FOX: Coverage-guided Fuzzing as Online Stochastic Control

Dongdong She*
Hong Kong University
of Science and Technology
Hong Kong, China
dongdong@cse.ust.hk

Adam Storek*
Columbia University
New York, NY, USA
astorek@cs.columbia.edu

Yuchong Xie
Hong Kong University
of Science and Technology
Hong Kong, China
yu3h0xie@gmail.com

Seoyoung Kweon
Columbia University
New York, NY, USA
sk4865@columbia.edu

Prashast Srivastava
Columbia University
New York, NY, USA
ps3400@columbia.edu

Suman Jana
Columbia University
New York, NY, USA
suman@cs.columbia.edu

Abstract

Fuzzing is an effective technique for discovering software vulnerabilities by generating random test inputs and executing them against the target program. However, fuzzing large and complex programs remains challenging due to difficulties in uncovering deeply hidden vulnerabilities. This paper addresses the limitations of existing coverage-guided fuzzers, focusing on the scheduler and mutator components. Existing schedulers suffer from information sparsity and the inability to handle fine-grained feedback metrics. The mutators are agnostic of target program branches, leading to wasted computation and slower coverage exploration.

To overcome these issues, we propose an end-to-end online stochastic control formulation for coverage-guided fuzzing. Our approach incorporates a novel scheduler and custom mutator that can adapt to branch logic, maximizing aggregate edge coverage achieved over multiple stages. The scheduler utilizes fine-grained branch distance measures to identify frontier branches, where new coverage is likely to be achieved. The mutator leverages branch distance information to perform efficient and targeted seed mutations, leading to robust progress with minimal overhead.

We present FOX, a proof-of-concept implementation of our control-theoretic approach, and compare it to industry-standard coverage-guided fuzzers. 6 CPU-years of extensive evaluations on the FuzzBench dataset and complex real-world programs (a total of 38 test programs) demonstrate that FOX outperforms existing state-of-the-art fuzzers, achieving average coverage improvements up to 26.45% in real-world standalone programs and 6.59% in FuzzBench programs over the state-of-the-art AFL++. In addition, it uncovers 20 unique bugs in popular real-world applications, including eight that are previously unknown, showcasing real-world security impact.

*Equal contribution.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670362

CCS Concepts

• **Security and privacy → Systems security**; **Software security engineering**.

Keywords

fuzzing; coverage-guided fuzzing; online stochastic control; software testing

ACM Reference Format:
Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. FOX: Coverage-guided Fuzzing as Online Stochastic Control. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670362

1 Introduction

Fuzzing is a popular technique for discovering software vulnerabilities by generating random test inputs and executing them against the target program [13, 15, 24, 49]. While it has been successful in detecting security vulnerabilities in real-world programs [10, 11], fuzzing large and complex programs remains challenging due to difficulties in uncovering deeply hidden vulnerabilities.

This paper focuses on coverage-guided fuzzers, the prevailing approach to fuzzing, aiming to maximize edge coverage within a given time budget. These fuzzers maintain a list of seed inputs and select inputs for further mutation at each stage. They consist of two main components: a scheduler (choosing inputs for mutation) and mutators (modifying the chosen input). The goal is to generate inputs that explore new edges for better coverage. Most existing fuzzers use randomized mutations to adapt to different branches in the target program. The effectiveness of a fuzzer, therefore, depends on two factors: (i) the mutator's likelihood to generate new inputs achieving new coverage given a specific seed input, and (ii) the scheduler's ability to identify seeds that, when mutated, are likely to trigger new edges.

Limitations of Existing Approaches. We identify the following main drawbacks in the existing design of scheduler and mutator components for coverage-guided fuzzing. Firstly, the schedulers use coarse-grained feedback to select candidates for further mutation. They rely on seeds that have previously resulted in coverage gain when mutated. However, this approach suffers from serious sparsity of information, as coverage-increasing inputs become increasingly

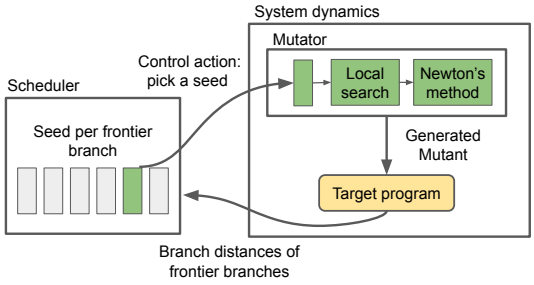

Figure 1: Workflow of FOX

rare as the fuzzing campaign progresses [45]. Consequently, the scheduler often degenerates into a round-robin approach. Attempting to use finer-grained data-flow-guided feedback metrics [31, 41] to address this issue can easily lead to an explosion in the seed corpus of the fuzzing scheduler [54]. Secondly, the current mutators are agnostic of the target program branches. They perform random mutations independently of the branch logic, with the hope of increasing coverage. As a result, these existing mutators waste computation while attempting to generate coverage-increasing mutations, leading to slower coverage exploration. Techniques using taint tracking to customize mutation operations for different branches tend to incur prohibitively high overhead [55]. Furthermore, despite the shared overarching goal of achieving new coverage, current fuzzers often treat the scheduler and mutator as separate entities with distinct objectives and little information exchange.

Our Approach. In this paper, we tackle these issues by presenting an end-to-end online stochastic control formulation for coverage-guided fuzzing, which encompasses both the scheduler and mutator components. In this framework, the stochastic mutator and target program represent the dynamics of the system, where the scheduler makes probabilistic online control decisions about which seed to mutate from the corpus, representing the fuzzer's state. Each scheduling step constitutes a stage of this control process, and our objective is to maximize the sum of expected coverage gain across multiple stages subject to a time budget constraint. To solve this problem, we introduce a novel scheduler and mutator that can efficiently adapt to branch logic, integrating them into a comprehensive control framework that can benefit from both the scheduler's multi-seed view and the mutator's seed- and branch-specific behavior. The workflow of FOX is shown in Figure 1.

Scheduler. To address the lack of meaningful information for the scheduler when mutators fail to achieve new coverage, we use fine-grained branch distance measures (see §2.3), indicating how close the current seed is to *flipping* a branch. Specifically, *flipping* indicates generating and executing a seed that can exercise alternative outgoing control-flow branches from the parent node of a previously seen branch. To avoid state space explosion while leveraging finer-grained feedback metrics, we apply the fine-grained feedback measure only to *frontier branches* — branches that have at least one unexplored outgoing control-flow edge from their corresponding parent node in the control-flow graph. Thus, our control problem is simplified to maximizing coverage by flipping frontier branches at each stage (see Section 2 for a formal definition).

Our approach estimates the potential for new coverage of a seed based on the potential for a branch distance decrease at frontier nodes. New coverage implies a decrease in branch distance, though a branch distance decrease does not necessarily imply new coverage. Since branch distance decreases are much more frequent, we use them as a reliable proxy and an upper bound on the probability of achieving new coverage. The scheduler keeps track of the branch distance decreases, estimating the probability of a seed, when mutated, to flip a frontier branch with minimal additional overhead. We present a greedy online scheduling algorithm that leverages branch-distance-based probabilistic estimates of expected coverage gain to make provably optimal decisions among all possible scheduling algorithms with access to the same estimates of the probability of achieving new coverage.

Mutator. In order to overcome the limitations of randomized mutators in adapting to branch logic, we present a new mutator that utilizes branch distance information for each frontier branch. Our mutator has two components: local search and Newton's method. First, we use local search to identify new seeds reaching each frontier branch and efficiently learn an approximate linear lower bound of the branch function. Next, we generate new seeds based on Newton's method of root finding, with a high probability of flipping the target frontier branch. This approach leads to fast and robust progress for a wide range of branch distance functions, while maintaining minimal overhead compared to more expensive techniques like byte-level taint inference [13, 26, 37].

Our design not only results in coverage gains but also provides actionable fine-grained, frontier-branch-specific feedback for fuzzing developers and users to debug and optimize their setup, going beyond the current work on fuzzing interpretability [3]. For instance, our scheduler can estimate the different sources of difficulty in flipping frontier branches, including the error in the linear approximation of a branch and the rate of reachable samples to a branch.

Result Summary. To evaluate the effectiveness of our control-theoretic approach to fuzzing, we implement FOX and compare it against the leading state-of-the-art fuzzers: AFL++, AFL++ with cmplog, AFL++ with cmplog and dictonary. We perform an extensive evaluation involving over 6 CPU-years of computation of the coverage achieved by FOX and the other evaluation candidates on a set of 38 programs (23 from the FuzzBench dataset [43] as well as 15 from a manually curated dataset of complex real-world programs). FOX outperforms all existing state-of-the-art fuzzers on standalone programs, achieving average improvements as high as 26.45% compared to AFL++, 16.98% compared to AFL++ with cmplog, and 12.90% compared to AFL++ with cmplog and dictionary. The same performance trend is reflected in the FuzzBench dataset where we see FOX gain a coverage improvement up to 6.59% over AFL++ including an average performance improvement of 3.50% when compared across all fuzzers. In addition, we evaluated the bug discovery capabilities of FOX, comparing its performance against the other candidate fuzzers on Magma dataset [29] with ground-truth bugs as well as its ability to find bugs in the wild for real-world applications. On the Magma dataset, FOX uncovers up to 16.67% more ground-truth bugs compared to other fuzzers. Finally, FOX

also uncovered 20 unique bugs as part of its in-the-wild evaluation of real-world programs, of which eight were previously unknown and have been responsibly disclosed to the affected vendors.

In summary, we make the following contributions:

- Formulating fuzzing as an online stochastic control problem, presenting a unified framework for reasoning about the scheduler and mutator components in tandem.
- Performing coverage exploration by scheduling frontier branches, drastically reducing the scheduler's control space while employing finer-grained feedback in the form of branch distance.
- Designing branch-aware mutators using optimization-oriented strategies to gain new coverage and provide interpretable feedback about frontier branches' characteristics.
- Introducing FOX as a proof-of-concept of our stochastic-control-guided approach, outperforming existing state-of-the-art fuzzers (AFL++, AFL++ cmplog, AFL++ cmplog with dictionary) on FuzzBench as well as a set of complex real-world programs, achieving coverage improvements up to 26.45%.
- Releasing FOX at https://github.com/FOX-Fuzz/FOX to foster further research and collaboration within the research community.

2 Theory

2.1 Problem Definition

Coverage-guided mutation-based fuzzing can be defined as an online optimization problem aiming to maximize the edge coverage of a target program. Let us begin with an arbitrary target program denoted as P, which can take inputs up to length M bits. A fuzzer maintains a seed corpus of inputs and iteratively mutates these seeds to generate new inputs. These mutated inputs are then executed by the target program to determine the coverage achieved on program edges.

At each fuzzing iteration i, the fuzzer selects a seed $S_i[u_i]$ from the current seed corpus S_i, where u_i represents the index of the chosen seed. Next, the fuzzer applies the mutator $mut(\cdot)$ to $S_i[u_i]$, generating a new input $\mathbf{x}$. The target program is then executed with this mutated input, and the achieved edge coverage increase is calculated using the function $cov(\mathbf{x}, S_i)$ over the mutated input $\mathbf{x}$ and the seed corpus S_i.

Based on the coverage outcome, the fuzzer decides whether/how to update the seed corpus S_i to S_{i+1}. Typically, popular fuzzers add $\mathbf{x}$ to S_i if it leads to a coverage increase.

The objective is to maximize the accumulated coverage across K stages. This can be formally represented as follows:

$$Maximize \sum_{i=1}^{K} [cov(mut(S_i[u_i]), S_i)] \tag{1}$$

Here, $S_i[u_i]$ denotes the seed selected for mutation at stage i, and $cov(mut(S_i[u_i]), S_i)$ represents the coverage achieved after executing the mutated input on the target program.

The optimal strategy for the fuzzer to achieve the highest coverage for a given target program is to find the sequence of indices u_i into the seed corpus S_i that maximizes the objective function representing the accumulated coverage across K stages. This problem is an online optimization problem [28] because the fuzzer can only observe the output of the *cov* function after applying the mutation operation at each stage sequentially. In an online optimization setting, the decision-maker (in this case, the fuzzer) must make choices at each stage without complete knowledge of the objective function's values for future stages. The objective of the fuzzer is to maximize the cumulative coverage over the sequence of stages, leading to an online optimization problem where the optimal sequence of seed selections and mutations needs to be found to achieve the highest coverage across the entire fuzzing process.

2.2 Fuzzing as Online Stochastic Control

One approach to addressing the online optimization problem described above (Equation 1) is to formulate it as an optimal control problem. This allows us to ground the development of fuzzing algorithms on the rich theory of optimal control. In an optimal control problem, the main goal is to find an optimal control strategy given a state space and control space. In our fuzzing scenario, the state space represents the state of the fuzzer at each stage, which is the current seed corpus S_i, and the control space corresponds to the choices the fuzzer makes, such as selecting a seed for mutation.

However, the dynamics of the target program are not fully known in advance due to the online nature of the problem. The system dynamics are revealed locally based on the actions selected by the fuzzer's mutation and control process. In essence, the fuzzer makes decisions at each stage based on the current seed corpus S_i, and then the program's behavior and response become apparent when the chosen mutated input is executed.

Moreover, most existing fuzzers employ randomized mutations to adaptively explore diverse program behaviors and various branch types. Since the input space of a fuzzer is often vast and complex, a randomized mutation approach avoids introducing specific biases toward particular inputs. Instead, it allows the fuzzer to explore a wide range of possibilities.

Due to both the mutator performing randomized mutations and the uncertainty in the target program's behavior, the system dynamic is best represented as a stochastic process. As a result, the problem at hand is considered a stochastic optimal control problem. In this context, the fuzzer aims to find an optimal mutation strategy that maximizes coverage despite the stochastic nature of the system dynamics. This probabilistic optimization approach allows the fuzzer to efficiently explore the input space and achieve effective generalization, ultimately improving coverage across diverse programs and input structures.

Formulating fuzzing as a stochastic control problem involves five key components as described below:

State Space. The state space encompasses all possible configurations or states of the system at any given stage. In the context of fuzzing, it includes information about the current seed corpus at stage i, denoted as S_i.

Control Space. The control space encompasses all the possible decisions that can influence the system's behavior. In the context of fuzzing, it refers to the choice of which seed, denoted with index u_i in the seed corpus S_i ($1 \leq u_i \leq |S_i|$), to select at each stage.

Fuzzer Dynamics. The dynamics summarize how the system evolves over stages based on the chosen actions and the current state. In fuzzing, it is characterized by three main steps in the mutation stage:

First, the fuzzer takes the seed corpus S_i and selects a seed $S_i[u_i]$ based on the control strategy. It then generates a new input $\mathbf{x}$ by applying the mutator $mut(\cdot)$ to the selected seed $S_i[u_i]$. Next, the fuzzer executes the generated mutant on the target program, resulting in a coverage value denoted by g_i. This coverage value represents the extent of coverage achieved for the specific mutated input. Finally, the fuzzer updates the seed corpus (i.e., state) S_i to create a modified seed corpus S_{i+1} based on the coverage result. This step involves incorporating the new input and its corresponding coverage into the seed corpus for the subsequent stages.

As mentioned earlier, the dynamics of the fuzzing process are stochastic due to the random nature of the mutator. Formally, we define the dynamics steps below:

$$\begin{aligned} \mathbf{x} &\leftarrow mut(S_i[u_i]) \\ g_i &= \mathbb{E}\left[cov(\mathbf{x}, S_i)\right] \\ S_{i+1} &\leftarrow update(S_i, \mathbf{x}, g_i) \end{aligned} \tag{2}$$

Objective Function. The objective function defines the goal or desired outcome of the system. In the context of fuzzing, it aims to maximize the expected total coverage gain across K stages:

$$Maximize \sum_{i=1}^{K} g_i \tag{3}$$

Constraints. These represent restrictions that need to be satisfied by the system. Fuzzing is typically subject to resource constraints like that the total execution time (i.e., the sum of execution time for each stage t_i) must not go over the total time budget T of the fuzzing campaign:

$$\sum_{i=1}^{K} t_i \leq T \tag{4}$$

2.3 Decomposing $cov(\mathbf{x}, S_i)$

The coverage function depends on both the mutated input $\mathbf{x}$ and the current set of seen edges based on S_i. Formally, $cov(\mathbf{x}, S_i)$ can be modeled as a random variable that counts how many unseen (i.e., not reached before by any input) edges at stage i have been flipped by the randomized mutation operation performed on $\mathbf{x}$. To achieve this, we use indicator functions $cov_b(\mathbf{x}, S_i)$, each corresponding to an unseen branch $b \in B_i$, where B_i represents the set of unseen branches at stage i. These indicator functions are modeled as random variables taking values of 0 or 1, indicating whether the unseen branch b is flipped or not. Utilizing the linearity of expectation, we can express the expectation of the cov function in the following manner:

$$g_i = \sum_{b \in B_i} \mathbb{E}\left[cov_b(\mathbf{x}, S_i)\right] \tag{5}$$

Most modern fuzzers employ customized randomized mutators, denoted as $mut(\cdot)$, which essentially generate new inputs drawn from an unknown distribution specific to the fuzzer. In our analysis, we assume that the inputs generated by the mutator have a fixed, yet unknown, probability of flipping a given unseen branch b (denoted as $\Pr(b \text{ flips} \mid mut(S_i[u_i]))$). Furthermore, different program inputs generated by $mut(\cdot)$ for different seeds are independently distributed. Popular existing fuzzers like AFL++ satisfy these assumptions. Mutators such as `havoc` randomly alter inputs without targeting specific branches, thus maintaining an equal chance of flipping any branch for a given seed. Likewise, the creation of one input does not influence the creation of another unless a branch is flipped, resulting in coverage gain.

As a result of these assumptions, we can model the coverage of a specific branch b, denoted as $cov_b(\mathbf{x}, S_i)$, where $\mathbf{x} \leftarrow mut(S_i[u_i])$, as a random variable X following a Bernoulli distribution with the probability of success given by $\Pr(b \text{ flips} \mid mut(S_i[u_i]))$.:

$$g_i = \sum_{b \in B_i} \Pr(b \text{ flips} \mid mut(S_i[u_i])) \tag{6}$$

To maximize the expected coverage gain in stage i, we need a good estimate for $\Pr(b \text{ flips} \mid mut(S_i[u_i]))$. However, the conventional approach of observing the frequency with which $mut(S_i[u_i])$ flips the branch does not work for our task. This is because using this approach, the probability for not-yet-flipped branches will be zero, whereas this probability is irrelevant to us once we have flipped a branch.

To address this issue, we utilize the concept of frontier nodes which is closely related to the concept of horizon nodes introduced by She et al. [49] in the context of graph-centrality-based seed scheduling. Consider a control flow graph (CFG) as $G = (N, E)$, where N denotes a set of basic blocks and E denotes control flow transitions between these blocks. Given a seed corpus S, we can classify N into a set of unvisited nodes U and another set of visited nodes V based on the code coverage information ($S.cov$ indicates whether a node n has been reached by the seed corpus S):

$$\begin{aligned} V &= \{n | n \in N, S.cov(n) = 1\} \\ U &= \{n | n \in N, S.cov(n) = 0\} \end{aligned} \tag{7}$$

Unlike She et al. [49] who identify horizon nodes as a set of unvisited nodes lying at the boundary between visited and unvisited code regions, we define frontier nodes as a set of visited nodes that dominate all the unvisited nodes:

$$F = \{v | (v, u) \in E, v \in V, u \in U\} \tag{8}$$

For the purposes of using frontier nodes to understand coverage, we focus solely on the subset of frontier nodes containing control instructions that result in conditional jumps. Therefore, in our setting, each frontier node is associated with a conditional jump, i.e., a **frontier branch**. Frontier branches at stage i are denoted by FB_i. We assume that each frontier branch b at stage i involves the evaluation of a predicate Q_b. The predicate evaluates to true or false based on a relation $f_b(\mathbf{x})\ R_b\ 0$, where R_b represents the condition type ($<, \leq, >, \geq, =, \neq$) and $\mathbf{x}$ indicates a test input. **Flipping a branch** indicates finding $\mathbf{x}'$ such that $Q_b(f_b(\mathbf{x}')) \neq Q_b(f_b(\mathbf{x}))$. As part of fuzzing instrumentation added to the target program, we assume that our fuzzer has access to the 2-tuple $(Q_b(f_b(\mathbf{x})), \mathbf{R_b})$ for each frontier branch $b \in FB_i$. With this information, we define branch distance function δ_b as a linear or piece-wise linear function of $f_b(\mathbf{x})$ for each frontier branch b as given in Table 1.

Consider a simple example: a branch b `if` $(x \leq 15)$, meaning $f_b(x) = x - 15$, with a reaching input $x = 5$. The branch is frontier because we have not found an input that exceeds 15. We aim to

define a linear branch distance function that outputs how close x is to exceeding 15. For an input $x = 5$, $Q_b(f_b(x)) = true$ (i.e., $x \leq 15$ is true) and $R_b = \leq$. Consequently, according to Table 1, the employed function is $1 - f_b(x)$ or $16 - x$. Finding the root input of $16 - x$, i.e. $x = 16$, will flip the frontier branch b.

Table 1: Branch distance function δ_b for a frontier branch $b \in FB_i$ based on the 2-tuple $(Q_b(f_b(\mathbf{x})), R_b)$. 2-tuples mapping to the same branch distance function are grouped together.

$Q_b(f_b(\mathbf{x}))$	R_b	$\delta_b(\mathbf{x})$
false	$<$	$f_b(\mathbf{x}) - 1$
true	$\geq$	
false	$\leq$	$f_b(\mathbf{x})$
true	$>$	
false	$>$	$1 - f_b(\mathbf{x})$
true	$\leq$	
false	$\geq$	$-f_b(\mathbf{x})$
true	$<$	
false	$=$	$\lvert f_b(\mathbf{x})\rvert$
true	$\neq$	
false	$\neq$	$1 - \lvert f_b(\mathbf{x})\rvert$
true	$=$	

Due to the nature of existing mutators as discussed above, we further assume that the program inputs generated by the mutator have an unknown but fixed probability of decreasing branch distance δ_b for a given unseen branch b ($\Pr(\delta_b$ decreases $\mid mut(S_i[u_i]))$). Therefore, we can model the event $\delta_b(\mathbf{x})$ decreases, $\mathbf{x} \leftarrow mut(S_i[u_i])$ as a random variable $Y \sim Bernoulli(\Pr(\delta_b$ decreases $\mid mut(S_i[u_i])))$. While it is evident that X implies Y (every flip always involves a branch distance decrease), it is not necessarily true that Y implies X (not every branch distance decrease involves a flip). Hence, we find that: $\Pr(b$ flips $\mid mut(S_i[u_i])) \leq \Pr(\delta_b$ decreases $\mid mut(S_i[u_i]))$. Therefore, optimizing based on p' serves as an approximation to optimizing based on p, where p' and p are the probabilities of δ_b decreasing and b flipping, respectively.

Naturally, p' cannot be used to reason about unseen branches, i.e., branches that have not been reached yet, since we lack branch distance δ_b information for them. Thus, we focus solely on frontier branches FB_i — branches that we have reached but not yet flipped.

$$g_i \approx \sum_{b \in FB_i} \Pr(\delta_b \text{ decreases} \mid mut(S_i[u_i])) \tag{9}$$

To optimize Equation 9, we use a two-pronged approach. First, we choose to schedule $S_i[u_i]$ for each stage i such that it maximizes the above expression. Second, assuming that most frontier branches in a program are approximately linear in the neighborhood of each seed in S_i, we introduce a custom mutation operation such that the $\Pr(\delta_b$ decreases $\mid mut(S_i[u_i]))$ increases for these branches.

2.4 Fuzzing Algorithm

2.4.1 Optimal-control-based Scheduler. We solve the following optimization problem using a greedy approach.

$$Maximize \sum_{i=1}^{K} \sum_{b \in FB_i} \Pr(\delta_b \text{ decreases} \mid mut(S_i[u_i])) \tag{10}$$

To optimize each stage i efficiently, we aim to select a seed $S_i[u_i]$ that maximizes the inner sum in Equation 10. Given that many of these probabilities are very small (0 or close to 0), we can approximate the inner sum by taking the maximum term instead. This approximation is not only effective but also computationally inexpensive in a streaming setting.

$$Maximize \sum_{i=1}^{K} \max_{b \in FB_i} \Pr(\delta_b \text{ decreases} \mid mut(S_i[u_i])) \tag{11}$$

We can tackle this problem in two steps: (a) for each branch b, find a seed $S_i[u_i]$ that maximizes $\Pr(\delta_b$ decreases $\mid \text{mut}(S_i[u_i]))$ and (b) select the frontier branch $b \in FB_i$ with the maximum $\Pr(\delta_b$ decreases $\mid \text{mut}(S_i[u_i]))$. For step (a), we maintain a mapping TS where each frontier branch b at stage i corresponds to a seed $TS_i[b].s$ (along with the corresponding branch distance $TS_i[b].d$) that achieves the lowest branch distance across all inputs reaching that frontier branch so far. Subsequently, scheduling can be simplified to step (b) as shown below.

$$\begin{aligned} u_{i+1} = TS_i[&\arg\max_{b \in FB_i} \Pr(\delta_b < TS_i[b].d \mid mut(TS_i[b].s)] \\ TS_{i+1}[b].s &= \arg\min_{s \in S_{i+1}} \delta_b(s) \\ TS_{i+1}[b].d &= \min_{s \in S_{i+1}} \delta_b(s) \end{aligned} \tag{12}$$

To estimate the branch distance decrease probability for a frontier branch, we measure the number of total executions of inputs that reach the frontier branch, denoted as $th(\cdot)$, and the total number of inputs that lower the global minimum branch distance for the frontier branch, denoted as $ph(\cdot)$. To incorporate the time constraint, we further refine the probability estimate by replacing the number of hits with time. Specifically, we track the total time spent executing inputs that reach a frontier branch $tt(\cdot)$ and the total time spent executing inputs lowering the branch distance for a frontier branch $pt(\cdot)$. Additionally, to prevent frequent scheduling of the same seeds, we introduce a discount factor λ_b based on how many times the seed mapped to the frontier branch b was scheduled. The final probability estimate is then given by:

$$\Pr(\delta_b \text{ decreases}) = \lambda_b \frac{pt(b)}{tt(b)} \tag{13}$$

Theorem 1. *Given a fixed branch flip probability before each stage i, a greedy schedule that chooses u_{i+1} such that it maximizes the expected coverage gain at each stage i of the problem described by Equation 3 is optimal.*

A proof of the theorem can be found in §A.1.

2.4.2 Optimizing Mutation . In our approach, the primary objective of the mutation stage is to enhance the likelihood of reducing branch distances for all frontier branches at any given stage. To

achieve this, we begin by creating a locally correct linear under-approximator through local search, essentially a subgradient. If the slope of this approximator is non-zero (indicating a decreasing direction), we proceed with the Newton's method.

Local Search. The central idea is to construct a local linear approximation of the branch distance function, denoted as $\delta_b(\mathbf{x})$, for a given branch b. This local approximation should act as a lower bound on δ_b within a neighborhood $N(\mathbf{x})$ surrounding the point $\mathbf{x}$. The local neighborhood depends on the number of bytes modified by the mutator in stage i. Unlike aiming to minimize the average error over the points within the neighborhood, we aim to minimize the under-approximation error. This choice ensures that the Newton's method remains stable.

To achieve this, we seek a vector g that satisfies the condition $\forall \mathbf{x}' \in N(\mathbf{x}) : g^T \cdot (\mathbf{x}' - \mathbf{x}) \leq \delta_b(\mathbf{x}') - \delta_b(\mathbf{x})$. To approximate g, we resort to a randomized local search, generating a fixed-size sample of program inputs $\mathbf{x}' \in N(\mathbf{x})$ using $mut(\cdot)$. We estimate the value of g by selecting $g = \arg\max_{g'} \|g'\|_1, g' = (\delta_b(\mathbf{x}') - \delta_b(\mathbf{x})) \oslash (\mathbf{x}' - \mathbf{x})$, where $\oslash$ indicates element-wise division.

The success of local search hinges on finding the right balance between accuracy and speed. A large stack of havoc mutations leads to sampling over a vast neighborhood $N(\mathbf{x})$, rendering an approximation imprecise. Consequently, by fine-tuning the mutation stack we can constrain the neighborhood in a manner that allows for relatively more precise approximations. Details pertaining to this fine-tuning of the mutator stack as implemented in FOX are discussed in §3.4.

Newton's Method. If we obtain a valid g (i.e., non-zero norm) in the local search, we utilize it to perform the Newton's method along the direction of g and identify the point at which the branch flips. Considering that we want to find an $\mathbf{x}$ such that $\delta_b(\mathbf{x})$ takes a value of 0 and flips the branch, we apply Newton's method to generate a new input $\mathbf{x} = \mathbf{x} - \delta_b(\mathbf{x}) \oslash g$. If the underlying function $\delta_b(\mathbf{x})$ is linear and the branch is feasible, this step will cause the branch to flip. However, if $\delta_b(\mathbf{x})$ is not linear but a well-behaved convex function, the step is still likely to decrease the branch distance [18].

3 Implementation

In this section, we present FOX, our proof-of-concept implementation for modeling fuzzing as an online stochastic control problem. Two essential primitives for FOX to implement its stochastic-control-guided strategy include: (i) identifying frontier branches and (ii) calculating branch distances. We first discuss how FOX manages these two primitives, outlining the strategies employed for efficient tracking. Subsequently, we discuss the implementation of scheduling and mutation strategies, as described formally in § 2. Finally, we explore how FOX leverages the semantics of string comparisons to effectively flip frontier nodes involved in such comparisons.

3.1 Frontier Branch Identification

FOX uses an intra-procedural control-flow graph (CFG) in conjunction with coverage information obtained during a fuzzing campaign to identify frontier branches dynamically. FOX augments the AFL++ LLVM-based instrumentation pass to extract the intra-procedural CFG for each function and embeds its metadata in the binary.

During a fuzzing campaign, we dynamically mark nodes in the CFG as either visited or unvisited based on coverage information as defined in Equation 7. Additionally, we update the list of frontier nodes following Equation 8. We only focus on frontier branches, a subset of frontier nodes where each node contains control instructions resulting in conditional jumps as outlined in §2.3.

3.2 Branch Distance Tracking

To facilitate the tracking of branch distances, we employ an LLVM pass to hook every branch instruction and obtain its corresponding branch distance. Specifically, we examine each conditional branch instruction and check if its condition is computed from a CMP instruction. If so, we insert a function immediately after the CMP instruction to intercept its two operands, `op1` and `op2`.

Integer Comparison. We compute the branch distance $\delta_b(\mathbf{x}) =$ `op1` $-$ `op2`.

String Comparison. We calculate an array of byte-wise branch distances $op1[i] - op2[i], 1 \leq i \leq k$, where k corresponds to $\max(\text{len}(str1), \text{len}(str2))$. We add zero-byte padding to ensure both strings have the same length if one is shorter than the other.

During the dynamic execution of instrumented programs, our hook functions compute the branch distances and save them into a shared memory accessible by FOX. In real-world programs, there can be many branch instructions, and invoking a hook function for every branch instruction would cause significant runtime overhead during dynamic execution. To reduce runtime overhead, we implement an adaptive switch for each hook function. The switch ensures that the hook function is invoked only for frontier branches, not for fully explored branches where all children nodes have been visited. This way, we only incur a relatively small runtime overhead while accurately computing branch distances for a small set of frontier branches.

3.3 Scheduler

Our seed scheduler is implemented according to Algorithm 1, following the theoretical description provided in §2.4. We maintain a mapping of seeds that achieve the lowest branch distance for each frontier branch TS. This approach allows us to reason about frontier branches rather than individual seeds, mitigating the target explosion issue discussed by Mouret et al. [44].

We also keep counters for each frontier branch's productive time PT and total time TT indexed by the branch's unique CFG node id. These counters, along with the mapping TS, get updated after each program input execution. Before scheduling the next seed for each stage i, we compute the set of unvisited nodes U_i and the set of frontier branches FB_i following the definitions in §2.3.

To compute the probability, we define the seed decay factor λ_b for each frontier branch b as $SC[TS[b.id]]$, where SC is an array of schedule counters indexed by the seed. Instead of directly computing $\Pr(\delta_b \textit{ decreases})$ for each frontier branch b, which might suffer from numerical underflows if $PT[b.id]$ is much smaller than $TT[b.id] \times \lambda_b$, we compute the log-probability, as it avoids underflow issues.

Algorithm 1 FOX scheduling algorithm for stage i

Input: $PT \leftarrow$ CFG node id to productive time mapping
$TT \leftarrow$ CFG node id to total time mapping
$TS \leftarrow$ CFG node id to lowest branch distance seed mapping
$SC \leftarrow$ Seed to number of scheduled count mapping

$b\_max = \emptyset$
$b\_max\_logprob = \infty$
for $f \in FB_i$ **do**
  $b\_logprob = \log(PT[b]) - \log(TT[b]) - \log(SC[TS[b].s])$
  if $b\_max\_logprob < b\_logprob$ **then**
    $b\_max = b$
    $b\_max\_logprob = b\_logprob$
$u_i = TS[b\_max].s$ ▷ Schedule *top_seed*
$SC[u_i]$ += 1

Algorithm 2 FOX mutator

Input: $\delta \leftarrow$ branch distance function for each frontier branch b
$S_i \leftarrow$ Seed corpus
$u_i \leftarrow$ Scheduled seed index
$k \leftarrow$ Local search sample size

/* Perform local search with sample size k */
$G = empty\_map()$
$X = empty\_map()$
for k *iterations* **do**
  $\mathbf{x} = mut(S_i[u_i])$
  for $f \in FB_{i,\mathbf{x}}$ **do**
    $g = \text{ComputeSubgradient}(\mathbf{x}, S_i[u_i], \delta_b)$
    if $b.id \notin X \vee \|g\|_1 > \|G[b.id]\|_1$ **then**
      $G[b.id] = g$
      $X[b.id] = \mathbf{x}$
/* Apply Newton method to the top input for each reached frontier branch */
for $f \in FB_i$ **do**
  if $b.id \in X.keys$ **then**
    $\mathbf{x}^* = \text{ApplyNewtonMethod}(X[b.id], S_i[u_i], \delta_b)$
  $exec(\mathbf{x}^*)$

3.4 Mutator

Our mutator is implemented as shown in Algorithm 2, following the theoretical description provided in §2.4.2. Given the scheduled seed $S_i[u_i]$, we perform a randomized local search by sampling $k = 1024$ program inputs $\mathbf{x}$ in the neighborhood $N(S_i[u_i])$. We empirically determined this to be a good trade-off between the number of local searches performed and Newton's method steps taken. We utilize AFL++ `havoc` mode stochastic mutator [5] for generating mutants, with the number of random perturbations of the seed reduced to keep them within the local neighborhood $N(S_i[u_i])$.

For each generated mutant $\mathbf{x}$, we determine the subset of frontier branches $FB_{i,\mathbf{x}} \subseteq FB_i$ it reaches and compute the subgradient $g_{\mathbf{x},b}$ for each frontier branch $b \in FB_{i,\mathbf{x}}$. We only consider a mutant for the Newton's method when the L1 norm of $g_{\mathbf{x},b}$ is greater than the maximum subgradient value $G[b.id]$ encountered so far. We then apply Newton's method to derive new inputs $\mathbf{x}^*$ for the reached frontier branches and execute them on the target program.

For frontier branches that rely on string comparisons, we have taken a different approach. We treat string comparison functions like `strcmp` and `strncmp` as sequences of multiple single-byte integer comparisons and solve them together using Newton's method. In contrast to the standard local search, for these functions, we take an additional step to analyze the byte differences between the seed $S_i[u_i]$ and the mutant $\mathbf{x}$ to investigate the effect of a branch distance δ_b change.

For each byte difference, we create a new program input by applying only a single byte difference to the seed and then execute this modified input to observe if it leads to a change in branch distance. If we detect a branch distance change, we identify the specified byte difference in the input as a hot (i.e., influential) byte, indicating that it directly influences a particular byte in the string comparison.

Our approach is designed to efficiently identify hot byte locations without the need to probe all byte locations, which is done in many previous works [13, 26, 38]. Once we find a hot byte location and know the length of the string being compared, we infer all the other hot byte locations by exploiting the fact that the hot bytes for a string comparison are typically adjacent.

4 Evaluation

In this section, we aim to answer the following research questions:

- **RQ1:** Can FOX enable testing code that was previously unreachable by state-of-the-art fuzzers?
- **RQ2:** Does FOX improve fuzzers' bug discovery capabilities?
- **RQ3:** How much control space reduction can FOX achieve compared with existing fuzzers?
- **RQ4:** To what degree do the individual components of FOX contribute to its overall performance?
- **RQ5:** Are there characteristics that make a branch amenable to be solved by FOX?

We perform a thorough evaluation of FOX against AFL++ version 4.09c (AFLPP) [24], which is the latest and most performant version of a widely-recognized state-of-the-art fuzzer [14, 39], topping the December 2023 FuzzBench report [2]. Additionally, we extend our comparison to AFL++ in `cmplog` mode (AFLPP+C), an industry implementation of REDQUEEN [13]. The AFLPP+C mode intercepts the operands of CMP instructions and applies tailored mutations. We specifically opt for this comparison because the mutation technique of AFLPP+C most closely aligns with our own mutator. The most potent setting of AFL++, widely embraced in industrial environments [4] and academic fuzzing competitions [8], is AFL++ with both `cmplog` and `dictionary` modes (AFLPP+CD). The `dictionary` mode of AFL++ automatically generates a fuzzing dictionary comprising string constants and integer constants extracted from the tested program. We also incorporate dictionaries provided by FuzzBench. In addition to FOX alone, we evaluate FOX with `dictionary` (FOX+D) to showcase its performance in a fair comparison against AFLPP+CD.

Benchmarks. For our evaluation, we use all 23 binaries from the FuzzBench dataset [43]. While FuzzBench is widely acknowledged

Table 2: Studied programs in our evaluation.

Targets	Version	# Edge	Targets	Version	# Edge
FuzzBench Targets			systemd	07faa49	127,142
bloaty	52948c1	163,494	vorbis	84c0236	13,729
curl	a20f74a	122,270	woff2	8109a2c	19,615
freetype	cd02d35	34,951	zlib	d71dc66	4,640
harfbuzz	cb47dca	78,203	**Standalone Targets**		
jsoncpp	8190e06	9,826	bsdtar	libarchive-3.6.2	38,244
lcms	f0d9632	14,693	exiv2	exiv2-0.28.0	122,543
libjpeg-t	3b19db4	17,789	ffmpeg	ffmpeg-6.1	741,421
libpcap	17ff63e	15,344	jasper	jasper-4.1.2	19,122
libpng	cd0ea2a	9,038	nm-new	binutils-2.34	54,848
libxml2	c7260a4	81,782	objdump	binutils-2.34	80,582
libxslt	180cdb8	61,727	pdftotext	xpdf-4.04	53,683
mbedtls	169d9e6	28,373	readelf	binutils-2.34	32,249
openh264	045aeac	19,599	size	binutils-2.34	54,348
openssl	b0593c0	80,662	strip-new	binutils-2.34	61,051
openthread	2550699	50,574	tcpdump	tcpdump-4.99.4	47,476
proj4	a7482d3	267,147	tiff2pdf	libtiff-v4.5.0	21,006
re2	b025c6a	15,179	tiff2ps	libtiff-v4.5.0	17,861
sqlite3	c78cbf2	77,154	tiffcrop	libtiff-v4.5.0	20,096
stb	5736b15	7,661	xmllint	libxml2-2.9.14	85,412

as a standard in fuzzer evaluation [14, 17, 39, 46], we observe that it suffers from two significant challenges: many project harnesses are either very small (about a third of binaries have less than 20,000 edges, see Table 2) or feature numerous test cases, leading to an early coverage saturation. This makes it challenging to differentiate the performance differences among fuzzers. Notably, in SBFT23, a FuzzBench-based competition, the mean coverage gain of the winner HasteFuzz over AFLPP+CD was a modest 1.28% [9]. To address these limitations and provide a more comprehensive assessment, we further evaluate FOX on 15 real-world standalone programs. These programs are carefully selected to represent the diversity of our current software ecosystem, encompassing a broad range of functionalities based on prior fuzzing literature [32, 49, 51, 57]. The details of these programs, along with their corresponding commit/version, are presented in Table 2. Our results demonstrate that the improvements achieved by FOX generalize beyond FuzzBench.

Experimental Setup. To answer **RQ1**, we evaluate the coverage over time achieved by all fuzzers. For **RQ2**, we measure the number of real bugs found in the standalone programs as well as the time taken by the fuzzers to uncover the ground truth bugs present in the MAGMA dataset [29]. For **RQ3**, we quantify the number of frontier branches that FOX explores and compare it to the number of seeds that AFL++, a conventional coverage-guided fuzzer, has to schedule. To answer **RQ4**, we perform an ablation study by comparing the coverage achieved by FOX against its variant with the optimized mutator turned off. Finally, for **RQ5**, we propose and leverage a set of quantitative metrics to interpret the performance of FOX.

We follow standard fuzzing evaluation practices [33] and run 10 campaigns of 24 hours each for each of the fuzzer-program pairs, totaling over 5 CPU-years of evaluation. As part of our evaluation on the FuzzBench dataset, we use the seeds as recommended in the benchmark. For our standalone program evaluation, we provide one well-formed seed for each of the targets sourced from the AFL++ repository [1]. To ensure fairness in our comparison, each fuzzer is assigned a single core for each of their fuzzing campaigns. In addition to standard summary statistics like mean and standard deviation over our previously mentioned evaluation metrics, we also perform the Mann-Whitney U test to ensure the observed performance differences are statistically significant. Since this test is non-parametric and therefore does not make any assumptions about the underlying distribution, it has been widely used in the software testing literature for testing randomized algorithms [12] and fuzzers [33]. All our experiments are conducted on 10 machines with Intel Xeon 2.00 GHz processors running Debian bullseye with 100 GB of RAM.

4.1 RQ1: Code Coverage

To answer **RQ1**, we evaluate the code coverage achieved by FOX and compare it against the existing state-of-the-art fuzzers across 23 FuzzBench and 15 standalone targets on 10x24 hour campaigns. To quantify the coverage achieved, we leverage the coverage collection module of AFL++ that keeps track of the number of control-flow edges exercised during the run using its coverage bitmap. In addition, we compare the mean coverage growth over time along with their standard deviations to showcase the stability of our approach in achieving coverage saturation. The Mann-Whitney U test scores as well as the total number of executions statistics can be found in the extended version of this work [50].

FuzzBench Targets. Across FuzzBench targets (see results in Table 3), FOX demonstrates superior performance compared to AFL++ on 18 programs and AFLPP+C on 13 programs, achieving significant mean coverage improvements. For instance, `freetype` shows up to a 39.97% coverage increase over AFL++ and `harfbuzz` exhibits a 34.46% improvement over AFLPP+C. Similarly, FOX+D outperforms AFLPP+CD on 15 programs, with notable gains of up to 21.17% on `harfbuzz`. Coverage progression over time, comparing FOX and FOX+D against other fuzzers, is detailed in the extended version of our work [50].

FOX performs comparably or with statistically insignificant differences to AFLPP+C on all remaining programs except for `lcms`, `openthread`, `proj4`, and `zlib`. The challenges in these targets stem from nested conditions and string comparisons with both operands as variables, requiring FOX to further adapt the Newton's method to handle these complex branch types. While our current prototype does not support them, it does not indicate a design limitation of FOX; we plan to extend its capabilities in future iterations (discussed further in §5).

With the dictionary, addressing the bottleneck of variable string comparisons, FOX+D excels on larger FuzzBench programs with quality seed corpora like `harfbuzz` and `sqlite3`. Particularly noteworthy is its ability to manage control space more efficiently by focusing solely on frontier branches, reducing redundancy and maximizing time allocation. This effect is elaborated on in §4.3. Even on relatively small programs, the Newton's method mutator

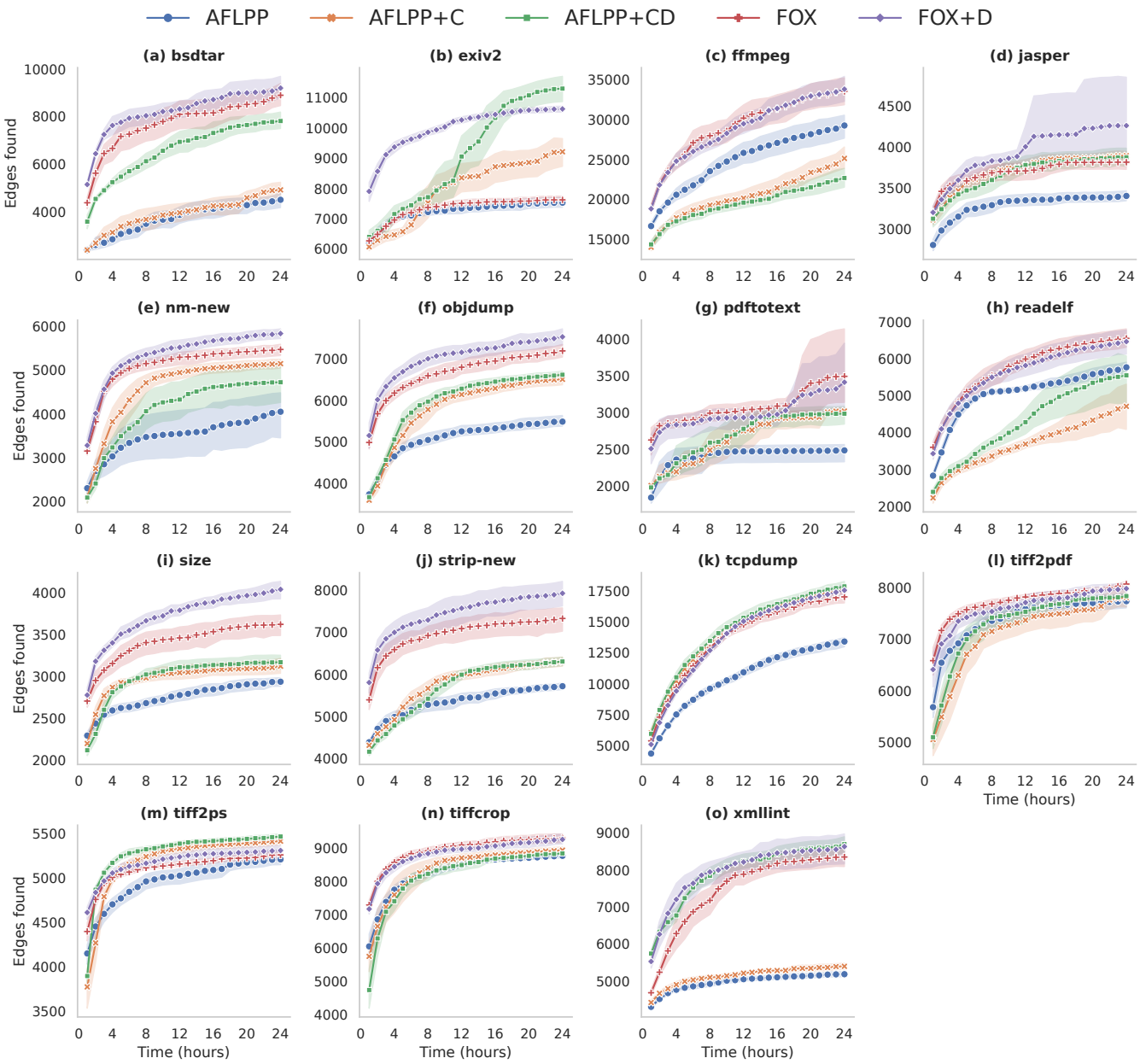

Figure 2: The arithmetic mean edge coverage for FOX and FOX+D against three other fuzzers running for 24 hours over ten runs on the standalone programs. The error bars indicate one standard deviation.

allows FOX to flip branches that neither AFL++ nor AFLPP+C can flip. We further discuss this ability in §4.5.

Standalone Targets. Comparison of mean coverage achieved by FOX against AFL++, AFLPP+C, and AFLPP+CD is presented in Table 4. Across standalone programs, FOX exhibits improvements over AFL++ on all programs and surpasses AFLPP+C on 11 programs, achieving up to 26.45% more code coverage on average across the standalone target set. One of the targets where we see notable improvement is `bsdtar`, where FOX uncovers 97.25% and 80.61% more edges than AFL++ and AFLPP+C, respectively. Similarly, FOX+D outperforms AFLPP+CD on 11 programs, including a remarkable 49.04% improvement on `ffmpeg`.

Given the larger size of our selected standalone programs compared to FuzzBench programs, tight integration of all fuzzer components becomes critical. Through comprehensive control space optimization, FOX effectively narrows the extensive control space further by prioritizing branches suitable for flipping by the Newton's method mutator, thereby enhancing coverage. In contrast,

AFL++ relies solely on prior observed execution behavior to schedule seeds, lacking guidance on which seeds would be most effective in expanding coverage.

Result 1: FOX outperforms existing state-of-the-art fuzzers by achieving up to 26.45% more coverage on average across the standalone targets and up to 6.59% more coverage on average across the FuzzBench targets.

Table 3: Mean edge coverage of FOX and FOX+D against three fuzzers on 23 FuzzBench programs for 24 hours over 10 runs. We mark the highest number in bold.

Targets	FOX	AFLPP	AFLPP+C	FOX+D	AFLPP+CD
bloaty	**8,646**	8,507	8,627	8,514	**8,684**
curl	**14,674**	14,510	14,358	**15,811**	15,359
freetype	**14,676**	10,485	13,401	**15,536**	13,812
harfbuzz	37,155	**37,276**	27,633	**37,103**	30,621
jsoncpp	1,341	1,341	**1,342**	1,342	**1,343**
lcms	1,224	728	**2,451**	**1,976**	1,882
libjpeg-turbo	3,293	**3,299**	3,297	3,283	**3,301**
libpcap	**2,366**	43	1,848	**3,074**	2,690
libpng	**2,846**	2,697	2,838	**2,848**	2,836
libxml2	**18,757**	18,623	18,616	**19,188**	19,003
libxslt	**12,184**	11,959	11,909	**12,424**	12,358
mbedtls	**3,987**	3,722	3,839	**4,037**	3,834
openh264	**13,780**	13,733	13,630	**13,826**	13,662
openssl	**11,128**	11,102	11,114	**11,127**	10,902
openthread	4,827	4,686	**5,140**	4,822	**5,041**
proj4	26,603	26,703	**28,904**	**32,995**	32,494
re2	6,077	**6,190**	6,105	6,085	**6,224**
sqlite3	37,801	37,701	**37,953**	**40,028**	38,846
stb	**4,332**	4,042	4,180	**4,440**	4,249
systemd	3,836	3,781	**3,856**	3,835	**3,857**
vorbis	2,050	2,045	**2,056**	2,054	**2,055**
woff2	**2,674**	2,458	2,550	**2,678**	2,514
zlib	884	883	**917**	879	**913**
Mean gain		6.59% †	0.95%	—	2.97 %

† We omit `libpcap`, as it would unrealistically skew this statistic in favor of FOX.

4.2 RQ2: Bug Discovery Effectiveness

In **RQ2**, we evaluate the bug discovery effectiveness of FOX compared to other state-of-the-art fuzzers. Following Klees et al.'s recommendation [33] on using datasets with curated bugs, we tested our fuzzer on the Magma dataset [29], comparing its performance with other fuzzers. Additionally, to understand FOX's real-world bug detection capabilities, we evaluate it on an array of widely-used standalone programs and libraries as specified in Table 2.

Curated Bug Dataset. Magma, a popular dataset in the fuzzing community [31, 35], contains 21 programs from nine open-source libraries with injected bugs. Our evaluation used 17 programs from eight of these libraries: `libpng`, `libtiff`, `libxml2`, `lua`, `openssl`, `poppler`, `sqlite3`, and `libsndfile`. `php` and its four fuzz drivers were the only targets omitted from our evaluation since this target encountered a compilation error during the final linking stage due to dependency incompatibility. We compared FOX and FOX+D with fuzzers like AFL++, AFLPP+C, and AFLPP+CD over five 24-hour test campaigns for each test program, totalling over 1 CPU-year of evaluation.

Table 4: Mean edge coverage of FOX and FOX+D against three fuzzers on 15 standalone programs over 10 fuzzing campaigns run for 24 hours each. We mark the highest number in bold.

Targets	FOX	AFLPP	AFLPP+C	FOX+D	AFLPP+CD
bsdtar	**8,893**	4,508	4,924	**9,194**	7,811
exiv2	7,618	7,526	**9,213**	10,631	**11,306**
ffmpeg	**33,544**	29,268	25,147	**33,821**	22,692
jasper	3,811	3,400	**3,890**	**4,257**	3,870
nm-new	**5,475**	4,049	5,150	**5,841**	4,724
objdump	**7,196**	5,484	6,508	**7,536**	6,619
pdftotext	**3,490**	2,482	3,016	**3,410**	2,982
readelf	**6,551**	5,767	4,709	**6,462**	5,549
size	**3,619**	2,934	3,120	**4,037**	3,167
strip-new	**7,330**	5,718	6,297	**7,930**	6,306
tcpdump	16,996	13,399	**17,601**	17,524	**17,841**
tiff2pdf	**8,061**	7,731	7,803	**7,974**	7,830
tiff2ps	5,257	5,206	**5,411**	5,306	**5,463**
tiffcrop	**9,298**	8,763	8,933	**9,254**	8,835
xmllint	**8,347**	5,179	5,392	8,628	**8,696**
Mean gain		26.45%	16.98%	—	12.90%

The results, detailed in Table 5, reveal that FOX was the top performer in identifying unique bugs, outperforming or matching other fuzzers in all the evaluated programs. We observe that FOX's superior performance is consistent with Bohme et al.'s [17] prior observation about a strong link between coverage and bug discovery.

As an interesting side note, we observe that fewer bugs were triggered in `lua` and `openssl` compared to the other libraries. Upon investigation, we found this was due to a limited number of detectable ground-truth bugs in these programs. Specifically, `lua` had only four injected bugs [6], and in `openssl`, only four have been confirmed to be reachable [29], with several being provably unreachable [7].

In-the-wild Bug Discovery. To evaluate FOX and FOX+D's bug discovery in real-world scenarios, we compared them with leading fuzzers on a dataset of common programs and libraries (Table 2). After ten 24-hour campaigns per target, we analyzed each crash using standard deduplication practices [33]. Specifically, we employ stack hashes of the crashing inputs to perform deduplication and follow it up with manual analysis to validate deduplicated crashes are unique.

Our findings (Table 6) show FOX and AFLPP+C discovering the same number of bugs, both surpassing AFL++. FOX+D, however,

uncovered 20 unique bugs, including 2 Use After Frees (CWE-416), 3 Invalid Frees (CWE-761), 12 Assertion Violations (CWE-617), 1 Infinite Loop (CWE-835), and 2 Null Pointer Dereferences (CWE-476). Remarkably, 10 of these bugs were exclusively found by FOX+D, highlighting its superior bug discovery, especially given the heavily fuzzed nature of these targets.

Another interesting observation is that FOX+D heavily outperforms all the other evaluation candidates, uncovering eleven more bugs than the next best performing candidates (FOX, and AFLPP+C). A key contributor to this performance difference is `xpdf` where FOX+D uncovered eight more bugs compared to the other fuzzers. Looking closer at the coverage results, we see that FOX+D uncovers on average 21.93% more edges than the other fuzzers. In an effort to understand if this coverage increase is correlated with the bugs uncovered by FOX+D in `xpdf`, we measured Pearson's correlation coefficient between the edges uncovered and unique crashes discovered across all the candidate fuzzers over the ten 24-hour campaigns. We observed a strong correlation with a 0.88 coefficient. This observation is in line with previous findings of a strong correlation between edges uncovered and bugs discovered while fuzzing [17].

Finally, a testament to FOX's real-world impact is its identification of 12 bugs in `xpdf`, a popular PDF library, with eight being previously unknown as confirmed by the developers.

Table 5: Cumulative number of unique bugs triggered and reached of FOX and FOX+D against three fuzzers in Magma programs for 24 hours over 5 runs. (triggered | reached)

Targets	FOX	AFLPP	AFLPP+C	FOX+D	AFLPP+CD
tiffcp	6 | 11	5 | 8	6 | 9	7 | 10	6 | 8
tiff_read	4 | 7	3 | 5	3 | 5	3 | 7	3 | 6
libpng	3 | 6	1 | 6	3 | 6	3 | 6	3 | 6
xmllint	3 | 7	2 | 7	3 | 7	3 | 8	3 | 8
libxml2_xml	5 | 9	3 | 8	3 | 8	4 | 9	4 | 9
lua	2 | 4	1 | 2	1 | 2	1 | 2	1 | 2
asn1parse	0 | 1	0 | 1	0 | 1	0 | 1	0 | 1
bignum	0 | 1	0 | 1	0 | 1	0 | 1	0 | 1
asn1	2 | 4	2 | 4	2 | 4	2 | 4	2 | 4
client	1 | 7	1 | 7	1 | 7	1 | 5	1 | 7
server	2 | 6	1 | 6	1 | 6	1 | 4	1 | 6
x509	1 | 5	0 | 5	0 | 5	0 | 5	0 | 5
pdftoppm	3 | 16	4 | 14	3 | 14	3 | 13	3 | 13
pdf_fuzzer	3 | 16	3 | 13	2 | 13	2 | 8	2 | 12
pdfimages	4 | 13	5 | 10	4 | 11	3 | 11	4 | 11
sqlite3_fuzz	3 | 10	5 | 13	3 | 11	3 | 10	4 | 13
sndfile_fuzzer	7 | 8	7 | 8	7 | 8	8 | 8	7 | 8
total	49 | 131	43 | 118	42 | 118	44 | 112	44 | 120

Result 2: FOX triggers up to 16.67% more ground-truth bugs compared to the state-of-the-art fuzzers on Magma and when coupled with dictionary uncovers 20 unique vulnerabilities in popular programs including eight that were previously unknown.

4.3 RQ3: Control Space Reduction

In our stochastic control framework, the control space refers to the pool of seeds available for a fuzzer to select from during fuzzing.

Table 6: Cumulative number of unique bugs identified by FOX and FOX+D against three fuzzers in FuzzBench Programs and standalone programs for 24 hours over 10 runs

Targets	FOX	AFLPP	AFLPP+C	FOX+D	AFLPP+CD
nm-new	1	0	1	2	1
size	2	2	1	2	1
objdump	1	0	1	1	1
pdftotext	4	2	4	12	4
woff2	1	1	1	1	0
sqlite3	0	0	1	1	0
libxslt	0	0	0	1	0
total	9	5	9	20	7

Unlike conventional coverage-guided fuzzers such as AFL++, which choose the next seed from a seed queue, FOX takes a different approach by scheduling based on frontier branches rather than individual seeds. AFL++, as the fuzzing campaign advances, adds more seeds to the queue, complicating the decision of which seed to schedule next. To address this complexity, AFL++ employs various techniques to manage the seed queue and prioritize specific seeds.

In contrast, FOX's scheduling strategy focuses on frontier branches. To assess the reduction in control space resulting from scheduling over frontier branches instead of seeds, we compare the number of frontier branches explored by FOX with the number of seeds explored by AFL++ throughout a fuzzing campaign. This comparison provides insights into the effectiveness of FOX's approach in streamlining the decision-making process during fuzzing.

As illustrated in Figure 3, the control space of AFL++ (i.e., the number of seeds) increases monotonically as it fuzzes `xmllint`, requiring more time to iterate through the entire seed corpus. In contrast, FOX reasons over a compact set of frontier branches that does not dramatically increase as the fuzzing campaign progresses. In fact, on some targets such as `readelf`, the set of frontier branches even decreases. The control space comparison between FOX and AFL++ on 15 standalone programs at 12 and 24 hours, respectively, is presented in Table 7. FOX demonstrates a remarkable median reduction in the control space by 58.20% during a 24-hour fuzzing campaign. Particularly noteworthy is FOX's ability to reduce the control space by a factor of 7 on programs like `readelf`.

Result 3: FOX reduces the control space by 49.18% for 12-hour run and 58.20% for 24-hour run when compared with AFL++.

4.4 RQ4: Ablation Study

To quantify the contribution of the scheduling and mutation strategies employed by FOX (§4.1), we compare three different variants of FOX: (i) FOX-BASE: representing FOX with both the scheduling and mutator components deactivated and serving as the baseline; (ii) FOX-SCHED: employing only the optimization-guided scheduler; and (iii) FOX: enabling both the scheduler and the optimization-guided mutator. We measure the code coverage achieved by these three variants (FOX-BASE, FOX-SCHED, and FOX) on our 15 standalone programs across 10 trials, each lasting 1 hour.

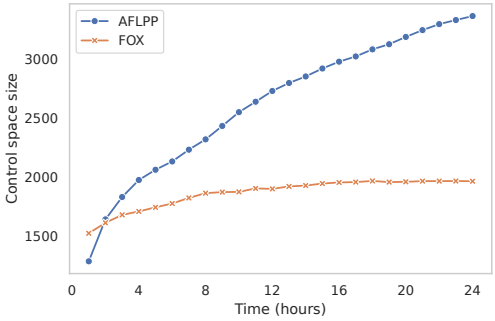

Figure 3: Control space comparison of FOX with AFLPP on `xmllint` over the course of a 24 hour fuzzing campaign.

Table 7: Control space comparison of FOX with AFLPP on 15 standalone programs at both 12 hours and 24 hours over 10 runs.

Targets	# FOX Frontier		# AFLPP Seeds	
	12 hrs	24 hrs	12 hrs	24 hrs
bsdtar	1,131	1,224	2,226	2,929
exiv2	2,953	3,057	2,121	2,446
ffmpeg	5,615	5,772	6,320	8,976
jasper	1,038	1,034	1,272	1,331
nm-new	971	944	1,970	2,772
objdump	1,299	1,285	3,239	3,940
pdftotext	929	981	1,466	1,661
readelf	1,561	1,215	7,075	8,970
size	671	670	1,470	1,692
strip-new	1,212	1,210	2,232	2,866
tcpdump	2,792	3,019	3,642	4,946
tiff2pdf	1,755	1,768	4,125	4,670
tiff2ps	1,093	1,098	2,316	2,692
tiffcrop	1,770	1,764	4,286	4,940
xmllint	1,894	1,964	2,731	3,366
Mean decrease			39.17 %	49.10%
Median decrease			49.18 %	58.20%

The results of our ablation study are presented in Table 8. FOX achieves an average of 20.07% more coverage than FOX-BASE and 9.23% more coverage than FOX-SCHED. These results explicitly demonstrate the importance and contributions of both FOX scheduler and mutator in enhancing the overall performance. Notably, the frontier-branch-based scheduling alone results in an almost universal performance improvement of up to 27.2% in mean coverage on `pdftotext` over the baseline, with the sole exception being `xmllint`. We attribute this anomaly to the fact that `xmllint` seems to have an unusually large control space where the number of frontier branches exceeds the number of seeds in the first hour. This trend is visualized in Figure 3, comparing the control space of FOX against AFL++. We plan to further investigate this effect in the future. Similarly, enabling the Newton's method mutator yields a nearly unanimous gain of up to 31.9% in mean coverage, with only a single exception of `tiff2ps`. We found that `tiff2ps` has a large proportion of non-convex branches. We discuss this phenomenon in greater depth in §4.5. Improving our mutator for non-convex branches is an important direction that we plan to pursue in future work.

Result 4: Frontier-branch-based scheduling and Newton-step-based mutator both contribute to the 20.07% edge coverage improvement of FOX over the baseline fuzzer in the first hour.

Table 8: Mean edge coverage of FOX against its two variants on 15 standalone programs over 10 fuzzing campaigns run for 1 hour each. We mark the highest number in bold.

Targets	FOX	FOX-BASE	FOX-SCHED
bsdtar	**3,952**	2,382	2,997
exiv2	**6,606**	6,152	6,291
ffmpeg	**17,316**	16,291	17,272
jasper	**3,153**	2,787	3,007
nm-new	**3,050**	2,249	2,449
objdump	**4,996**	3,672	4,017
pdftotext	**2,567**	1,905	2,423
readelf	**3,235**	2,846	3,152
size	**2,621**	2,131	2,351
strip-new	**5,476**	4,289	4,715
tcpdump	**5,875**	4,656	5,523
tiff2pdf	**6,124**	5,754	5,950
tiff2ps	4,373	4,202	**4,458**
tiffcrop	**6,995**	6,626	6,936
xmllint	4,220	**4,429**	4,090
Mean gain		20.07%	9.23%

Table 9: Frontier branch flipping comparison between FOX, AFLPP+C, and AFLPP+CD. R stands for frontier branches reached, and F stands for frontier branches flipped. We mark the highest number in bold.

Targets	FOX		AFLPP		AFLPP+C		FOX+D		AFLPP+CD	
	R	F	R	F	R	F	R	F	R	F
bsdtar	**2,873**	**1,694**	1,714	776	1,837	890	**3,083**	**1,854**	3,017	1,723
exiv2	3,929	948	3,718	886	**4,477**	**1,088**	**5,664**	**1,597**	5,617	1,482
ffmpeg	**12,839**	**7,800**	10,219	6,093	8,922	4,478	**13,993**	**7,766**	8,908	3,921
jasper	**1,718**	624	1,492	501	1,704	**631**	**1,759**	**638**	1,696	624
nm-new	**2,127**	**1,153**	1,591	802	1,964	1,056	**2,213**	**1,158**	1,840	947
objdump	**2,746**	**1,368**	2,097	1,010	2,495	1,240	**2,817**	**1,422**	2,536	1,267
pdftotext	**1,425**	**439**	1,146	297	1,361	**439**	**1,623**	**574**	1,342	444
readelf	**1,475**	**1,052**	1,274	874	1,191	770	**1,706**	**1,249**	1,328	893
size	**1,459**	**771**	1,228	619	1,281	671	**1,572**	**850**	1,303	671
strip-new	**2,974**	**1,526**	2,421	1,212	2,615	1,381	**3,238**	**1,633**	2,625	1,351
tcpdump	6,601	**3,578**	5,196	2,670	**6,789**	3,496	6,816	**3,784**	**6,850**	3,592
tiff2pdf	**3,316**	**1,575**	3,211	1,483	3,227	1,487	**3,326**	**1,572**	3,230	1,502
tiff2ps	**2,155**	**1,098**	2,051	997	2,115	1,041	2,078	1,043	**2,133**	**1,058**
tiffcrop	**3,695**	**1,887**	3,590	1,784	3,642	1,812	**3,769**	**1,942**	3,611	1,784
xmllint	**3,502**	**1,489**	2,322	836	2,409	898	**3,894**	**1,750**	3,723	1,623
Mean gain			23.32%	33.99%	14.16%	21.05%	–	–	13.17%	19.47%

4.5 RQ5: FOX Performance Introspection

In **RQ5**, we delve into the sources of performance improvement of FOX and leverage the fine-grained information it provides about frontier branches to guide FOX mutator enhancements. Initially, we

analyze how successful FOX is at reaching and flipping branches compared to other state-of-the-art fuzzers. Subsequently, we assess the relationship between branch convexity and FOX's ability to flip branches.

Reaching and Flipping Branches. Distinguishing itself from traditional coverage-guided fuzzers, FOX focuses on targeting and flipping frontier branches to boost coverage. We expected FOX to not only reach but also flip more branches than its competitors. In 10x24-hour fuzzing campaigns, we compared the branch-flipping abilities of FOX and FOX+D with AFL++, AFLPP+C, and AFLPP+CD (Table 9). The results show that both FOX and FOX+D reach and successfully flip more frontier branches, with FOX flipping 33.99% more than AFL++ and up to 21.05% more than AFLPP+C, while FOX+D surpasses AFLPP+CD by 19.47%. This underscores FOX's effective framework in achieving new coverage by handling different types of frontier branches.

Branch Convexity Estimation. As FOX's Newton's method mutator is primarily optimized for convex functions [18], we test FOX on 15 standalone programs over 5x24-hour campaigns to evaluate its performance on both convex and non-convex branches. Identifying branch convexity is challenging, so we used midpoint convexity checks during fuzzing as an estimate. We consider two inputs to be midpoint convex if the branch distance of the average of the two inputs is less than or equal to the average of the branch distances of the two inputs.

The experiment involved tracking midpoint convexity before and after applying Newton's method. For each branch, we calculated a ratio of successful midpoint checks to branch reaches, indicating convexity. Higher ratios suggest convex behavior, while lower ratios indicate non-convexity. We expected FOX to excel in flipping convex branches. This was analyzed using logistic regression, modeling branch flips against the convexity estimate and its interaction with the binary.

The regression (details in extended version of this work [50]) showed a positive correlation between branch convexity and FOX's flipping success, with a McFadden's pseudo-R-squared of 0.048, indicating a good fit [23, 30]. The correlation between branch convexity and flipping success varied among programs, which was anticipated. Although midpoint convexity is a useful estimate, it does not always precisely represent a branch's true convexity and feasibility. For instance, non-convex branches might pass the convexity check in certain regions, whereas some convex branches could be infeasible, i.e., impossible to flip.

Result 5: FOX can flip up to 33.99% more frontier branches, showcasing the efficacy of its stochastic control-guided framework.

5 Limitations and Future Work

In this section, we discuss the theoretical and engineering limitations of the current design of FOX, and outline potential future directions for improvement.

Newton's Method. FOX utilizes Newton's method to generate inputs that decrease branch distances and potentially flip frontier branches. The main advantage of this approach is its simplicity and efficiency — it only needs two distinct inputs that reach the target branch to estimate a Newton step. However, Newton's method struggles with non-convex functions like checksums, hash functions, and complex string operations. We aim to enhance our approach by selectively applying more accurate, albeit computationally intensive methods for hard-to-flip non-convex branches.

Local Search. The local search module of FOX establishes a local linear approximation (w.r.t. a seed) of branch distance functions, utilized by Newton's method for generating new inputs. Despite the accuracy demonstrated by methods like REDQUEEN [13], their complexity and inefficiency make them less practical compared to local search, as evidenced in our experimental findings. Currently, local search utilizes AFL++ `havoc` for mutations, which has difficulty in adaptively controlling the extent and scale of mutations, resulting in inefficient and redundant mutations. In the future, we aim to enhance FOX by integrating a sophisticated mutator capable of dynamically adjusting its mutation strategy based on past successes in reaching the frontier branches.

Frontier Branch Scheduler. Our scheduler selects seeds associated with promising frontier branches, refining the control space more effectively than traditional seed-based methods. However, the scheduler depends on estimates of the likelihood of reducing branch distance to identify promising branches. Despite the theoretical optimality demonstrated in §A.1, its practical efficacy relies on the accuracy of these estimates. In fact, we have identified several pathological branches within target programs where the mutator consistently decreases branch distances but fails to flip them throughout the entire fuzzing campaign. We speculate that such behavior suggests the infeasibility of these branches. Therefore, we plan to modify the scheduler in the future to deploy an aggressive re-weighting strategy to detect and deprioritize such pathological branches.

Engineering Limitations. The current prototype of FOX is optimized for integer and string comparisons, which are prevalent in target applications, as shown by our coverage (§4.1) and bug discovery (§4.2) performance. However, our prototype currently does not support branches with floating point operations or variable-variable string comparisons, which appear in benchmarks like `lcms` and `proj4`. This limitation occasionally results in lower coverage compared to leading fuzzers. Future updates will aim to expand FOX's applicability to these branch types and involve the community in enhancing the tool through open-source contributions.

6 Related Work

Search-based Software Testing. Such approaches use metaheuristic optimization for tasks like test-case generation [25, 34, 42]. Szekeres et al. [53] suggested using stochastic local search and taint tracking for targeted fuzzing of specific branches to minimize the distance to a target branch. However, FOX differs by using local search to estimate the gradient of the branch distance function, employing Newton's method for more effective root finding. There are also hybrid methods that integrate path constraint information into their search strategy [19, 22, 27, 52], but these face scaling challenges in large programs. FOX avoids these challenges by focusing on frontier branches and using branch distance as a metric to create inputs that flip these branches.

Mutator Policies. Several prior studies have aimed to optimize mutation policies for fuzzing [13, 20, 35, 40, 48]. REDQUEEN [13] employed input colorization with strong assumptions about input consistency, contrasted by FOX's flexible mutations. Angora [20] used gradient descent and taint tracking, while FOX identifies input bytes using local search and Newton's method. Neuzz [48] adopted a neural network approach for branch behavior, whereas FOX constructs a simpler model using fewer samples. Lastly, FairFuzz [36] focused on rare branches, whereas FOX uses a branch distance metric for targeted branch-flipping inputs.

Scheduler Policies. K-Scheduler [49] uses a centrality-based seed scheduling approach that might overestimate the potential of seeds to discover new edges, possibly including many unreachable nodes in the entire CFG. In contrast, FOX uses data solely from frontier branches for scheduling and mutation, leading to more accurate decisions.

Feedback Metrics. Several studies use detailed feedback like path coverage for seed selection [16, 56], which can lead to seed explosion [54]. FOX uses frontier branch granularity and branch distances to manage this issue. Other research proposes different feedback metrics, like data-flow coverage [26, 31] or objectives like reaching sanitizer instrumentation [21, 47], to enhance bug discovery by exploring various aspects of program state space. In the future, we aim to investigate how FOX can be combined with these diverse fuzzing objectives.

7 Conclusion

In conclusion, this paper presents a unified framework for coverage-guided mutation-based fuzzing by treating fuzzing as an online stochastic control problem. Our extensive experimental results demonstrate that our proof-of-concept implementation FOX can address many challenges of coverage-guided fuzzing for large and complex programs.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their constructive and valuable feedback. We would also like to thank László Szekeres, Jonathan Metzman, Franjo Ivancic, Xinhao Yuan, Junfeng Yang, and Baishakhi Ray for their helpful comments and discussions that improved the paper. This work is supported partially by an NSF CAREER award, a Google Faculty Fellowship, and an award from the Google Cyber NYC Institutional program. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not reflect those of NSF or Google.

References

[1] 2023. AFL++ Seed Bank. https://github.com/AFLplusplus/AFLplusplus/tree/stable/testcases.
[2] 2023. Evaluation report of aflpp on the Fuzzbench dataset. (Dec 1 2023). https://www.fuzzbench.com/reports/experimental/2023-12-01-aflpp/index.html.
[3] 2023. Fuzz introspector. https://github.com/ossf/fuzz-introspector.
[4] 2023. FuzzBench AFL++ setup. https://github.com/google/fuzzbench/blob/master/fuzzers/aflplusplus/description.md.
[5] 2023. Havoc mode mutation. https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/afl-fuzz_approach.md.
[6] 2023. Magma lua bugs. https://github.com/HexHive/magma/tree/v1.2/targets/lua/patches/bugs.
[7] 2023. Magma openssl bugs. https://github.com/HexHive/magma/issues/68.
[8] 2023. SBST'23 Fuzzing Competition (C/C++ Programs). https://sbft23.github.io/tools/fuzzing.
[9] 2023. SBST'23 Fuzzing Competition (C/C++ Programs) Report. https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Coverage/index.html.
[10] 2023. Syzbot kernel bug tracker. https://syzkaller.appspot.com/upstream.
[11] 2023. Syzbot kernel bug tracker. https://github.com/google/oss-fuzz.
[12] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1985793.1985795
[13] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*.
[14] Dario Asprone, Jonathan Metzman, Abhishek Arya, Giovani Guizzo, and Federica Sarro. 2022. Comparing Fuzzers on a Level Playing Field with FuzzBench. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 302–311. https://doi.org/10.1109/ICST53961.2022.00039
[15] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. 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*. 678–689.
[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[17] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*. 1621–1633.
[18] Stephen Boyd and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press.
[19] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.
[20] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[21] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.
[22] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 736–747.
[23] Thomas A. Domencich and Daniel McFadden. 1975. *Urban Travel Demand: A Behavioral Analysis : a Charles River Associates Research Study*. North-Holland Publishing Company. Google-Books-ID: KwK3AAAAIAAJ.
[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[25] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
[26] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *29th USENIX security symposium (USENIX Security 20)*. 2577–2594.
[27] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.
[28] Elad Hazan et al. 2016. Introduction to online convex optimization. *Foundations and Trends® in Optimization* (2016).
[29] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (nov 2020), 29 pages. https://doi.org/10.1145/3428334
[30] D.A. Hensher and P.R. Stopher. 2021. *Behavioural Travel Modelling*. Taylor & Francis. https://books.google.com/books?id=Z_UlEAAAQBAJ
[31] Adrian Herrera, Mathias Payer, and Antony L Hosking. 2022. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Transactions on Software Engineering and Methodology* (2022).
[32] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2023. DARWIN: Survival of the Fittest Fuzzing Mutators. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. https://doi.org/10.14722/ndss.2023.23159
[33] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
[34] Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2010. AUSTIN: A tool for search based software testing for the C language and its evaluation on deployed automotive systems. In *2nd International symposium on search based software engineering*. IEEE, 101–110.
[35] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. 2023. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE.

[36] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.

[37] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1–17. https://doi.org/10.1109/SP46214.2022.9833594

[38] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.

[39] Dongge Liu, Jonathan Metzman, Marcel Böhme, Oliver Chang, and Abhishek Arya. 2023. SBFT Tool Competition 2023–Fuzzing Track. *arXiv preprint arXiv:2304.10070* (2023).

[40] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*.

[41] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 286–302.

[42] Phil McMinn. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 153–163.

[43] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*.

[44] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. *ArXiv* abs/1504.04909 (2015).

[45] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[46] Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. 2023. Revisiting Neural Program Smoothing for Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/3611643.3616308

[47] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.

[48] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[49] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. *2022 IEEE Symposium on Security and Privacy (SP)* (2022), 2194–2211.

[50] Dongdong She, Adam Storek, Yuchong Xie, Seoyoung Kweon, Prashast Srivastava, and Suman Jana. 2024. FOX: Coverage-guided Fuzzing as Online Stochastic Control. *arXiv* (2024). https://arxiv.org/abs/2406.04517

[51] Ji Shi, Zhun Wang, Zhiyao Feng, Yang Lan, Shisong Qin, Wei You, Wei Zou, Mathias Payer, and Chao Zhang. 2023. AIFORE: Smart Fuzzing Based on Automatic Input Format Reverse Engineering. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4967–4984. https://www.usenix.org/conference/usenixsecurity23/presentation/shi-ji

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

[53] László Szekeres and R Sekar. [n. d.]. Search-based Fuzzing. ([n. d.]). http://seclab.cs.stonybrook.edu/lszekeres/Papers/Szekeres-2017-SBF.pdf

[54] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. 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)*.

[55] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[56] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. 2307–2324.

[57] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1343–1360. https://www.usenix.org/conference/usenixsecurity23/presentation/zheng

A Theory Addendum

A.1 Proof of Theorem 1

Proof. Let expected new coverage gain given seed list S_i and chosen seed index u_i be defined as:

$$\sum_{b \in B_i} \Pr(b \text{ flips} \mid mut(S_i[u_i])) \tag{14}$$

We can transform Equation 14 to sum over the set of all branches B instead of the set of not-flipped branches at stage i B_i:

$$\sum_{b \in B} \Pr(b \text{ not flipped prior } i) \Pr(b \text{ flips} \quad | \quad mut(S_i[u_i])) \tag{15}$$

Now consider an optimal schedule $\pi = [u_1, u_2, ..., u_n]$. Given π, we can obtain another assignment π' from π by exchanging u_1 with a u_1' such that u_1' is the first greedy choice:

$$\sum_{b \in B} \Pr(b \text{ flips} \mid mut(S_1[u_1])) \leq \sum_{b \in B} \Pr(b \text{ flips} \mid mut(S_1[u_1'])) \tag{16}$$

To simplify the equations below, let:

$$\begin{aligned} p &= \Pr(b \text{ flips} \mid mut(S_1[u_1])) \\ p' &= \Pr(b \text{ flips} \mid mut(S_1[u_1'])) \\ q &= \max_i \Pr(b \text{ flips} \mid mut(S_i[u_i])) \end{aligned} \tag{17}$$

The total expected coverage gain for u_1 is then:

$$\sum_{b \in B} \left[p + (1-p)(q + (1-q)q + ... + (1-q)^{n-2}q)\right] \tag{18}$$

and analogously for u_1':

$$\sum_{b \in B} \left[p' + (1-p')(q + (1-q)q + ... + (1-q)^{n-2}q)\right] \tag{19}$$

Subtracting Equation 18 from Equation 19, we get:

$$\sum_{b \in B} \left[(p'-p)(1 - (q + (1-q)q + ... + (1-q)^{n-2}q))\right] \tag{20}$$

Note that the term involving q is in fact a CDF of the geometric distribution. We can therefore simplify further:

$$\sum_{b \in B} \left[(p'-p)(1 - (1 - (1-q)^{n-1}))\right] \tag{21}$$

$$\sum_{b \in B} \left[(p'-p)(1-q)^{n-1}\right] \tag{22}$$

Since q is a probability, therefore $0 \leq q \leq 1$, the equation above can be bounded as follows:

$$0 \leq \sum_{b \in B} \left[(p'-p)(1-q)^{n-1}\right] \leq \sum_{b \in B} (p'-p) \tag{23}$$

Note that $p' - p \geq 0$ since u_1' is the first greedy choice. Therefore, the total expected coverage gain of u_1' is at least as high as the total expected coverage gain of the optimal assignment u_1. This argument exactly applies to the assignment $u_2, u_3, ..., u_n$. Therefore, the greedy scheduling strategy is the optimal scheduling strategy for fuzzing as online stochastic control. □