving fuzzing through seed scheduling. Seed scheduling consists of two main components: input prioritization [134, 136, 177] and the input's corresponding mutation budget (i.e., power schedule) [149, 5]. Prior seed scheduling work has prioritized seeds based on edge or path coverage [84, 5, 149, 140] as well as more security-sensitive metrics such as execution time [19, 178], exploitability [179], memory accesses [180,

181, 134], or a combination of them [136, 177] Another line of work prioritizes seeds based on call graphs [182]. In contrast, we prioritize seeds based on the entire inter-procedural CFG. While AFLGo [183] also uses the entire inter-procedural CFG, it computes the distance over the CFG for directed fuzzing and assigning a seed's mutation budget. In contrast, we approximate the count of reachable and feasible edges from a seed and use it for coverage-guided fuzzing. `SAVIOR` [171] also approximates this count but uses it for bug-driven hybrid testing. Its approximation assumes all edges are equally likely to be reachable and feasible, independent of their distance from a seed's execution path, which does not hold true for many real-world programs. In contrast, we use the multiplicative decay property of Katz centrality to reflect this behavior in real-world programs and better approximate this count. Moreover, `SAVIOR` [171]'s approximation is equivalent to setting $\alpha = 1$ (i.e, no multiplicative decay) and our design choice experiments show this approximation performs worse than K-Scheduler's default settings. Nonetheless, both K-Scheduler and `SAVIOR` utilize the mutation history information to improve their approximation.

Seed scheduling has also been a topic in other program testing techniques aside from fuzzing such as concolic execution [153, 154]. Our preliminary experiments suggest that K-Scheduler can improve seed scheduling for concolic execution.

5.5 Search-Based Software Testing

Search-Based Software Testing (SBST) applies meta-heuristic search techniques such as Hill Climbing, Simulated Annealing and Genetic Algorithms on software testing tasks [184]. The main difference between SBST and fuzzing lies in the application scope and the guiding techniques [185]. SBST is often used in unit testing due to the runtime overhead incurred by the optimization search, while fuzzing is lightweight and can be used to test the entire system. SBST leverages the fitness function to guide the search and fuzzing uses coverage feedback. One line of SBST applications is worst-case testing. [186, 187, 188] define the runtime as the fitness function and search for inputs that can maximize the runtime overhead. Another line of SBST works is to find input satisfying particular branch conditions. [189, 190, 191] measure the branch distance of the tested branch

statement as a fitness function and optimizes input towards the minimal value of branch distance i.e., the branch condition is flipped.

Conclusion

We present a data-driven approach to building adaptive and effective fuzzers. Compared with existing rule-based fuzzers, our data-driven fuzzers first model fuzzing as diverse data-centric problems, then propose algorithmic solutions to solve these problems, and in the end, the solutions yield useful knowledge to guide effective fuzzing.

The data-driven approach proposed in this dissertation has two primary limitations.

First, the NN model in NEUZZ and MTFuzz is an expensive solution to the optimization problem derivated from fuzzing. A NN model requires a large number of data samples to converge to a stable stage. However, on some small programs, it is quite hard to obtain enough training samples to bootstrap the NN model, hence hindering the further fuzzing performance. One future research direction is to use lightweight data-driven solutions, such as the hill-climbing algorithm. We have an ongoing project that leverages the hill-climbing search algorithm instead of NN model to solve the optimization problem in fuzzing without a large number of data samples since hill-climbing only needs a few data samples to perform an efficient derivative-free search scheme.

Second, the data-driven fuzzers presented in this dissertation only apply the data-driven approach to a single module of a general fuzzing framework. NEUZZ and MTFuzz employ ML to seed mutation module to generate high-quality mutations. K-Scheduler employs graph centrality analysis in the seed scheduling module to find the most promising seed. A future research direction is to explore applying the data-driven approach to seed mutation and seed scheduling at the same time. In this way, we implement a prototype data-driven fuzzer that incorporates data-driven approach in both seed mutation and seed scheduling. Our preliminary results demonstrate superior

performance than the state-of-the-art fuzzer AFL++ by a large margin.

We have shown the applicablity of our approach using three data-driven fuzzers that can greatly imporve fuzzing perforamnce.

All source code of the three fuzzers is publicaly avaialbe.

- NEUZZ: https://github.com/Dongdongshe/neuzz
- MTFuzz: https://github.com/rahlk/MTFuzz
- K-Scheduler: https://github.com/Dongdongshe/K-Scheduler

References

[1] *Ransomeware Statistics*, https://www.cloudwards.net/ransomware-statistics, 2023.

[2] *twitter news*, https://www.bleepingcomputer.com/news/security/hacker-claims-to-be-selling-twitter-data-of-400-million-users/, 2023.

[3] *bank news*, https://www.upguard.com/blog/biggest-data-breaches-financial-services, 2023.

[4] M. Zalewski, *American Fuzzy Lop (AFL) README*, http://lcamtuf.coredump.cx/afl/README.txt, 2018.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2016, pp. 1032–1043.

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[7] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," 2018.

[8] C. Cadar, D. Dunbar, and D. 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*, ser. OSDI'08, San Diego, California: USENIX Association, 2008.

[9] S. Canakci *et al.*, "Processorfuzz: Processor fuzzing with control and status registers guidance," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, Los Alamitos, CA, USA: IEEE Computer Society, 2023.

[10] Dongdong She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[11] Dongdong She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: efficient dynamic taint analysis with neural networks," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[12] Dongdong She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MTFuzz: fuzzing with a multi-task neural network," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[13] Dongdong She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[14] Darpa, *Cyber Grand Challenge*, `http://archive.darpa.mil/cybergrandchallenge/`, 2016.

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

[16] *Google fuzzing service for OS finds 1k bugs in five months*, `http://www.eweek.com/cloud/google-fuzzing-service-for-os-finds-1k-bugs-in-five-months`, 2017.

[17] *Guided in-process fuzzing of Chrome components*, `https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html`, 2016.

[18] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "KameleonFuzz: Evolutionary fuzzing for black-box XSS detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, ACM, 2014, pp. 37–48.

[19] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Dallas, Texas, USA, 2017, pp. 2155–2168.

[20] K Serebryany, *libFuzzer – a library for coverage-guided fuzz testing*, `http://llvm.org/docs/LibFuzzer.html`, 2018.

[21] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," *In Proceedings of the Network and Distributed System Security Symposium*, vol. 16, pp. 1–16, 2016.

[22] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, "Automated whitebox fuzz testing.," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 8, 2008, pp. 151–166.

[23] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," *Proceedings of the IEEE Symposium on Security & privacy*, 2018.

[24] B. P. Miller, "Fuzz testing of application reliability," *UW-Madison Computer Sciences*, 2007.

[25] S Hocevar, *Zzuf—multi-purpose fuzzer*, `http://caca.zoy.org/wiki/zzuf`, 2011.

[26] D. W. Zingg, M. Nemec, and T. H. Pulliam, "A comparative evaluation of genetic and gradient-based algorithms applied to aerodynamic optimization," *European Journal of Computational Mechanics/Revue Européenne de Mécanique Numérique*, vol. 17, no. 1-2, pp. 103–126, 2008.

[27] R. Horst and P. M. Pardalos, *Handbook of global optimization*. Springer Science & Business Media, 2013, vol. 2.

[28] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[29] D. Parnas, "Software aspects of strategic defense systems," *Communications of the ACM*, vol. 28, no. 12, pp. 1326–1335, 1985.

[30] S. Chaudhuri and A. Solar-Lezama, "Smoothing a program soundly and robustly," in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 277–292.

[31] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[32] S. Chaudhuri and A. Solar-Lezama, "Smooth interpretation," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 279–291, 2010.

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

[34] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 30, 2005, pp. 263–272.

[35] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 8, 2008, pp. 209–224.

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

[37] C. Cadar *et al.*, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, IEEE, 2011, pp. 1066–1071.

[38] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *ACM Sigplan Notices*, ACM, vol. 40, 2005, pp. 213–223.

[39] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2003, pp. 553–568.

[40] K.-I. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural networks*, vol. 2, no. 3, pp. 183–192, 1989.

[41] B. Dolan-Gavitt *et al.*, "LAVA: Large-scale automated vulnerability addition," in *Proceedings of the IEEE Symposium on Security & Privacy*, 2016, pp. 110–121.

[42] DARPA, *Cyber Grand Challenge Repository*, `https://github.com/cybergrandchallenge`, 2017.

[43] S. Wright and J. Nocedal, "Numerical optimization," *Springer Science*, vol. 35, no. 67-68, p. 7, 1999.

[44] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed Systems Security Conference (NDSS)*, 2017.

[45] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: Program-state based binary fuzzing," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[46] S. Chaudhuri, S. Gulwani, and R. Lublinerman, "Continuity and robustness of programs," *Communications of the ACM*, vol. 55, no. 8, pp. 107–115, 2012.

[47] D. P. Bertsekas and A. Scientific, *Convex optimization algorithms*. Athena Scientific Belmont, 2015.

[48] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A limited memory algorithm for bound constrained optimization," *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, 1995.

[49] J. Nocedal, "Updating quasi-newton matrices with limited storage," *Mathematics of computation*, vol. 35, no. 151, pp. 773–782, 1980.

[50] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*, vol. 24, Elsevier, 1989, pp. 109–165.

[51] A. R. Conn, K. Scheinberg, and P. L. Toint, "Recent progress in unconstrained nonlinear optimization without derivatives," *Mathematical programming*, vol. 79, no. 1-3, p. 397, 1997.

[52] A. Conn, K. Scheinberg, and P. Toint, "On the convergence of derivative-free methods for unconstrained optimization," *Approximation theory and optimization: tributes to MJD Powell*, pp. 83–108, 1997.

[53] M. J. Powell, "Uobyqa: Unconstrained optimization by quadratic approximation," *Mathematical Programming*, vol. 92, no. 3, pp. 555–582, 2002.

[54] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM review*, vol. 45, no. 3, pp. 385–482, 2003.

[55] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the Advances in neural information processing systems (NIPS)*, 2012, pp. 1097–1105.

[56] A. Garg and K. Tai, "Comparison of statistical and machine learning methods in modelling of data with multicollinearity," *International Journal of Modelling, Identification and Control*, vol. 18, no. 4, pp. 295–312, 2013.

[57] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps.," in *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[58] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[59] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proceedings of the IEEE European Symposium on Security & Privacy*, 2015.

[60] J. Yosinski, J. Clune, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," in *2015 ICML Workshop on Deep Learning*, 2015.

[61] A. Mahendran and A. Vedaldi, "Understanding deep image representations by inverting them," in *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[62] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*, vol. 24, Elsevier, 1989, pp. 109–165.

[63] W. C. Abraham and A. Robins, "Memory retention–the synaptic stability versus plasticity dilemma," *Trends in neurosciences*, vol. 28, no. 2, pp. 73–78, 2005.

[64] G. E. Hinton and D. C. Plaut, "Using fast weights to deblur old memories," in *Proceedings of the ninth annual conference of the Cognitive Science Society*, 1987, pp. 177–186.

[65] J. Kirkpatrick *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, 2017.

[66] C. Fernando *et al.*, "Pathnet: Evolution channels gradient descent in super neural networks," *arXiv preprint arXiv:1701.08734*, 2017.

[67] B. Ren, H. Wang, J. Li, and H. Gao, "Life-long learning based on dynamic combination model," *Applied Soft Computing*, vol. 56, pp. 398–404, 2017.

[68] T. J. Draelos *et al.*, "Neurogenesis deep learning: Extending deep networks to accommodate new classes," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2017, pp. 526–533.

[69] B. Goodrich and I. Arel, "Unsupervised neuron selection for mitigating catastrophic forgetting in neural networks," in *Proceedings of the International Symposium on Circuits and Systems*, Citeseer, 2014, pp. 997–1000.

[70] A. Robins, "Catastrophic forgetting, rehearsal and pseudorehearsal," *Connection Science*, vol. 7, no. 2, pp. 123–146, 1995.

[71] R. Kemker, A. Abitino, M. McClure, and C. Kanan, "Measuring catastrophic forgetting in neural networks," *arXiv preprint arXiv:1708.02072*, 2017.

[72] *Keras: The python deep learning library*, `https://keras.io/`, 2018.

[73] *An open source machine learning framework for everyone*, `https://www.tensorflow.org/`, 2018.

[74] *Address sanitizer, thread sanitizer, and memory sanitizer*, `https://github.com/google/sanitizers`, 2018.

[75] *Undefined behavior sanitizer*, `https://clang.llvm.org/docs/UndefinedBehaviorSanitihtml`, 2018.

[76] B. Dolan-Gavitt, *Of Bugs and Baselines*, `http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html`, 2018.

[77] *DARPA challenge for Linux, Windows, and macOS*, `https://github.com/trailofbits/cb-multios`, 2017.

[78] M. Rajpal, W. Blum, and R. Singh, "Not All Bytes Are Equal: Neural Byte Sieve for Fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[79] M. Zalewski, "American fuzzy lop," *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[80] V. J. Manes *et al.*, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[81] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proc. 41st International Conf. Software Engineering*, ser. Icse '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 724–735.

[82] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, "Diffuzz: Differential fuzzing for side-channel analysis," in *Proc. 41st Intl. Conf. Software Engineering*, ser. Icse '19, Montreal, Quebec, Canada: IEEE Press, 2019, pp. 176–187.

[83] W. Z. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "Slf: Fuzzing without valid seed inputs," in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19)*, Acm, 2018, pp. 712–723.

[84] C. Lemieux and K. Sen, "Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, Acm, 2018.

[85] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program learning," *In Proceedings of the IEEE Symposium on Security & Privacy*, 2019.

[86] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proc. 29th ACM SIGPLAN Conf. Programming Language Design and Implementation*, ser. Pldi '08, Tucson, AZ, USA: Acm, 2008, pp. 206–215.

[87] A. Arya and C. Neckar, "Fuzzing for security," vol. 1, p. 2013, 2012.

[88] C. Evans, M. Moore, and T. Ormandy, "Fuzzing at scale," *Google Online Security Blog*, 2011.

[89] M. Moroz and K. Serebryany, *Guided in-process fuzzing of chrome components*, 2016.

[90] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in *Proc. 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. Foundations of Software Engineering*, Acm, 2018, pp. 213–223.

[91] P. Godefroid, H. Peleg, and R. Singh, "Learn&Fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[92] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*, Acm, 2018, pp. 259–269.

[93] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *In Proceedings of the IEEE Symposium on Security & Privacy*, pp. 711–725, 2018.

[94] D. She, Y. Chen, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," *arXiv preprint arXiv:1907.03756*, 2019.

[95] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A Review of Machine Learning Applications in Fuzzing," 2019. arXiv: 1906.11133.

[96] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, pp. 1–10, 2017. arXiv: 1711.04596.

[97] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[98] P. McMinn and M. Holcombe, "Hybridizing Evolutionary Testing with the Chaining Approach," in *Lecture Notes in Computer Science*, 2004, pp. 1363–1374.

[99] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *3rd Intl. Conf. Software Testing, Verification and Validation*, Ieee, 2010, pp. 205–214.

[100] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Ieee, 2011, pp. 153–163.

[101] A. Pachauri and G. Srivastava, "Automated test data generation for branch testing using genetic algorithm: An improved approach using branch ordering, memory and elitism," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1191–1208, 2013.

[102] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, Sep. 2019, pp. 1–15.

[103] A. Argyriou, T. Evgeniou, and M. Pontil, "Convex multi-task feature learning," *Machine learning*, vol. 73, no. 3, pp. 243–272, 2008.

[104] P. Gong, J. Zhou, W. Fan, and J. Ye, "Efficient multi-task feature learning with calibration," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 761–770.

[105] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, 2008, pp. 160–167.

[106] J. Binge and A. Sogaard, "Identifying beneficial task relations for multi-task learning in deep neural networks," *15th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2017 - Proceedings of Conference*, vol. 2, no. 2016, pp. 164–169, 2017. arXiv: 1702.08303.

[107] T. Standley, A. R. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese, "Which tasks should be learned together in multi-task learning?" *arXiv preprint arXiv:1905.07553*, 2019.

[108] R. Caruana, "Algorithms and applications for multitask learning," in *Icml*, 1996, pp. 87–95.

[109] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[110] A. R. Zamir, A. Sax, W. Shen, L. J. Guibas, J. Malik, and S. Savarese, "Taskonomy: Disentangling task transfer learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 3712–3722.

[111] Y. Zhang and Q. Yang, "A survey on multi-task learning," *arXiv preprint arXiv:1707.08114*, pp. 1–20, 2017. eprint: 1707.08114.

[112] I. Kokkinos, "Ubernet: Training a universal convolutional neural network for low-, mid-, and high-level vision using diverse datasets and limited memory," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6129–6138.

[113] J. Uhrig, M. Cordts, U. Franke, and T. Brox, "Pixel-level encoding and depth layering for instance-level semantic labeling," in *German Conference on Pattern Recognition*, Springer, 2016, pp. 14–25.

[114] M. Teichmann, M. Weber, M. Zoellner, R. Cipolla, and R. Urtasun, "Multinet: Real-time joint semantic reasoning for autonomous driving," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2018, pp. 1013–1020.

[115] Y. Liao, S. Kodagoda, Y. Wang, L. Shi, and Y. Liu, "Understand scene categories by objects: A semantic regularized scene classifier using convolutional neural networks," in *2016 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2016, pp. 2318–2325.

[116] J. Baxter, "A bayesian/information theoretic model of learning to learn via multiple task sampling," *Machine learning*, vol. 28, no. 1, pp. 7–39, 1997.

[117] P. McMinn and M. Holcombe, "Hybridizing evolutionary testing with the chaining approach," in *Genetic and Evolutionary Computation – GECCO 2004*, K. Deb, Ed., Springer Berlin Heidelberg, 2004.

[118] A. Kumar, P. Sattigeri, and A. Balakrishnan, "Variational inference of disentangled latent concepts from unlabeled observations," *arXiv preprint arXiv:1711.00848*, 2017.

[119] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets," in *Advances in neural information processing systems*, 2016, pp. 2172–2180.

[120] T. D. Kulkarni, W. F. Whitney, P. Kohli, and J. Tenenbaum, "Deep convolutional inverse graphics network," in *Advances in neural information processing systems*, 2015, pp. 2539–2547.

[121] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[122] A. B. Owen, *Monte Carlo theory, methods and examples*. 2013.

[123] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[124] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[125] *Undefinedbehaviorsanitizer*, `https://clang.llvm.org/docs/UndefinedBehaviorSaniti html`, 2020.

[126] S. Gan *et al.*, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA: USENIX Association, Aug. 2020.

[127] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," *In Proceedings of the Network and Distributed System Security Symposium*, 2019.

[128] *Xmlwf xml parser*, `https://libexpat.github.io/`, 2020.

[129] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.

[130] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

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

[132] C. Lyu *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, USENIX Association, 2019.

[133] V. J. M. Manès, S. Kim, and S. K. Cha, "Ankou: Guiding grey-box fuzzing towards combinatorial difference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, Seoul, South Korea: Association for Computing Machinery, 2020.

[134] Y. Wang *et al.*, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *NDSS*, 2020.

[135] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017.

[136] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15, ISBN: 978-1-939133-07-6.

[137] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1099–1114.

[138] *Honggfuzz - A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options.* `https://github.com/google/honggfuzz`, 2021.

[139] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021, Virtual, Denmark: Association for Computing Machinery, 2021.

[140] T. Yue *et al.*, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020.

[141] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proceedings of the IEEE Symposium on Security & Privacy*, 2018.

[142] R. Jacob, D. Koschützki, K. A. Lehmann, L. Peeters, and D. Tenfelde-Podehl, "Algorithms for centrality indices," in *Network Analysis: Methodological Foundations*, U. Brandes and T. Erlebach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 62–82.

[143] M. E. J. Newman, "Mathematics of networks," in *The New Palgrave Dictionary of Economics*. London: Palgrave Macmillan UK, 2016.

[144] L. Lü, D. Chen, X.-L. Ren, Q.-M. Zhang, Y.-C. Zhang, and T. Zhou, "Vital nodes identification in complex networks," *Physics Reports*, vol. 650, pp. 1–63, 2016.

[145] E. Nathan and D. A. Bader, "A dynamic algorithm for updating katz centrality in graphs," in *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, ser. ASONAM '17, Sydney, Australia: Association for Computing Machinery, 2017, 149–154.

[146] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[147] *Whole Program LLVM*. `https://github.com/travitch/whole-program-llvm`, 2021.

[148] *Networkit: Large-scale Network Analysis*. `https://networkit.github.io/`, 2021.

[149] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020.

[150] *FuzzBench guidelines about setting havoc mode for AFL evaluation*, `https://github.com/google/fuzzbench/blob/master/fuzzers/afl/fuzzer.py#L113`, 2021.

[151] M. Wu *et al.*, "One fuzzing strategy to rule them all," 2022.

[152] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" In *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 181–198, ISBN: 978-1-939133-17-5.

[153] Y. Chen, M. Ahmadi, R. Mirzazade farkhani, B. Wang, and L. Lu, "MEUZZ: Smart seed scheduling for hybrid fuzzing," in *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID'20, 2020.

[154] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *NDSS*, 2019.

[155] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, Ieee, 2017, pp. 579–594.

[156] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[157] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations," in *Proceedings of the IEEE Symposium on Security & Privacy*, 2017, pp. 521–538.

[158] K. Böttinger, P. Godefroid, and R. Singh, "Deep Reinforcement Fuzzing," *arXiv preprint arXiv:1801.04589*, 2018.

[159] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *arXiv preprint arXiv:1711.02807*, 2017.

[160] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proceedings of the IEEE Symposium on Security & Privacy*, 2010.

[161] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[162] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos, "The BORG: Nanoprobing binaries for buffer overreads," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015.

[163] L. Szekeres, "Memory corruption mitigation via hardening and testing," Ph.D. dissertation, Stony Brook University, 2017.

[164] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary Linux programs," in *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[165] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proceedings of the IEEE Symposium on Security & Privacy*, 2015.

[166] J. Fietkau, B. Shastry, and J.-P. Seifert, *KleeFL - seeding fuzzers with symbolic execution*, `https://github.com/julieeen/kleefl`, 2017.

[167] S. Shen, S. Ramesh, S. Shinde, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: The feasibility of an inductive approach to symbolic execution," *arXiv preprint arXiv:1807.00575*, 2018.

[168] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," 2005.

[169] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[170] Shellphish, *angr: a powerful and user-friendly binary analysis platform*, `https://github.com/angr/angr`, 2018.

[171] Y. Chen *et al.*, "Savior: Towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1580–1596.

[172] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[173] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484.

[174] S. Gan *et al.*, "Greyone: Data flow sensitive fuzzing," in *USENIX Security Symposium*, 2020.

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

[176] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 2255–2269, ISBN: 978-1-939133-17-5.

[177] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," in *NDSS*, 2021.

[178] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," ser. ISSTA 2018, Amsterdam, Netherlands: Association for Computing Machinery, 2018.

[179] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013.

[180] N. Coppik, O. Schwahn, and N. Suri, "Memfuzz: Using memory accesses to guide fuzzing," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019*.

[181] C. Wen *et al.*, "Memlock: Memory usage guided fuzzing," in *ICSE '20: 42nd International Conference on Software Engineering*.

[182] Y. Li *et al.*, "Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, Tallinn, Estonia: Association for Computing Machinery, 2019.

[183] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017.

[184] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 153–163.

[185] G. Guizzo and S. Panichella, "Fuzzing vs sbst: Intersections & differences," 2023.

[186] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, pp. 275–298, 1998.

[187] J. Wegener, H. Sthamer, B. Jones, and D. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, pp. 127–135, Jun. 1997.

[188] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, 1998, pp. 134–143.

[189] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[190] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 205–214.

[191] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation," in *Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No.98EX239)*, 1998, pp. 285–288.