MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing

Yaohui Chen
Northeastern University
yaohway@gmail.com

Mansour Ahmadi
Northeastern University
Mansosec@gmail.com

Reza Mirzazade farkhani
Northeastern University
reza699@ccs.neu.edu

Boyu Wang
Stony Brook University
boywang@cs.stonybrook.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Seed scheduling highly impacts the yields of hybrid fuzzing. Existing hybrid fuzzers schedule seeds based on fixed heuristics that aim to predict input utilities. However, such heuristics are not generalizable as there exists no one-size-fits-all rule applicable to different programs. They may work well on the programs from which they were derived, but not others.

To overcome this problem, we design a Machine learning-Enhanced hybrid fUZZing system (MEUZZ), which employs supervised machine learning for adaptive and generalizable seed scheduling. MEUZZ determines which new seeds are expected to produce better fuzzing yields based on the knowledge learned from past seed scheduling decisions made on the same or similar programs. MEUZZ extracts a series of features for learning via code reachability and dynamic analysis, which incurs negligible runtime overhead (in microseconds). MEUZZ automatically infers the data labels by evaluating the fuzzing performance of each selected seed. As a result, MEUZZ is generally applicable to, and performs well on, various kinds of programs.

Our evaluation shows MEUZZ significantly outperforms the state-of-the-art grey-box and hybrid fuzzers, achieving 27.1% more code coverage than QSYM. The learned models are reusable and transferable, which boosts fuzzing performance by 7.1% on average and improves 68% of the 56 cross-program fuzzing campaigns. When fuzzing 8 well-tested programs under the same configurations as used in previous work, MEUZZ discovered 47 deeply hidden and previously unknown bugs, among which 21 were confirmed and fixed by the developers.

1 Introduction

Hybrid testing as a research topic has attracted tremendous attention and made significant contributions to bug discovery. For instance, the winning teams in the DARPA Cyber Grand Challenge [6] all used hybrid testing [17]. Compared with plain fuzzing, hybrid testing features an extra concolic execution component, which revisits the fuzzed paths, solves the path conditions, and tries to uncover new paths.

One key challenge in hybrid testing is to recognize high-utility seeds (*i.e.,* seeds of high potential to guide concolic execution to crack complex conditions guarding more coverage and bugs). Prioritizing such seeds allows the hybrid fuzzer to achieve higher code coverage more quickly, and in turn, discover more bugs in a fixed time frame. Moreover, this prioritization matters in practice because the concolic execution engine usually has limited time budget and can explore only a (small) subset of all fuzzer-generated seeds. Being able to estimate seed utility allows hybrid fuzzers to use concolic execution more efficiently.

The existing work [8, 19, 25, 33, 49, 54, 55] uses purely heuristic-based seed selection. For example, some prefer seeds with smaller sizes while some value those that lead to new code coverage. These heuristics, despite their simplicity, do not perform equally well across different kinds of programs and are not universally suitable for all programs. Contradicting the previous belief [8, 13, 33], our experiments show that seeds leading to new coverage sometimes have the lowest utility (§6.3). Similarly, previous work [33, 54] suggested that smaller seeds should have higher utility, which however is not true in certain programs as our evaluation shows. As a result, these simple and fixed heuristics may cause non-optimum seed selections, overwhelming the concolic engine with low-utility seeds and slowing down bug discovery.

Compared to heuristics, Machine Learning (ML) algorithms, when trained with sufficient data, can discover complex and implicit patterns automatically [43]. We show that seed selection strategies that are automatically learned based on individual programs perform better than manually defined heuristics that fail to consider all kinds of programs. As our experiment shows that the influence of each feature varies across different programs, suggesting that no single feature (or rule) can work well for all programs. ML-based seed selection avoids the need for manually designing, testing, and reasoning about seed selection rules, which can be daunting, non-scalable, or even impossible when the volume of data to be analyzed is overwhelming.

In this paper, we introduce MEUZZ, an ML-enhanced hybrid fuzzing system. Unlike existing work, which schedule

seeds using simple heuristics derived from a particular set of test programs, MEUZZ uses ML and a set of static and dynamic features, computed from seeds and individual programs, to predict seed utility and perform seed scheduling. MEUZZ also has a built-in evaluation module that measures prediction quality for continuous learning and improvement. To the best of our knowledge, MEUZZ is the first work [42] that applies ML to seed prioritization and scheduling.

To effectively apply ML to seed scheduling for hybrid fuzzing, our design of MEUZZ pays special attention to two ML tasks: *feature engineering* and *data labeling*. While these are the essential steps to bootstrap ML, they could be time-consuming and thus too costly or impractical to be included in the fuzzing workflow. For instance, feature extraction can be very slow if it requires heavy computation or extensive data collection. Moreover, it is not straightforward to quantify seed utility, which is essential for labeling.

To tackle the aforementioned challenges, we first engineer a set of lightweight features based on code reachability and dynamic analysis. Second, we propose a labeling method using the input descendant tree to quantify the utility of a seed. Our evaluation shows that MEUZZ takes only 5μs on average to extract an individual feature. It also confirms that the descendant tree of a seed accurately reflects seed utility.

Collecting data and training a new model for every program might not be economical or necessary. Therefore, we investigate the feasibility of model reusability and transferability to answer the question: *Is a learned model transferable to different fuzzing configurations or programs?* Since the learning is designed to predict the likelihood of seeds triggering bugs, rather than any specifics of the fuzzed program, a model learned by MEUZZ turns out to be applicable beyond the program from which the model is learned.

We compare MEUZZ with the state-of-the-art fuzzers [19, 24, 33] as well as the most recent hybrid testing systems [25, 54]. The results, based on a set of real-world benchmark programs, show that MEUZZ achieves much higher code coverage than the tested fuzzers that use simple seed selection heuristics. Particularly MEUZZ expands the code coverage by as much as 27.1% compared to QSYM, the start-of-the-art hybrid fuzzing system. The experiments also show that the prediction models learned by MEUZZ have good reusability and transferability. The reused models boost the coverage by 7.1% on average. The transplanted models improve fuzzing performance in 38 out of 56 cases (67.9% of cases), with 10 cases seeing more than 10% improvement.

This paper makes the following contributions.

- *Effective and generalizable approach.* We design, implement, and evaluate MEUZZ, the first system that applies machine learning to the seed selection stage of hybrid fuzzing. MEUZZ performs better and is more widely applicable than heuristic-based seed selection.
- *Practical feature and label engineering.* We address two major challenges, namely feature engineering and label

inference, when applying ML to seed selection in hybrid fuzzing. Our feature selection and extraction allow for online/continuous learning. They are compatible with the existing hybrid fuzzing workflow and require no changes to either fuzzers or concolic execution engines. We also propose an automatic label inference method based on seed descendant trees.

- *Reusable and transferable ML models.* Our seed selection models demonstrate strong *reusability* and *transferability*. As a result, MEUZZ can reuse a well-trained model on different programs (or different fuzzing configurations) to quickly bootstrap the fuzzing campaign and continuously improve and adapt the model to the current program or configuration.
- *Open-Source.* The full implmentation of MEUZZ will be open-sourced after acceptance.

2 Motivation

The seed selection (or scheduling) in fuzzing aims to solve this problem: given a program and a set of seeds, in which order the fuzzer should test the seeds to maximize the gain during a fixed period. Seed selection plays a critical role in hybrid fuzzing because the concolic execution engine can only explore an (often small) subset of the seeds due to time constraints. Hence, hybrid fuzzing cannot fully benefit from concolic execution if the seed selection is not optimal.

Why seed selection is important for hybrid fuzzing: Hybrid fuzzers without a seed scheduling mechanism (*e.g.,* Driller [49]) have to explore all inputs. This "brute force" strategy has two main drawbacks. First, concolic engines cannot keep up with the speed of plain fuzzing because they run relatively slowly and often encounter path explosions and timeouts. As an experiment, we used QSYM [54] to fuzz a set of real-world benchmark programs. QSYM is one of the state-of-the-art concolic execution engines for hybrid testing[1]. As shown in Figure 1, for a continuous 24-hour run, QSYM was only able to explore 23.1% of the seeds in fuzzer's queue.

Second, a seed selection strategy affects fuzzing results drastically. A naive strategy delays a fuzzer's exploration of interesting program locations, and sometimes, prematurely forces the fuzzer to skip deep program paths and states. Some recent research [19, 25, 51, 54, 55] studied a few seed selection heuristics of various levels of sophistication. In their experiments, fuzzers using these seed selection heuristics produce better results (*e.g.,* higher code coverage) than fuzzers with naive or no strategies.

Why exploring machine learning for seed selection: All the existing seed selection strategies are based on manually defined heuristics. Although performing well on their selected benchmarks, these strategies may not be generalizable to, or suitable for, other programs. For instance, DigFuzz [55] and AFLFast [19] prioritize seeds with less explored paths

---

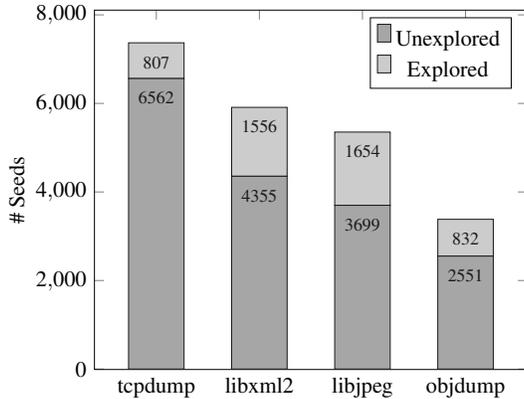[1]Reportedly, QSYM is 3x faster than Driller [54].

Figure 1: The total number of inputs explored by the concolic execution engine of QSYM in 24 hours. On average, only 23.1% of the inputs were explored by the concolic execution even though the engine was continuously running.

by fuzzer. Savior [25] prefers seeds dominating more UBSan-labeled code paths. QSYM [13] prioritizes seeds with smaller sizes. These heuristics are all based on intuition or empirical observations gained from limited test cases or benchmarks.

A biased or unsuitable seed selection strategy delays or prevents fuzzers' exploration of deep program states or the discovery of bugs. For instance, QSYM [13] and ProFuzzer [53] prioritize inputs with smaller sizes. Their developers observed in their evaluation benchmarks that smaller inputs lead to higher code coverage. However, as [25] pointed out, QSYM fails to explore a large chunk of code in program who (a program in the LAVA-M benchmark [27]) due to the unsuitable seed selection strategy (*i.e.,* only inputs larger than a certain size can trigger the vulnerable functions in this case).

This clearly indicates that fixed seed selection heuristics can hardly be suitable for a wide set of programs (See Figure 10).

Due to the diverse scheduling scenarios, modern fuzzers (*e.g.,* AFL [2], QSYM [13]) often employ multiple heuristics for seed prioritization. Unfortunately, relying on human efforts to learn and generalize seed selection strategies, as the previous work did, is not scalable to a large number of features. In fact, it is just infeasible to manually reason about a big set of selection criteria when the number of features and the amount of data to be analyzed become overwhelming (*e.g.,* OSS-Fuzz generates four trillion seeds per week [4] for different programs).

In contrast to heuristics, machine learning (ML) is good at discovering underlying connections between data attributes [36, 43]. ML can be applied to seed selection because, as shown by existing studies, the selection strategies are indeed learnable (*i.e.,* exhibiting statistically significant patterns). With sufficient learning data, ML can not only infer the importance of different features but also mine the integration rules at scale.

MEUZZ is the first to explore the ML-based, data-driven approach to seed selection in hybrid fuzzing. Our result confirms that automatically and continuously learned seed selection strategies are more suitable for individual programs.

3 Background

Hybrid fuzzing [25, 49, 54] combines fuzzing and concolic execution to address the deficiencies of both the approaches. Figure 2 shows an overview of a general hybrid fuzzing framework. The whole system consists of three major components: fuzzer, concolic testing, and coordinator. For the sake of brevity, we refer the interested readers to [2, 8, 23, 29] for the technical details of fuzzing and concolic execution.
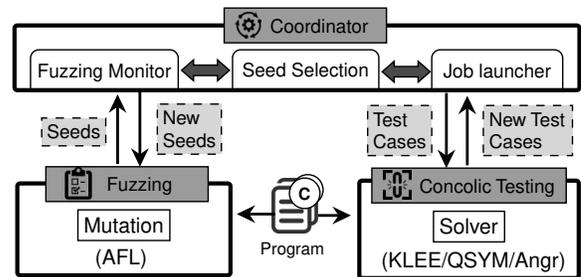
3.1 Hybrid Fuzzing

Figure 2: General hybrid fuzzing workflow.

We dissect the coordinator component as it is less discussed in the literature and is the focus of this work. The coordinator is a middleware that regulates the other two components. Its major tasks include (*i*) monitoring the fuzzer to decide when to launch the concolic execution engine, and (*ii*) prepare the running environments for concolic testing; and (*iii*) select and filter inputs that flow between fuzzer and concolic executor.

The seed selection module in the coordinator needs to decide which seeds in the fuzzer's queue should be transferred to the concolic testing first (*i.e.,* Seed utility prediction phase). Before launching the concolic execution, the coordinator needs to rank all inputs in the fuzzer's queue based on their utility. The *utility* of seed should correspond to the estimation of its power to produce additional coverage if it is selected to fuzz. As we mentioned in Section 2, current methods use various heuristics to achieve this prioritization goal.

3.2 Supervised Machine Learning

Supervised ML is the task of learning from labeled data and applies the knowledge to unknown data. Classification and regression are two foremost categories of such algorithms. While classification is used for predicting categorical responses, regression predicts a numerical value to the new data based on previously observed data. Supervised learning has shown thriving employment in application security, including bug discovery [30, 35, 37].

Supervised machine learning can be either online or offline. The difference between these two lies in how models are updated.

Online learning: Some learning environments can change from second to second and their models need to get updated

(or relearned) as fast as they see a new sample. Under this constraint of time, online learning shows promises by only considering the new data to update the model, which makes it an efficient approach. Basically, most learning algorithms that are compatible (but not limited) with the standard optimization algorithms like stochastic gradient descent (SGD) can learn incrementally.

Offline learning: In contrast to online learning, the models in offline learning need to be retrained with the whole dataset as newer data appear. One of the successful examples of offline supervised learning techniques is Random Forest (RF), which has shown promising achievements, and in certain domains, has even better performance than neural networks [28]. In addition to RF, deep learning has been shown success in different domains; however, they are usually practiced on unstructured data such as images and they require a relatively larger amount of data to perform well [16]. Moreover, such techniques need high computational power and longer time to train; hence they are not suitable for the online fuzzing workflow.

4 System Design

4.1 System Overview

MEUZZ is the first machine learning-based hybrid fuzzer that learns from the previously observed seeds and identifies which kinds of seeds have the potentials to more effectively explore the program being tested.

Figure 3 shows an overview of MEUZZ. MEUZZ starts fuzzing (❶) a program with pre-defined or empty seeds. It then extracts features (❷) from the program as well as the seeds (§4.3) to model coverage gains. Such features are used to predict (❸) the coverage that unknown seeds may provide (§4.5). Concolic engine (❹) then receives the potentially influential seeds from the prior step and produces mutated seeds. Next, MEUZZ guides the fuzzer to use these seeds and their generated mutants–by the evolutionary algorithms–to continually test the program. In the beginning, the prediction model is randomly initialized, so the prediction quality is uncertain. But as fuzzing continues, the model gets improved and will provide a more reliable prediction. MEUZZ updates the seed selection model in three steps. First, it infers the descendent trees (❺) of those seeds selected to the concolic engine in (❹); then, it derives a label (❻) based on the descendent trees of the previously selected seed (§4.4); finally, it updates or retrains the model (❼) depending on the type of learning process (§4.5, §4.6).

4.2 System Requirements

MEUZZ aims to predict the seed utility in a more accurate and generalizable fashion than the existing heuristic-based approaches while keeping the fuzzing efficiency intact. One of the steps that contribute the most in achieving these goals is feature extraction. MEUZZ can potentially derive various semantic features because it has access to complex program
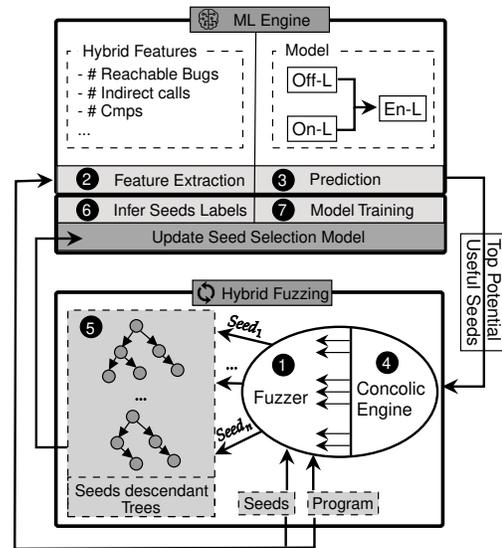

Figure 3: System overview of MEUZZ. The coordinator is extended with a ML engine, which consists of 4 modules – Feature extraction, label inference, prediction and training modules. During fuzzing, utility prediction and model training are carried out consecutively. After extracting features for inputs in the fuzzer's queue, the ML engine can predict their utilities based on the current model. Then, with the seed labels inferred from previously selected seeds, the model is trained iteratively with the new data.

structures, such as the Control Flow Graph (CFG) with sanitizer instrumentations. However, there are some challenges that MEUZZ may encounter during feature extraction because it requires to adapt the ML engine to the online-style fuzzing workflow. To cope with such challenges, the feature engineering stage should meet the following requirements (*R*1–*R*3).

R1 - Utility Relevant: The ultimate goal of fuzzing is higher code coverage as well as discovering a higher number of hidden bugs. The features should reflect the characteristics that may improve such measures. For instance, how much a seed is likely to trigger more potential bugs or how much unexplored code a mutated seed will reach during its execution. As it is obvious, a seed is only meaningful in the context, which is the program it is executed upon. Accordingly, feature extraction needs to consider the seed and the program as a bundle.

R2 - Seed-/Program-Agnostic: To achieve generalizability, the features should be seed-/program-agnostic. If a feature is target-dependent, it downgrades the ability to generalize. For example, one could engineer a boolean feature based on the magic number that shows if a generated seed is genuine or not. Although this feature looks useful to ignore invalid seeds for fuzzing a specific program, it needs to be customized for fuzzing different programs as the inputs' formats change. Contrarily, "meta properties" like the execution path triggered by the input are more preferable, as it is a universally usable characteristic regardless of the program.

R3 - Online Friendly: To keep the efficiency comparable to heuristic-based approaches, it is not only important how fast each feature can be extracted, but also the number of features is concerned during model construction. If the features are both light-weight and effective, it is assured that the coordinator will not be blocked from launching the concolic executor and at the same time able to construct meaningful models to predict the seed utility. As a result, suitable features should strike a balance between analysis richness (*i.e.,* how informative is the analysis result) and computation complexity (*i.e.,* what is the time complexity for the analysis).

4.3 Feature Engineering

The aforementioned requirements (*R*1–*R*3) guide us to engineer the following list of features. We discuss them in four categories.

Bug-triggering: Inspired by existing research [25], we use the number of reachable sanitizer instrumentations as guidance for measuring how likely bugs can be triggered. As sanitizer instrumentations are based on sound analysis (*i.e.,* no missed bugs), it provides a good over-approximation when trying to quantify the number of bugs that can be found. Hence, we extract these two features:

1. *Count of reachable sanitizer instrumentations*: For all branches throughout the path triggered by a given seed, the number of reachable sanitizer instrumentations is computed and then sum up. For instance, there are two branches in the left example of Figure 4. There are six potential bugs by following the branches, so the value for this feature is six.

2. *Count of reached sanitizer instrumentations*: For all branches throughout the path triggered by a given seed, we sum up the number of reached sanitizer instrumentations by the fuzzer. The major difference between this feature and the prior one is that this feature reflects the expectation of *immediately* solvable sanitizer bugs, while the former feature is an *indirect* reflection. For instance, the value of this feature in the right example of Figure 4 is two because the potential bugs can be directly reached by negating the constraints from `b1` and `b2`.

Coverage: Concolic execution is good at solving complex branch conditions. Hence if there are a lot of previously unsolved branches the concolic executor may encounter when executing on the given input, it will significantly improve the code coverage. The most common situations where concolic execution can help is when a conditional statement (*i.e.,*`if-then-else` or `switch-case`) exists. As the given input will only follow one of the branches, we call those branches stemmed from the same conditional statement *neighbor branches*. So we extract the following feature to estimate each seed's potential of new coverage.

1. *Count of undiscovered neighbor branches*: For all branches along the path triggered by the given seed,
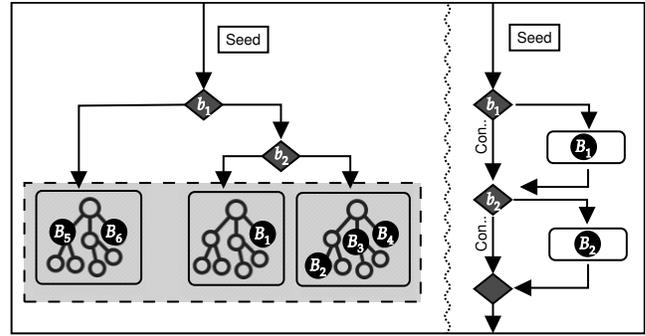

Figure 4: The examples that show how bug-triggering and coverage features are computed.

we compare their neighbors, if any, with all previously triggered branches. We then sum up the previously undiscovered neighbors for each branch. For instance, the value of this feature in the right example of Figure 4 is two if the seed follows the path with continue labels.

Constraint Solving: We also devised a set of features that impact the solving capabilities of the concolic execution engine. The incentive behind selecting such features is that the performance of the concolic executor significantly influences the entire hybrid fuzzing system.

1. *Count of external calls*: Existing concolic executors either rely on a simulated procedure or simply terminate the path execution when encountering an external function. As a result, external function calls may have negative impacts on the concolic executor, such as misleading the path and causing failure to generate correct seeds. This feature records the count of external function calls along the path executed by the given seed.

2. *Count of comparison instructions*: This feature records the count of `cmp` instructions along the path executed by the given seed. Comparison instructions pose the constraints on the execution path, which will later be solved by the SMT solver. However, constraint-solving is very time-consuming and is often the reason for the timeout.

3. *Count of indirect calls*: This is the number of indirect call instructions along the path executed by the given seed. Indirect calls may cause state explosion because when the concolic executor encounters an indirect call with a symbolic pointer, it simply forks a state for each possible value that can be resolved for the symbolic pointer [44]. In large programs, there could be many possible values for a symbolic function pointer.

4. *Length of path*: This feature records the number of executed branches (not deduplicated) by the given input. It helps identify the existence of large loops, which is another common reason that causes state explosion and solver timeout.

Empirical: This set of features is devised based on the empirical observations by existing works. They might

indirectly affect fuzzing performance.

1. *Input size*: Size of the input is often employed by existing tools as a heuristic to make a scheduling decision. On the one hand, smaller size inputs often end the execution more quickly and then leave more time for the fuzzer or concolic executor to explore other inputs [13, 53]. On the other hand, larger input has a better chance to trigger more functionalities [25]. Therefore, we consider the input size as one of the potential features for our approach.

2. *First seed with new coverage*: This is a boolean value indicating whether the given seed is the first one to discover some new branches or not. This is based on the intuition that such seeds are more likely to trigger more new coverage. This feature is used in many popular fuzzers [8, 33].

3. *Queue size*: This feature records how many inputs are saved in the fuzzing queue at the time of the query. If the queue is long, it is less likely to see more new coverage. Since MEUZZ needs to predict the utility of each seed during runtime, namely how much *more* new coverage can be discovered by fuzzing with the given input, the prediction should consider the current status of fuzzing.

4.4   Seed Label Inference

Labeling is an indispensable stage of data preprocessing in supervised learning. Well-defined labels make the prediction much easier and more reliable. As we aim to predict the utility of a selected seed and there is no direct indication to show if the selected seed is definitely useful, we need to derive a label by which we show the proportion of the seed utility.

To understand the utility of a seed, we need to fuzz the program with that seed and check the outcome. Fuzzers that use genetic algorithms (GAs) for seed generation represent such an outcome as a forest of *input descendant tree*, which depicts the parent-child relationship of the seeds in the fuzzer's queue. Each node of the tree represents a seed, and each edge connects a seed to one of its mutants.

In plain fuzzing, the root nodes are the original seeds provided by the user. Similarly, in hybrid testing, we model the inputs that are selected to be executed concolicly as the root nodes. When an input is selected to explore, the concolic engine will produce mutants of the running input. These mutants can further cover the neighbor branches (§ 4.3) of the re-visited path. After these mutants get transferred back to the fuzzer's queue, the fuzzer can use GA to further mutate them. As a result, we can draw the parent-child edges from the selected input to the mutants generated by the concolic engine, and to their GA-derived offsprings to form a mega descendant tree.

If the descendant tree of a seed is larger, it comparatively means the seed contributes more to the fuzzer's code coverage. Hence, to derive the label, we measure the size of the input descendant tree of a seed and consider it as the label.

In reality, it is not feasible to compute the complete descendant tree since it could grow indefinitely if the user never terminates the fuzzing process. As a result, we have to limit the tree analysis to a time window to make the label inference possible. Specifically, after the fuzzer imports a seed from concolic executor, we wait for a certain number of fuzzing epochs for the fuzzer to explore the imported seed and then compute the size of its descendant tree.

4.5   Model Construction and Prediction

The next step after preparing the data is to predict the seed prominence (*i.e.,* label). As the seed labels are the number of nodes in the *seed descendant tree*, their values are continuous so we need a regression model to predict them. Hence, we embed a regression model in MEUZZ in a way that when new seeds are generated by the fuzzer, the model predicts the utility of the seeds and then transfer the potential seeds to the concolic engine.

MEUZZ predicts very naively or just random at the beginning of fuzzing because the model just sees a few samples. However, the prediction becomes more reliable when more seeds are generated–data plays a crucial role in advancing model–and the model receives updates.

As seeds are mutated continuously during fuzzing a program in real-time, prediction and model update need to be done in a limited time window. Such limitation makes online learning approaches desirable candidates for model construction. In online-learning, the model can be incrementally updated by only considering new data. It does not need to store all previous data and to learn a model from scratch in every iteration. Instead, the model can be updated incrementally based on the incoming input, previous model and historical fuzzing yields. Such an update is very fast and requires less storage, which fits our use case very well. Thus we adopt online learning as one of the techniques for model construction.

4.6   Updating Model

To assure the model is entirely up-to-date with the prevailing seeds, ideally, we need to dynamically update/retrain the model, depending on the learning type (*i.e.,* online vs. offline). By doing so, we can both predict and learn in real-time.

For online learning, we use the Recursive Least Square (RLS) algorithm [21, 46] to update our linear model. Suppose at time t, the input data and the label are $\mathbf{x}_t$ and y_t correspondingly, where $\mathbf{x}_t$ is a vector of d-dimension. The following formula shows how the weight of the model at time t (*i.e.,* $\mathbf{w}_t$) is updated based on the weight obtained from the previous model (*i.e.,* $\mathbf{w}_{t-1}$):

$$\mathbf{w}_t = \mathbf{w}_{t-1} + \mathbf{C}_t^{-1} \mathbf{x}_t \left[y_t - \mathbf{x}_t^T \mathbf{w}_{t-1} \right]$$

where $\mathbf{C}_t^{-1}$ is the inverse of for $\mathbf{C}_t$, and $\mathbf{C}_t$ is defined as:

$$\mathbf{C}_t = \sum_{i=1}^{t} \mathbf{x}_i \mathbf{x}_i + \lambda \mathbf{I}$$

Note that to calculate $\mathbf{C}_t^{-1}$, we do not need to store all previous data and compute the inverse. Based on the Woodbury formula, $\mathbf{C}_t^{-1}$ can also be updated recursively as follows:

$$\mathbf{C}_t^{-1} = \mathbf{C}_{t-1}^{-1} - \frac{\mathbf{C}_{t-1}^{-1} \mathbf{x}_t \mathbf{x}_t^T \mathbf{C}_{t-1}^{-1}}{1 + \mathbf{x}_t^T \mathbf{C}_{t-1}^{-1} \mathbf{x}_t}$$

The complexity for such an update is $O(d^2)$.

To update the offline learning algorithms, the model needs to be retrained with all historical data in every iteration. Although retraining the model with the whole dataset every time a new seed is coming seems to be time-consuming, we show in our evaluation the approach is still practical in our case (§6.3). One reason is that the seed attributes are not of very high dimension and the number of seeds that need to be retrained is within an acceptable order of magnitude.

5 Implementation

Among the three components of MEUZZ, two of them are based on off-the-shelf software. We employ AFL-2.52b [33] for the fuzzing module and the re-engineered variant of KLEE from SAVIOR [25] for concolic execution. We develop the coordinator component from scratch in Python in 3,152 SLOC. Below, we detail the implementation of the major components of the ML engine, namely feature extraction and label inference.

Feature extraction: As discussed in §4.2, considering the trade-off between computational complexity and accuracy is key in feature extraction. Hence, for developing complicated features, we use a combination of static and dynamic analyses to offload the heavy tasks to compile time as much as possible. For instance, to extract the bug triggering features, we first instrument the target program with UBSan [15] at compile time. Then, a reachability analysis based on SVF [50] is used to extract the number of sanitizer instrumentations that can be reached from each branch. During runtime, we simply collect all the triggered branches by replaying the input and add up the number of reachable instrumentations from these branches.

To extract the feature of undiscovered neighbors, we record the branches and their neighbors at compile time. This information is later used to query whether any neighbor of a triggered branch is covered. To facilitate fast queries, we store the neighbor list as a disjoint-set data structure and use the union-find algorithm to query during runtime.

We extract the rest of the features either via compile-time instrumentation (*e.g.,* cmp, call instructions) and runtime input replay or via operating system APIs (*i.e.,* size, queue size, and new coverage).

Label inference: To collect the size of *seed descendant tree*, we traverse AFL's fuzzing queue. Thanks to the seed naming system of AFL (*i.e.,* [id, source, mutation, new cov]), we can iteratively traverse the seeds and use transitive closure to collect all the inputs imported from the concolic executor and their descendant trees.

Table 1: Evaluation settings

Program			Settings	
Name	Version	Driver	Initial Seeds	Options
tcpdump	4.10.0	tcpdump	[14]	-r @@
binutils	2.32	objdump	[5]	-D @@
binutils	2.32	readelf	[5]	-A @@
libxml	2.9.9	xmllint	[11]	stdin
libtiff	4.0.10	tiff2pdf	[10]	@@
libtiff	4.0.10	tiff2ps	[10]	@@
jasper	2.0.16	jasper	[9]	-f @@ -T pnm
libjpeg	jpeg9c	djpeg	[9]	stdin

6 Evaluation and Analysis

We conduct a comprehensive set of experiments to answer the following research questions:

- *RQ1*: Can ML-based seed scheduling outperform heuristics-based approaches (§ 6.2 and § 6.6)?
- *RQ2*: Which features are more important in predicting seed utility and which learning mode is more effective (§ 6.3)?
- *RQ3*: Does the learned model adapt well to different fuzzing configurations (§ 6.4)?
- *RQ4*: Is it feasible to transfer the learned model from a program to other programs to improve fuzzing yields (§ 6.5)?

6.1 Evaluation setup

Following the general fuzzing evaluation guideline [32], we choose 8 real-world benchmark programs commonly used by existing work [19, 24, 25, 54, 55]. Table 1 shows the configurations used for fuzzing each program. All experiments are conducted on AWS c5.18xlarge servers running Ubuntu 16.04 with 72 cores and 281 GB RAM. Without explicitly mention, all tests run for 24 hours each by assigning three CPU cores to each fuzzer and are repeated at least 5 times; we report the average result with Mann-Whitney U-test.

We compare MEUZZ with the state-of-the-art grey-box fuzzers, such as AFL [33], AFLFast [19], and Angora [24], as well as hybrid testing systems including QSYM [54] and SAVIOR [25]. The seed selection modules of all these previous systems are based on heuristics. We could not test Driller [49] on the chosen benchmarks because its concolic execution engine fails to run them. Moreover, we test Vuzzer [41] and T-Fuzz [39] but we compare them with MEUZZ in a different way than we do with the other fuzzers. This separate comparison is because these two fuzzers do not support concurrent fuzzing. Due to the space limit, we discuss our observations and show the results of their branch coverage in Appendix C.

For MEUZZ, we consider three different configurations according to the learning process, namely MEUZZ-OL, MEUZZ-RF and MEUZZ-EN, which refer to online learning linear model, offline learning random forest model and the arithmetic average of the first two models' utility predictions, respectively.

Since Savior and QSYM need at least three CPU cores, we enforce this fuzzing setting to all the fuzzers to build a

fair comparison environment. We launch one master and two slaves for the grey-box fuzzers; and one master, one slave, and one concolic execution engine for the hybrid fuzzers. To reduce the randomness of OS scheduling, we pin each component of the fuzzers on the specific core. Because MEUZZ and SAVIOR instrument the testing program with UBSAN [15], we also apply this sanitizer to all other fuzzers, as enabling sanitizers is shown to improve the fuzzer's effectiveness for finding bugs.
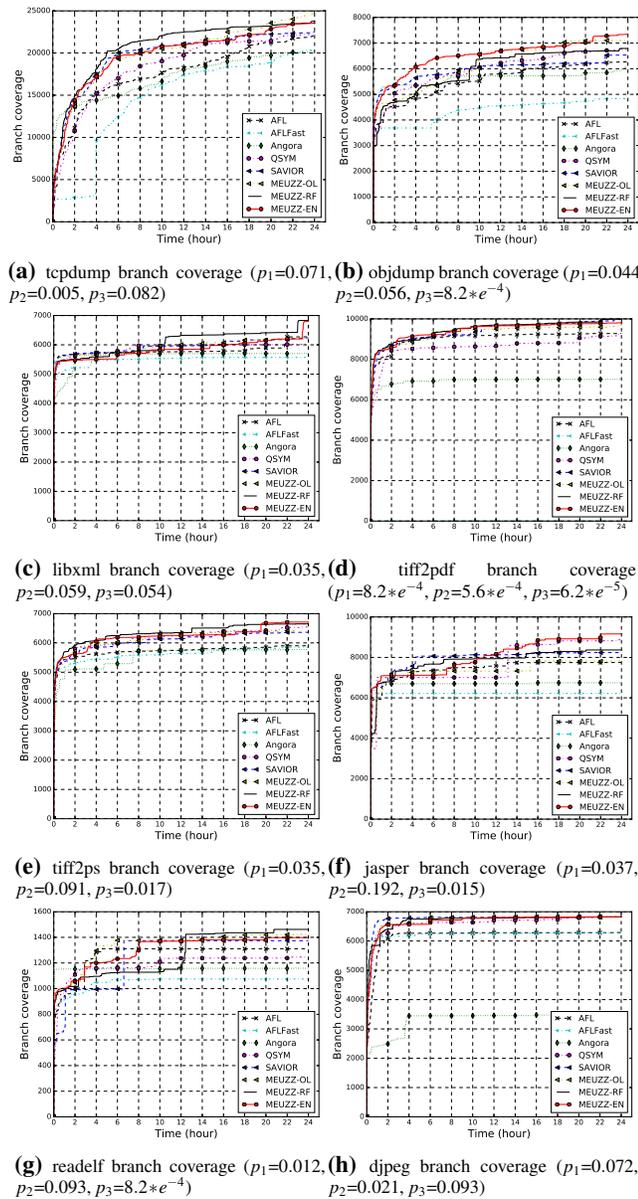
6.2 Learning Effectiveness

(a) tcpdump branch coverage (p_1=0.071, p_2=0.005, p_3=0.082)

(b) objdump branch coverage (p_1=0.044, p_2=0.056, p_3=8.2$*e^{-4}$)

(c) libxml branch coverage (p_1=0.035, p_2=0.059, p_3=0.054)

(d) tiff2pdf branch coverage (p_1=8.2$*e^{-4}$, p_2=5.6$*e^{-4}$, p_3=6.2$*e^{-5}$)

(e) tiff2ps branch coverage (p_1=0.035, p_2=0.091, p_3=0.017)

(f) jasper branch coverage (p_1=0.037, p_2=0.192, p_3=0.015)

(g) readelf branch coverage (p_1=0.012, p_2=0.093, p_3=8.2$*e^{-4}$)

(h) djpeg branch coverage (p_1=0.072, p_2=0.021, p_3=0.093)

Figure 5: Branch coverage fuzzing with valid seeds (higher is better). p_1, p_2 and p_3 are p-values in Mann-Whitney U Test by comparing QSYM with MEUZZ-OL, MEUZZ-RF and MEUZZ-EN, respectively.

Table 2: Execution time spend on different learning stages

Model Update (s)		Prediction(s)		Feature Extraction (s)
Online	Offline	Online	Offline	
0.000636	0.326139	0.000016	0.003168	5$e-6$

The most straightforward metric to measure the effectiveness of MEUZZ is code coverage, which is also a widely accepted and evaluated metric. Figure 5 shows the branch coverage achieved by different fuzzers to the required time for fuzzing. Based on the coverage result, we have several interesting findings.

First, MEUZZ covers more code than other fuzzers in most programs after 24 hours of fuzzing. Among the *non-ML* fuzzers, QSYM performs the best in terms of code coverage, thanks to its efficient concolic execution engine tailored specially for hybrid fuzzing. Compared with QSYM, the MEUZZ variants achieve various levels of coverage improvements. In tcpdump, objdump, readelf and libxml, MEUZZ improves code coverage over QSYM by more than 10%, and particularly 27.1% by MEUZZ-RF in readelf. In tiff2pdf and tiff2ps, MEUZZ also has moderate coverage improvements. However, in jasper and djpeg, there is no much difference between MEUZZ and QSYM; we speculate it is because all fuzzers are saturated and hit a plateau after 6 hours.

Second, MEUZZ covers less code in the beginning but gradually surpasses other fuzzers as time progresses. For example, in objdump MEUZZ-OL and MEUZZ-RF did not cross QSYM and SAVIOR until after 9.6 hours of fuzzing, but MEUZZ eventually achieves 14% higher code coverage. Similar situations can be observed in libxml, readelf and tiff2ps. This observation is expected, as MEUZZ starts seed scheduling with random parameters, hence the performance of seed selection is unpredictable at the beginning. But as time passes, fuzzing data are increasingly collected and used to refine the prediction model. Hence, the prediction becomes more accurate.

Lastly, the effectiveness of ML is presented in Figure 10 in Appendix D. It is shown that different programs are variously affected by different sets of features. For instance, *External Calls* has more influence on six of the programs except for tcpdump and djpeg, showing that no single feature is sufficient to predict high-utility seeds. By using a data-driven approach, we cannot only *automatically* select the high impactful features in different programs or situations, but also integrate them in a more optimal way than manual-crafting rules.

6.3 Insights and Analyses

Online v.s Offline learning: As mentioned in the previous section, offline learning with the random forest model sometimes beats online learning with the linear model; however, the main concern with using offline learning is time delays, especially during the model updating stage.

To further analyze the effects of time delays caused by offline learning, we profile each learning stage during the 24
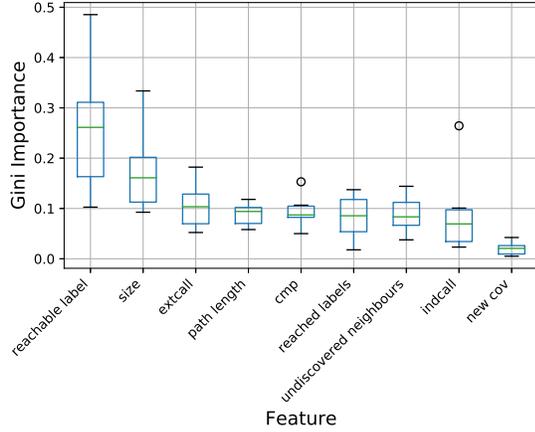
Figure 6: The box plots show the importance of the features on nine programs. The importance is extracted by training an offline random forest model and they are ranked by the median of their importance. *Reachable label* and *New Cov* are the most and the least important ones, respectively.

hours of fuzzing and report the average time spend on different learning steps. As shown in Table 2, although offline learning spent 512x and 198x more time than online learning on updating the model and making predictions, respectively, the absolute time-lapse is negligible (*i.e.,* in milliseconds). Hence, offline learning is not a critical hindrance throughout the hybrid fuzzing loop, which endorses the offline learning effectiveness discussed in Section 6.2. Having said that, if fuzzing continues for a longer time and the number of seeds significantly increases, offline learning can become an obstacle.

Feature Analysis: Figure 6 presents the distribution of the importance of each feature separately in all programs. The importance score is computed by capturing the *mean decrease impurity* from the offline random forest models [22]. The figure shows the contribution of the *New Cov* feature is the least among all the features. While it is difficult to entirely disregard the minor contribution of *New Cov*, this suggests that putting much effort to follow the seeds that bring new coverage might jeopardize the chance to explore unknown seeds. This is also known as the famous Multi-Armed Bandit (MAB) problem [18]. This finding might shed some light on the scheduling algorithm implemented in the popular fuzzers like AFL [33] that heavily rely on the *New Cov* heuristics.

Also, the variance of change in the figure shows some of the features like *Path Length* and *New Cov* are less subject to programs, while others like *Reachable Label* are more tied to programs. If the extraction of a feature heavily depends on static analysis, it is less precise compared with dynamic analysis because the sensitivity of static analysis affects the precision (*i.e.,* flow/context/field sensitivity). We speculate this is one of the reasons that make a feature (*e.g., Reachable Label*) more dependent on individual programs. Also, there are additional factors that might affect dependability. Program loops as a

common trait in all programs uniformly affect the *Path Length* feature, which makes the feature more agnostic to programs. Similarly, *New Cov* is set to a seed during runtime when it is the first one to trigger new behaviors (*e.g.,* coverage); this attribute is generally applicable to a variety set of programs.

It is worth noting that the average time to extract each feature is only $5\mu s$ (as shown in Table 2), thanks to our light-weight feature extractions. This indicates that the online-friendly requirement is satisfied in MEUZZ.
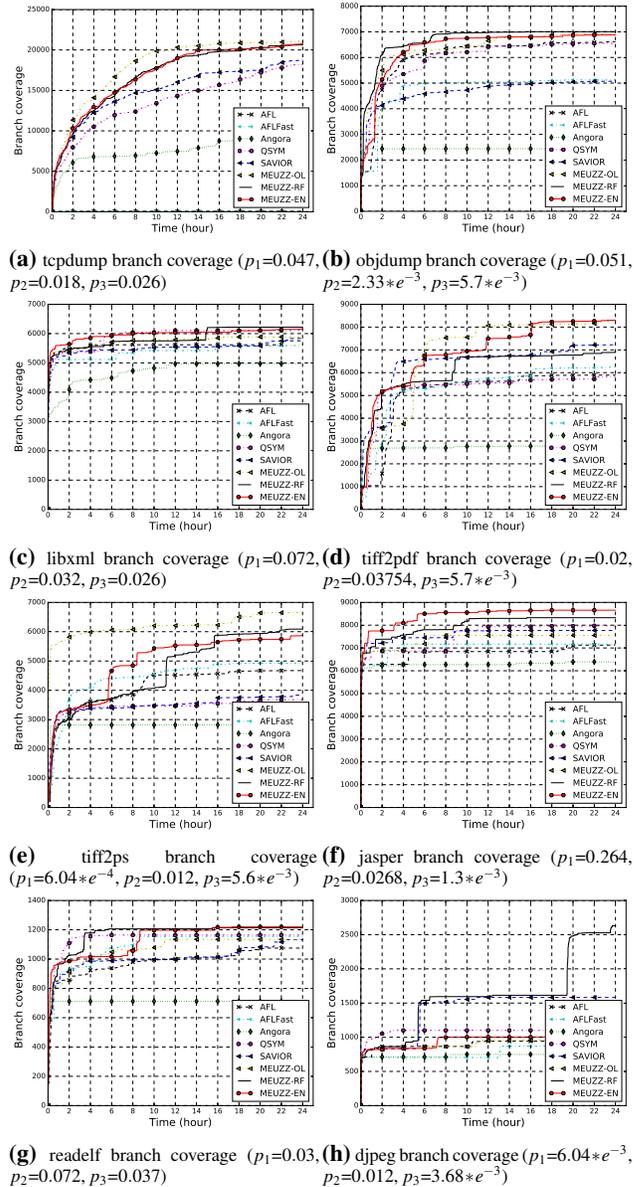
6.4 Model Reusability

(a) tcpdump branch coverage (p_1=0.047, **(b)** objdump branch coverage (p_1=0.051, p_2=0.018, p_3=0.026) p_2=2.33$*e^{-3}$, p_3=5.7$*e^{-3}$)

(c) libxml branch coverage (p_1=0.072, **(d)** tiff2pdf branch coverage (p_1=0.02, p_2=0.032, p_3=0.026) p_2=0.03754, p_3=5.7$*e^{-3}$)

(e) tiff2ps branch coverage **(f)** jasper branch coverage (p_1=0.264, (p_1=6.04$*e^{-4}$, p_2=0.012, p_3=5.6$*e^{-3}$) p_2=0.0268, p_3=1.3$*e^{-3}$)

(g) readelf branch coverage (p_1=0.03, **(h)** djpeg branch coverage (p_1=6.04$*e^{-3}$, p_2=0.072, p_3=0.037) p_2=0.012, p_3=3.68$*e^{-3}$)

Figure 7: Branch coverage fuzzing with naive seeds (higher is better). p_1, p_2 and p_3 are p-values in Mann-Whitney U Test by comparing QSYM with MEUZZ-OL, MEUZZ-RF and MEUZZ-EN, respectively.

Building machine learning models is a valuable but time-consuming task. It is reasonable to build and reuse models where possible. By reusing a model, one can improve generalization, speed up training, as well as improving the model accuracy. Also, reusability can be good evidence that our model correctly captured what kind of inputs have higher utility when testing the target programs. Hence, we test the reusability of the learned models obtained via the previous fuzzing experiments.

We conduct an experiment in which we use a pre-trained model for fuzzing the same target program and compare the coverage difference. We make the following two changes in the experiment performed in § 6.2: (*i*) the initial seeds are replaced by a naive input that only consists 4 whitespaces; and (*ii*) all MEUZZ variants are initialized with the models they learned in the effectiveness test (with valid initial seeds).

Figure 7 shows the coverage result with Mann-Whitney U Test. There are several interesting observations. The most important one is that the MEUZZ variants start performing well even at the beginning of fuzzing compared with when there is no model initialization. We believe this improvement is brought by the initial models. Additionally, "pure-AFL" fuzzers do not perform well with this naive initial seed. For instance, in `tcpdump`, AFL and AFLFast only generate 6 inputs in total after 24 hours of fuzzing (see Figure 7a). On the contrary, systems augmented with other input generation techniques such as concolic execution and taint analysis can generate more inputs and consequently can explore significantly more code. Lastly, MEUZZ-RF outperforms its peers in `djpeg`, and its p-value indicates the improvement is significant (< 0.05), suggesting the non-linear model works better on `djpeg`.

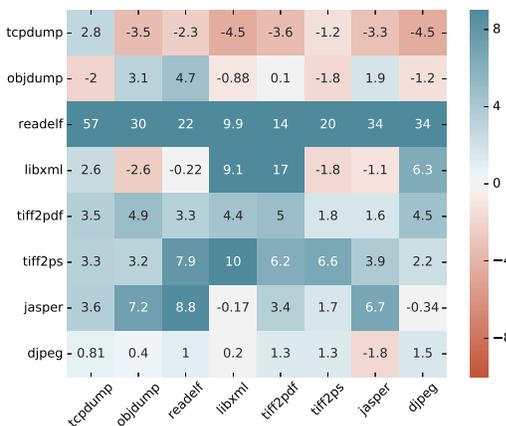### 6.5 Model Transferability

Figure 8: This heat map shows Coverage improvement with model initialization for MEUZZ-OL over vanilla MEUZZ-OL. Y-axis is the tested programs, X-axis is the models used for initialization. Each cell shows the relative coverage comparison (%). The diagonal values show the coverage improvement on each program after initializing MEUZZ with model learn from the same program (reusability). Model transferability is shown in 7 out of the 8 programs.

To further evaluate the model reusability explained in the previous section, we conduct a cross-program experiment to determine whether a model trained on one type of program will transfer well to fuzzing a new program. This is known as transfer learning in the ML field [38]. As far as we know, no prior research has attempted to show this invaluable analysis in fuzzing [42].

In this experiment, we augment MEUZZ with a pre-trained model from one program and compare the result of the fuzzer on different programs with a baseline. Our baseline is the coverage result from the learning effectiveness experiment (§ 6.2), in which we use valid seeds to bootstrap fuzzing without model initialization. We choose MEUZZ-OL as the representative of our system to measure this transferability experiment. We then fuzz each program using MEUZZ-OL initialized with the 8 pre-learned models; the models are fixed afterward.

Figure 8 visualizes the comparative coverage improvements (*i.e.,* percentage) produced by each fuzzing configuration. The Y-axis shows the tested program and the X-axis shows the programs by which the models are built. This result shows three interesting findings.

First, MEUZZ-OL observes 7.1% more code coverage on average when it is tested on the same program it is initialized with. The amount of improvement for each program is shown in the diagonal of Figure 8, from top left to bottom right. Note that these models are only learned in 24 hours from previous experiments; we expect to see more improvement in continuous fuzzing services (*e.g.,* [12]). This again confirms that the previously learned models are reusable.

Second, MEUZZ-OL observes improvement in 38 out of 56 cross-testing cases, which shows 67.9% success rate when the model is transferred from a program to another program. Among them, 10 cases see more than 10% coverage improvement. Such improvement also indicates that the program-agnostic requirement is satisfied in MEUZZ.

Last but not least, we notice different programs have different "sensitivity" towards the transferred models. For instance, almost all the transferred models can strengthen fuzzing `readelf`, `tiff2pdf`, `tiff2ps` and `djpeg` programs, among which `readelf` sees the highest improvement. Interestingly, `readelf` achieves even higher improvement when using the `tcpdump` model than the `readelf` model by itself. However, other programs are only partially accepting foreign models. For instance, the model of `tcpdump` can outperform almost all of the programs, while none of the other seven external models can improve its fuzzing yields.

Two main reasons can justify the aforementioned observation, namely the number of data points as well as feature importance. When there is more data, the model can better generalize [31]. For instance, the `tcpdump` model contains a higher number of seeds compared with others (see Figure 1), which justifies the effectiveness of the transferred model built from the `tcpdump` program. We also compared the importance of the features of each program (see Appendix D). The shape

Table 3: The table shows the unique bugs found by all evaluated fuzzers.

Program	AFL	AFLFast	Angora	QSYM	Savior	MEUZZ	All unique bugs
tcpdump	14	13	12	11	12	14	14
objdump	2	2	5	2	8	6	9
readelf	3	2	5	5	4	4	6
tiff2pdf	1	1	1	2	2	2	2
tiff2ps	1	2	2	5	4	6	6
jasper	2	1	0	3	1	6	8
djpeg	9	7	7	9	9	9	9
Total	32	28	32	37	40	47	54

of the final importance chart in `tcpdump` diverges more from the rest of the programs. Moreover, the values of some features such as *Indirect Call* and *Path Length* are higher than other programs. By looking at these statistics as well as checking the source code of `tcpdump` we noticed `tcpdump` is designed with heavier use of function handlers for different types of network packets and recursive loops for parsing packet fields. While other models contain different feature value distribution as well as fewer data points, which justify the failure of using them to improve fuzzing `tcpdump`.

6.6 Discovered Bugs

To prove the effectiveness of our system in discovering new bugs, we performed various analyses. We manually analyzed all of the reported undefined behaviors and crashes. UBSan reports a large amount of undefined behaviors; however, the majority of them are deemed benign after our triage process. We also triage additional bugs with the help of ASAN [1] and LeakSAN [7].

Table 3 shows our triage result for all the fuzzers. In total, 54 unique bugs were uncovered. MEUZZ outperforms other fuzzers and found 47 unique bugs, which supports the fact that higher code coverage correlates to a higher number of triaged bugs. Due to space limit, we present more detailed triage result and one of the discovered bugs only found by MEUZZ in Appendix B. This result shows MEUZZ is more effective in terms of finding bugs than state-of-the-art systems with manually crafting heuristics.

7 Related Work

7.1 ML for Fuzzing

Despite the promising potential to improve fuzzing, the application of ML has not been very-well investigated in the past and only a few research have leveraged ML. ML can be integrated into various stages of fuzzing, from input generation to crash categorization.

Input generation: The most intelligent stage of fuzzing has been the input generation stage so far, thanks to genetic algorithms. Deep learning (DL) techniques have been recently applied to input generation for both mutation/generation-based fuzzing. Such approaches [30, 40, 47] use various neural network methods to learn the patterns that exist in input files and then identify the likely input forms to trigger new coverage. Similarly, reinforcement learning (RL) [20] can learn input grammar for generation-based fuzzers.

Crash analysis: Automating the analysis of outputs/crashes generated by fuzzers is another ML application. For instance, ML can be used to categorize crashes by identifying the root cause of them. This helps remove duplicate outputs and therefore reduces manual analysis effort [26]. Or another example is employing ML to predict whether the reported crashes by fuzzers are exploitable [52].

To the best of our knowledge, there has not been any research that practices ML for seed selection. In general, the practicality of ML for fuzzing has not been shown clearly in the past due to the uncertainty about reusability and transferability.

7.2 Seed Scheduling Heuristics

Scheduling in fuzzing: FuzzSim [51] models the seed scheduling problem as a weighted coupon collector problem and found out that scheduling can have a direct impact on fuzzing campaign yields. Later, in grey-box fuzzing, AFL [33] implements a scheduling algorithm that consists of simple heuristics such as preferring first seed with new coverage, and with smaller size and less execution time. This simple algorithm is later improved by Fairfuzz [34] and AFLFast [19] which steer the fuzzer towards less explored paths.

Scheduling in hybrid testing: As hybrid testing becomes more popular, seed scheduling also becomes a research topic. Driller [49] implements a random scheduling algorithm, while QSYM [13] implements heuristics similar to AFL. Later, DigFuzz [55] shows the ineffectiveness of random scheduling and proposes a Monte-Carlo model to predict the difficulty of each path explored by the fuzzer by far, and send the most difficult ones to concolic executor. SAVIOR [25], on the other hand, uses bug-driven scheduling heuristics. By selecting the seeds that can reach more sanitizer instrumentations, it triggers more bugs in the given timeframe than other fuzzers.

Compared with these approaches, MEUZZ applies machine learning techniques that can learn a utility prediction model, which is adaptive to the program being tested. As our evaluation suggests, this approach is more scalable and more performant than the manual-crafting scheduling heuristics.

8 Conclusion

We present MEUZZ, a hybrid fuzzing system featuring machine learning and data-driven seed scheduling. Theoretically, MEUZZ is more generalized than systems using fixed seed selection heuristics. For effective integration of machine learning workloads into the online hybrid fuzzing loop, MEUZZ follows the requirements of being utility relevant, online friendly and program agnostic for its feature engineering and label inference. Our evaluation shows that MEUZZ outperforms state-of-the-art fuzzers in both code coverage and bug discovery. In addition, the learned models demonstrate good reusability and transferability, making it more practical to apply machine learning to hybrid fuzzing.

Acknowledgment

References

[1] Addresssanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[2] Afl technical details. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[3] angr/tracer: Utilities for generating dynamic traces. https://github.com/angr/tracer.

[4] Announcing oss-fuzz: Continuous fuzzing for open source software. https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[5] Binutils test cases. https://github.com/mirrorer/afl/tree/master/testcases/others/elf.

[6] Darpa cyber grand challenge. http://archive.darpa.mil/cybergrandchallenge/.

[7] Leaksanitizer. https://clang.llvm.org/docs/LeakSanitizer.html.

[8] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[9] Libjpeg test cases. https://github.com/mirrorer/afl/tree/master/testcases/images/jpeg.

[10] Libtiff test cases. https://github.com/mirrorer/afl/tree/master/testcases/images/tiff.

[11] Libxml test cases. https://github.com/mirrorer/afl/tree/master/testcases/others/xml.

[12] Oss-fuzz - continuous fuzzing for open source software. https://github.com/google/oss-fuzz.

[13] Qsym: A practical concolic execution engine tailored for hybrid fuzzing. https://github.com/sslab-gatech/qsym.

[14] Tcpdump test cases. https://github.com/the-tcpdump-group/tcpdump/tree/master/tests.

[15] Undefined behavior sanitizer - clang 9 documentation. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks.

[16] When does deep learning work better than svms or random forests? https://www.kdnuggets.com/2016/04/deep-learning-vs-svm-random-forest.html, 04 2016.

[17] Cyber grand shellphish. http://www.phrack.org/papers/cyber_grand_shellphish.html, 2017.

[18] Donald A Berry and Bert Fristedt. Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability). *London: Chapman and Hall*, 5:71–87, 1985.

[19] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.

[20] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. *CoRR*, abs/1801.04589, 2018.

[21] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142.

[22] L Breiman, JH Friedman, R Olshen, and CJ Stone. Classification and regression trees. 1984.

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

[24] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[25] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 2–2, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[26] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1084–1093, June 2012.

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

[28] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.*, 15(1):3133–3181, January 2014.

[29] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[30] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

[31] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, March 2009.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[33] lcamtuf. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2015.

[34] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *CoRR*, abs/1709.07101, 2017.

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[36] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[37] M. Nayrolles and A. Hamou-Lhadj. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 153–164, May 2018.

[38] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, Oct 2010.

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

[40] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR*, abs/1711.04596, 2017.

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[42] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[43] Adam Santoro, David Raposo, David G. T. Barrett, Mateusz Malinowski, Razvan Pascanu, Peter W. Battaglia, and Timothy P. Lillicrap. A simple neural network module for relational reasoning. *CoRR*, abs/1706.01427, 2017.

[44] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, pages 28–28. USENIX Association, 2012.

[46] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[47] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*. IEEE, 2018.

[48] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 46–55. IEEE, 2015.

[49] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[50] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[51] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.

[52] G. Yan, J. Lu, Z. Shu, and Y. Kucuk. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *2017 IEEE Symposium on Privacy-Aware Computing (PAC)*, pages 164–175, Aug 2017.

[53] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*. IEEE.

[54] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 745–761. USENIX Association, 2018.

[55] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

Appendix A   Why use UBSAN

Note that although the design of MEUZZ is generically compatible with mainstream sanitizers [15, 45, 48], our implementation uses UBSan for the following reasons: (*i*) UBSan instruments programs with pure static checks that can be easily converted to solvable SMT constraints. In contrast, other sanitizers, such as ASAN and MSAN, employ red-zones and status bitmap, which are less amenable to constraint solving. (*ii*) Our concolic engine is based on SAVIOR's KLEE, which uses UBSan as the primary sanitizer. Using UBSan makes concolic execution more effective as shown in [25].

Appendix B   Bugs found by MEUZZ

We provide a more detailed triage information of the bugs found by MEUZZ. In total, MEUZZ found 30 undefined behaviors, among which 21 have been confirmed/fixed so far by the developers and the rest are pending. For the reported bugs, we found the potential UBs with UBSAN [15] and manual analysis; we found the memory errors and DoS with ASAN [45] and memory leaks with LeakSAN [7].

Table 4: The table shows the discovered bugs by MEUZZ. UB, ME, DoS, and ML refers to Undefined Behavior, Memory Error, Denial of Service, and Memory Leak, respectively.

Program	Potential UB	ME	DoS	ML	Confirmed
tcpdump	14				4
objdump	4			2	
readelf	2	1		1	1
tiff2pdf			2		2
tiff2ps	1	4	1		4
jasper			4	2	4
djpeg	9				6
Total	30	5	7	5	21

```
1  for (; cc < tf_bytesperrow; cc += samplesperpixel)
   ↪  {
2
3  adjust = 255 - cp[nc];
4  switch (nc) {
5  case 4: c = *cp++ + adjust; PUTHEX(c,fd);
6  case 3: c = *cp++ + adjust; PUTHEX(c,fd);
7  case 2: c = *cp++ + adjust; PUTHEX(c,fd);
8  case 1: c = *cp++ + adjust; PUTHEX(c,fd);
9  }
```

Figure 9: Off-by-one heap read overflow in tiff2ps.

One of the heap overflow vulnerabilities in `tiff2ps` is discovered only by MEUZZ. Figure 9 shows the vulnerable code snippet. This bug has been confirmed and fixed by the developers. It is an out-of-bound read vulnerability that can lead to information disclosure. The vulnerability takes place at `PSDataColorContig` function where `cp` buffer with the size of 4 bytes is allocated in heap and the 5th element of the buffer is accessed by `cp[4]` which leads to out-of-bound read. To trigger this bug, the loop needs to be executed without early breaks. Moreover, to control the buffer size, the input needs to satisfy many constraints in the `TIFFScanlineSize` function so that it will return value 4. Based on the feature importance of `tiff2ps` (Appendix D), *Size*, *Cmp* and *External Call* play more important roles in its model, we believe this is why MEUZZ is able to guide the fuzzer to explore and trigger this bug. On the contrary, by replaying the fuzzing corpora, we found that other fuzzers miss this bug because they either exit the loop early or fail the checks in `TIFFScanlineSize`.

Appendix C   Discussion on Extra Experiments

We attempted to compare MEUZZ with many state-of-the-art fuzzing systems but cannot conduct an apple-to-apple comparison with some of them due to various reasons.

Driller uses [3] as its concolic engine, which has limited support for system calls, causing the engine's failure to generate new test cases. Similar issue was also reported by Insu at el. [54]. Vuzzer and T-Fuzz do not have support for concurrent fuzzing. After discussing with the developers we assigned only one core to them and run them for 72 hours

(3×24) instead of 24 hours. We report the branch coverage results of Vuzzer and T-Fuzz in Table 5.

Table 5: The table shows the number of branches covered by Vuzzer and T-Fuzz. ✗ means fuzzer crashed on the program.

Program	Vuzzer	T-Fuzz
tcpdump	1103	11566
objdump	711	4216
readelf	1025	842
libxml2	715	✗
tiff2pdf	✗	4892
tiff2ps	✗	3534
jasper	✗	6084
djpeg	1317	763

Appendix D   Detailed Feature Importance Study

Figure 10 demonstrates how the randomly initialized model evolved with more and more training data available during fuzzing. MEUZZ automatically identifed which features are more important for each specific programs, showing it is more scalable than manually-written heuristics.
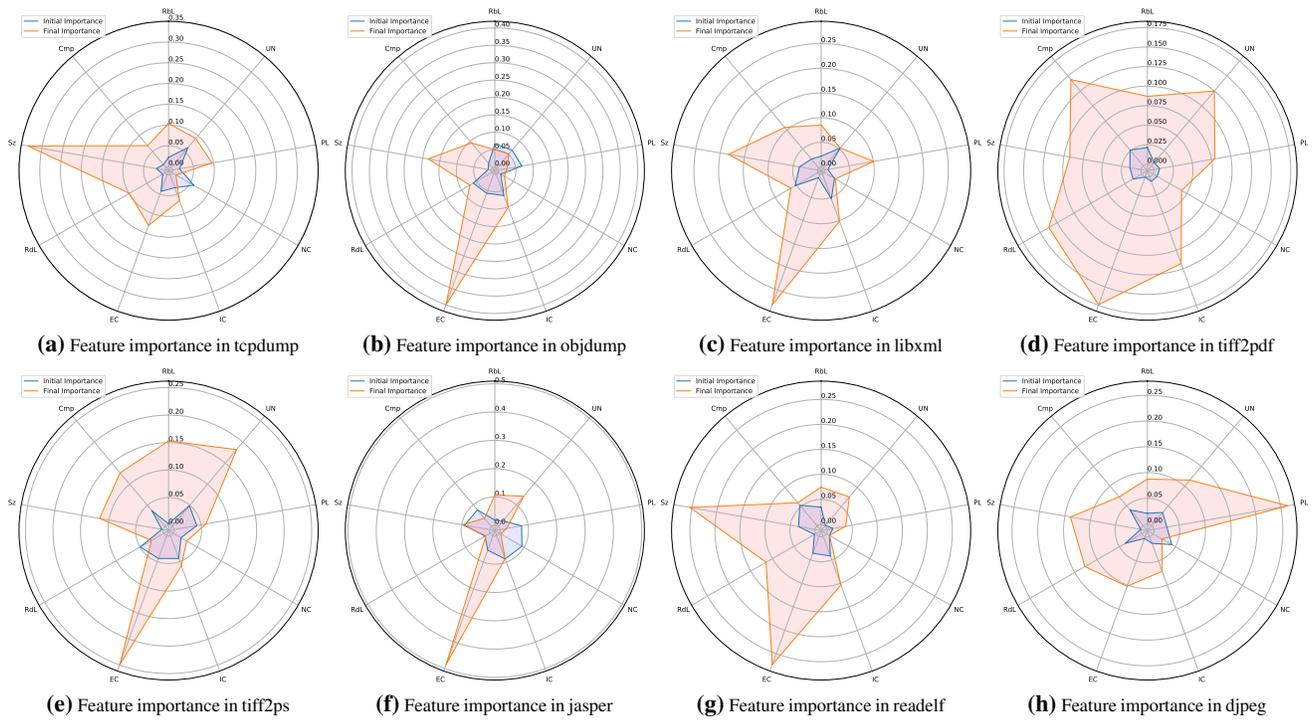
(a) Feature importance in tcpdump

(b) Feature importance in objdump

(c) Feature importance in libxml

(d) Feature importance in tiff2pdf

(e) Feature importance in tiff2ps

(f) Feature importance in jasper

(g) Feature importance in readelf

(h) Feature importance in djpeg

Figure 10: Feature importance extracted from models learned in the effectiveness test (§ 6.2). The initial importances are randomly generated. Sz: *Size*, RdL: *Reached Label*, EC: *External Call*, IC: *Indirect Call*, NC: *New Coverage*, PL: *Path Length*, UN: *Undiscovered Neighbors*, RbL: *Reachable Labels*, Cmp: *Comparisons*.