

# MC<sup>2</sup>: Rigorous and Efficient Directed Greybox Fuzzing

Abhishek Shah  
Columbia University

Dongdong She  
Columbia University

Samanway Sadhu  
Columbia University

Krish Singal  
Columbia University

Peter Coffman  
Columbia University

Suman Jana  
Columbia University

## ABSTRACT

Directed greybox fuzzing is a popular technique for targeted software testing that seeks to find inputs that reach a set of target sites in a program. Most existing directed greybox fuzzers do not provide any theoretical analysis of their performance or optimality.

In this paper, we introduce a complexity-theoretic framework to pose directed greybox fuzzing as an oracle-guided search problem where some feedback about the input space (e.g., how close an input is to the target sites) is received by querying an oracle. Our framework assumes that each oracle query can return arbitrary content with a large but constant amount of information. Therefore, we use the number of oracle queries required by a fuzzing algorithm to find a target-reaching input as the performance metric. Using our framework, we design a randomized directed greybox fuzzing algorithm that makes a logarithmic (wrt. the number of all possible inputs) number of queries in expectation to find a target-reaching input. We further prove that the number of oracle queries required by our algorithm is optimal, i.e., no fuzzing algorithm can improve (i.e., minimize) the query count by more than a constant factor.

We implement our approach in MC<sup>2</sup> and outperform state-of-the-art directed greybox fuzzers on challenging benchmarks (Magma and Fuzzer Test Suite) by up to two orders of magnitude (i.e., 134×) on average. MC<sup>2</sup> also found 15 previously undiscovered bugs that other state-of-the-art directed greybox fuzzers failed to find.

## CCS CONCEPTS

• Security and privacy → Software and application security.

## KEYWORDS

Greybox Fuzzing; Automated Vulnerability Detection; Noisy Binary Search; Monte Carlo Counting; Execution Complexity

### ACM Reference Format:

Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC<sup>2</sup>: Rigorous and Efficient Directed Greybox Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, November 7–11, 2022, Los Angeles, CA, USA. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3548606.3560648>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9450-5/22/11...\$15.00

<https://doi.org/10.1145/3548606.3560648>

## 1 INTRODUCTION

Directed greybox fuzzing is a popular technique for targeted software testing with many security applications such as bug finding, crash reproduction, checking static analyzer reports, and patch testing [13, 16, 30, 36, 66, 70]. Given a set of target sites in a program, directed greybox fuzzers automatically search a program's input space for inputs that reach the targets. Since the input spaces of real-world programs are very large, most existing directed greybox fuzzers use evolutionary algorithms to focus their search on promising input regions identified using feedback information through instrumented program execution. For example, the fuzzers often collect feedback information about control-flow graph distance or branch constraint distance from the target and prioritize mutating inputs that are close to the target [13, 16, 36].

The designs of existing evolutionary directed fuzzers in the literature are typically guided by intuition and empirical results, but, to the best of our knowledge, they do not provide any theoretical analysis of their performance. While strong empirical results are a necessary metric for evaluating any fuzzer design, without a rigorous theoretical understanding, it is difficult to understand the key guiding principles of fuzzer design. For example, what is the best possible (i.e., optimal) directed fuzzer? How well does a fuzzing algorithm scale with the input space size? What kind of feedback information is most useful for fuzzing? How can an algorithm best use the feedback? Can one do better than evolutionary algorithms in using this feedback information?

In this paper, we introduce a novel computational complexity-theoretic framework to answer some of these questions and design an asymptotically optimal directed greybox fuzzing algorithm. We further demonstrate the practical advantages of our algorithm with extensive empirical results where our algorithm is up to two orders-of-magnitude faster, on average, than the state-of-the-art directed greybox fuzzers in challenging benchmarks (Magma and Fuzzer Test Suite).

**Complexity-Theoretic Framework.** To reason about an optimal fuzzer, we introduce a complexity-theoretic formulation of directed greybox fuzzing that abstracts away the specific details about the type of instrumentation and fuzzing algorithm into a unified framework. We model the task of directed greybox fuzzing as an oracle-guided search problem: to find inputs that reach the target given query access to an oracle that executes the program and reveals some information about the search space (i.e., the identity of the promising input regions) to the fuzzing algorithm.

Our framework makes no assumptions about program behaviors or input/output distributions and faithfully adheres to the design of modern greybox fuzzing. To model the lightweight fuzzing instrumentation used by practical fuzzers for collecting feedback

information during a program execution, we allow the oracle to return arbitrary content with a large but constant amount of information per query. Formally, we allow the oracle to be any function  $O : I \rightarrow \{0, 1\}^c$  that internally executes the instrumented target program on a constant number of inputs from an input region  $I$  and returns  $c$  bits of information (see Section 2.2 for more details). We model the adaptive, feedback-driven paradigm used in practical fuzzers by enabling the fuzzer to arbitrarily adapt its choice of executions/input regions provided to the oracle based on the information received from the oracle in prior queries.

**Execution Complexity.** We then introduce the notion of execution complexity, a metric to asymptotically measure the performance of any fuzzing algorithm in our framework in terms of the number of oracle queries made by the fuzzer before finding an input that reaches the target site. This metric is a very good fit for our setting as program execution and feedback generation dominate the runtime of real-world fuzzers [42]. At a conceptual level, our complexity metric is similar to the query complexity used to reason about the lower bounds on potential advantages provided by quantum algorithms by separating the quantum part of the algorithm from the classical part through an oracle [5, 6].

In our case, we want to establish lower bounds on the advantages provided by information feedback to the fuzzing algorithms and measure the corresponding execution (i.e., query) complexity. It has been shown that any search in an input space of size  $N$  with a boolean oracle that only indicates whether a given test input is the desired one or not (e.g., a blackbox fuzzer) will take at least  $O(N)$  queries [51]. In this paper, we use our framework to explore the lower bounds on the advantages of extra feedback information (i.e., large but constant per oracle query) and how to design an adaptive algorithm that can best exploit such feedback.

**An Optimal Fuzzing Algorithm.** For designing a concrete fuzzing algorithm, we further introduce a special type of oracle called the noisy counting oracle. A noisy counting oracle takes two arbitrary input regions, approximately counts the number of inputs in each region reaching the target sites, and returns the one with higher count as the more promising one. Due to the approximate nature of the counts, we assume that the noisy oracle returns the incorrect answer with probability  $p < 1/2$ . Later in this section, we describe how we design an approximate Monte Carlo counting algorithm to implement the noisy oracle in practice.

Given access to such a noisy oracle, we design a randomized algorithm for directed greybox fuzzing that achieves the optimal expected execution complexity  $O\left(\frac{\log(N)}{(\frac{1}{2}-p)^2}\right)$ , up to constant factors, in an input space with  $N$  inputs. We further prove that this execution complexity cannot be improved (beyond a constant factor) by any other fuzzing algorithm (evolutionary or otherwise) for any given  $p < 1/2$ .

As we detail in Section 2.3, our fuzzing algorithm, at its core, uses a counting oracle to select regions with higher counts and repeatedly narrows down promising regions with binary search. If our counting oracle was noiseless, a simple binary search based fuzzing algorithm will efficiently find target-reaching inputs. However, the noise in the counting oracle can cause the fuzzer to sometimes incorrectly select input regions with low counts. As we do

not know which input regions have larger counts with full certainty, our algorithm must be robust to such noise. In this paper, we build on the noisy binary search approach introduced by Ben-Or et al. [10]. To deal with the noise, the randomized noisy binary search algorithms [10, 32] maintain a set of weights for each region representing the belief of the algorithm about how likely the desired input is in that region. The algorithm iteratively increases the weights in promising regions based on each oracle query and prioritizes narrowing down these promising input regions.

**Approximate Counting with Monte Carlo.** We develop a Monte Carlo algorithm for implementing the noisy counting oracle that is needed by our randomized directed greybox fuzzing algorithm. Monte Carlo random sampling is one of the popular approaches to approximate counts [15, 33]. For example, to approximately count the number of people in the world who like ice cream, Monte Carlo methods will poll a random subset of individuals if they like ice cream and multiply the world’s population by the ratio of people who liked ice cream in the poll to the total number of participants. The more people one polls, the more accurate the count is.

However, applying such techniques naively in our setting can result in most of the approximate counts being zero even though their true values might be small non-zero numbers. Consider a target program with multiple branch constraints guarding the target. Suppose we wish to count the number of inputs that reach the target in an input region with 1 million inputs and the true count of inputs reaching the target, unknown to us, is 10. To efficiently approximate this count, we might execute the target program on a small number of inputs selected uniformly at random from the input region and multiply the size of the input region by the ratio of the number of inputs that reach the target in our executions to the total number of tested inputs. The challenge, however, is that unless we generate a prohibitively large number of inputs  $\sim 10^5$ , we will estimate the count as zero because we are unlikely to find the few target-reaching inputs with a small number of randomly selected inputs.

Estimating the count of inputs reaching the target as zero can be highly detrimental to the fuzzing algorithm. In practice, most fuzzing target sites are only reachable by a small number of inputs satisfying one or multiple branch constraints. Estimating the corresponding count for any large input region as zero will degrade our fuzzing algorithm’s performance significantly because it will fail to identify input regions with larger counts. To overcome this problem, we observe that even if we did not find the few inputs that reach the target, we can still compute an upper bound on the count with high confidence.

**Concentration Bounds.** We compute such probabilistic bounds by using concentration bounds [22], a well-studied technique to upper-bound the probability of a random variable taking a value within a given range based on the variable’s mean and variance. The intuition behind such bounds is that the closer the mean (with small variance) is to the range, the higher the likelihood that the random variable will take a value within the range. For example, if we model the branch distance [38] of a branch constraint as a random variable and compute the mean and variance of the values observed at the branch during the program executions with the randomly selected inputs, we can use the concentration bounds to

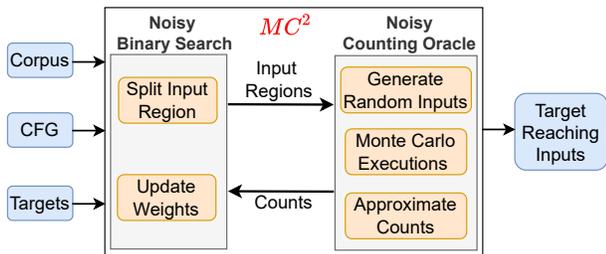


Figure 1: Workflow of  $MC^2$

derive an upper bound on the likelihood of satisfying the branch constraint.

One issue with these concentration bounds is that they might incur large over-approximation error that can increase the noise in the oracle, hindering the performance of the fuzzing algorithm. Therefore, we use the concentration bounds sparingly. Specifically, during a single oracle query with a set of randomly selected inputs, we first observe which branches are satisfied for which inputs and apply probabilistic upper bounds only for the branches that have never been satisfied for any input. For the rest, we use the empirically observed non-zero ratio of the number of inputs satisfying a branch to the total number of tested inputs.

**Counting along Multiple Branches.** For any given path reaching a target, we approximately count the number of inputs that satisfy each branch in the path as described above and combine them to get an estimate on the count of inputs that reach the target (See Section 2.4). To efficiently approximate counts for nested branches, we further introduce Monte Carlo Execution, a lightweight form of program execution that modifies control-flows at runtime to ensure nested inner branches will be visited during the program execution even if the input did not satisfy the outer branch constraints. This technique enables us to collect information about any branch and subsequently efficiently approximate their counts with a small number of program executions. We note that even though our counting algorithm cannot provide guaranteed error bounds for arbitrary programs, our experimental results demonstrate that the method is highly effective on real-world programs.

We implement our approach in  $MC^2$  (Monte Carlo Counting) shown in Figure 1 and evaluate  $MC^2$  against state-of-the-art directed greybox fuzzers on challenging benchmarks with real-world programs. Our results are very promising. In the Magma benchmark,  $MC^2$  finds bugs faster by 134x, on average, and finds 16 more bugs than the next-best fuzzer. In the Fuzzer Test Suite benchmark,  $MC^2$  reaches targets 77x faster, on average, and reaches 2 more targets than the next-best fuzzer. In addition,  $MC^2$  found 49 previously undiscovered bugs in real-world programs, 15 more than the next-best fuzzer. We release an open source version of  $MC^2$  at <https://hub.docker.com/r/abhishekshah212/mc2>.

Our contributions are summarized as follows.

- We introduce a complexity-theoretic framework for defining directed greybox fuzzing as an oracle-guided search problem and introduce execution complexity, a metric, to measure a fuzzing algorithm’s asymptotic performance.

- We design an asymptotically optimal randomized directed greybox fuzzer that has logarithmic expected execution complexity in the number of possible inputs. We also show that this expected execution complexity cannot be improved, up to constant terms, by any fuzzer.
- We develop a Monte Carlo algorithm for implementing a noisy counting oracle that can work efficiently together with our fuzzing algorithm.
- We implement our technique in  $MC^2$  and show its promise, outperforming existing state-of-the-art greybox fuzzers in challenging benchmarks (Magma and Fuzzer Test Suite) by up to two orders-of-magnitude.

## 2 METHODOLOGY

We first introduce a generic complexity-theoretic framework for reasoning about the best possible directed greybox fuzzer in terms of an oracle-guided search problem. We next instantiate a specific type of oracle called the noisy counting oracle and use it to build an optimal directed greybox fuzzer using noisy binary search. We then show how to implement a noisy counting oracle in practice.

### 2.1 Terminology and Notation

Below, we provide a summary of the terminology used throughout this paper. In this paper we use the word fuzzer to refer to a directed greybox fuzzer unless otherwise noted. We denote the target program as  $P$  and its large but finite-sized input space to be explored during fuzzing as  $\mathbb{I}$ . We denote the count of elements in a set with the cardinality  $|\cdot|$  symbol and the size of the input space as  $N = |\mathbb{I}|$ .

**Input Region.** We refer to any subset  $I \subseteq \mathbb{I}$  as input region  $I$ . Since programs being fuzzed generally accept inputs as a sequence of bytes, we express, without loss of generality, the target program’s input space as all possible combinations of byte values in the form of a hyperrectangle:  $\mathbb{I} = [0, 255]^d$  where  $d$  indicates the number of input bytes. Although  $N = |\mathbb{I}|$  is finite, it is exponentially large in  $d$ , the number of inputs bytes to be fuzzed, which is generally bounded in real-world fuzzers for performance reasons [1, 3].

**Control Flow Graph.** In the greybox setting, we have access to a target program’s control flow graph:  $CFG = (V, E)$ , where the set of vertices  $V$  represents basic blocks and the set of edges  $E$  represents control-flow transitions (e.g., branches). We define a path  $\pi$  as a finite sequence of edges in the CFG  $\pi : E_0 \rightarrow E_1 \rightarrow \dots \rightarrow E_k$ , in which any two consecutive edges are adjacent.

Since we only care about paths that start from the program entry point (e.g., main), we denote the set of paths in the CFG that start from a program entry basic block and terminate at a program exit basic block as  $\Pi$ . As directed fuzzers are interested in reaching some target  $E_T \in E$ , we say that a path  $\pi \in \Pi$  reaches a target if  $E_T \in \pi$  and denote the subset of paths that reach target  $E_T$  as  $\Pi_T$ .

**Program Execution.** We denote executing a program  $P$  on input  $i$  as  $P(i)$  and its corresponding path through the CFG as  $\pi_{P(i)}$ . We also say an input  $i$  reaches the target  $E_T$  if its path  $\pi_{P(i)}$  reaches the target  $E_T$ .

**Branch Constraint.** Each edge in our CFG corresponds to a conditional branch constraint:  $c : \mathbb{I} \rightarrow \{0, 1\}$  that takes the following normalized form:

Constraint	$c(i)$	$:=$	$d(i) \bowtie \emptyset$
Predicate	$\bowtie$	$:=$	$\{=, <, <=, >, >=\}$
Input	$i$	$:=$	$[i_1, i_2, \dots, i_d]$

The branch distance [38]  $d(i)$  captures the effect of all program instructions preceding the branch constraint during a program execution on input  $i$ .

## 2.2 A Framework for Directed Greybox Fuzzing

While empirical measures can compare the performances of fuzzers with different designs, such measurements alone cannot tell us how close the designs are to the best possible (i.e., optimal) fuzzer. Towards this end, in this section, we introduce a complexity theoretic framework to reason about the lower bounds on the performance of an optimal fuzzer in terms of the number of target program executions. We design our framework to faithfully capture the characteristics of modern greybox fuzzers: collecting feedback information about a program execution through lightweight instrumentation and adapting their algorithms based on such feedback.

**Fuzzing as Oracle-Guided Search.** Our framework allows the fuzzer to learn information about any bounded input region  $I$  by querying an oracle. Formally, the oracle  $O : I \rightarrow \{0, 1\}^c$  is any function that internally executes the program  $P$  on some pre-determined constant number of inputs  $i \in I$ , since brute-forcing all inputs is not practical, and returns arbitrary content with a large but constant amount,  $c$  bits, of information. In this context, constant means independent of input size, as is customary in complexity-theoretic analysis.

Specifically, we assume that each oracle query can provide at most  $c$  bits of information about a given input region either covering the entirety of the input space or some parts of it, where  $c$  is some constant, capturing the information collected from a fuzzer’s lightweight instrumentation. As we later demonstrate in Section 2.3, a fuzzer can potentially use these  $c$  bits to reduce the number of inputs it considers by a multiplicative factor of  $\frac{1}{2^c}$  (e.g.,  $c = 1$  bit cuts input space in half), which captures modern fuzzer’s use of feedback information to reduce the number of inputs by adapting the generated inputs towards particular seeds.

We model each oracle query as providing at most  $c$  bits of information because it captures the trade-off made by real-world fuzzers: lowering the amount of instrumentation for faster execution times. Unlike symbolic/concolic execution that capture more information by collecting all the path constraints along the execution path and invoking a Satisfiability Modulo Theory (SMT) solver repeatedly for a large number of paths, modern fuzzers use lightweight instrumentation to minimize execution overhead. Hence, the upper bound of a constant number of bits of information is a natural fit for these fuzzers.

To ensure it can reason about generic programs, our framework is distribution-free: it makes no assumptions about the target program behaviors or the types of inputs. Our framework also makes no assumptions on the type of instrumentation or the data (e.g., distance, dataflow, etc.) collected by the fuzzers as feedback. Moreover, our framework posits that fuzzers have no prior knowledge about programs and only acquire information through oracle queries, mimicking real-world fuzzing deployments that run on a large

collection of programs without pre-built knowledge of program specifics.

**Problem Definition.** We define the task of directed greybox fuzzing as an oracle-guided search problem: given access to a program  $P$ , its control-flow graph  $CFG$ , a bounded input space  $\mathbb{I}$ , a target edge  $E_T$ , and an oracle  $O : I \rightarrow \{0, 1\}^c$ , a fuzzing algorithm has to find an input  $i \in \mathbb{I}$  such that  $\pi_{P(i)}$  reaches the target  $E_T$ .

**Execution Complexity.** In this framework, to measure a fuzzer’s performance, we analyze the number of oracle queries to solve the underlying search problem. Since the number of oracle queries directly maps to the number of program executions, up to constant factors, we define the performance metric of fuzzers in our framework as *execution complexity*: the number of oracle queries needed before finding an input that reaches the target. This asymptotic performance metric is well-fit for real-world fuzzers because instrumented target program executions dominate the fuzzing performance overhead [42]. In our analysis, we ignore constant factors because they significantly depend on the hardware and we desire a hardware-independent analysis as fuzzers are run over a heterogeneous collection of hardware.

We can now reason about the best possible fuzzer with our framework. The framework provides a lower bound on the performance of any fuzzer (evolutionary or otherwise) including the best possible fuzzer, a result which we prove in Appendix A.

**THEOREM 2.1 (LOWER BOUND FOR ANY FUZZING ALGORITHM).** *Given any oracle revealing a constant  $c$  bits of information per query, any fuzzing algorithm requires  $\Omega(\log(N))$  execution complexity to find inputs that reach the target in an input space of size  $N$ .*

In the next section, we design an asymptotically optimal fuzzer.

**Greybox vs Blackbox Oracle.** We highlight that this lower bound is not achievable with a blackbox oracle, where an oracle query outputs a boolean value indicating if the target was reached or not for a given input. Such a blackbox fuzzing oracle, unlike the greybox one, does not provide  $c = 1$  bit of information as one oracle query decreases the number of inputs a fuzzer considers by only 1, rather than by a multiplicative factor of  $\frac{1}{2^1}$ . Therefore, a fuzzer using a blackbox oracle (e.g., blackbox fuzzer) has  $O(N)$  execution complexity [51] to find target-reaching inputs in an input space of size  $N$  and is asymptotically slower than a greybox fuzzer, which also matches empirical evidence [46].

## 2.3 Optimal Directed Fuzzer with Noisy Counting Oracle

In this section, we first introduce a special kind of oracle called the noisy counting oracle that identifies which region, among two arbitrary input regions, is more promising by approximately counting the number of inputs in each region that reach the targets. Given this oracle, we then design an asymptotically optimal fuzzing algorithm that we introduce in two stages. We first describe our algorithm in an idealized setting with a noiseless counting oracle and then extend our algorithm to a realistic setting with a noisy counting oracle.

**Noisy Counting Oracle.** We define a noisy counting oracle as taking two arbitrary input regions, approximately counting the

number of inputs in each region that reach the targets, and returning  $c = 1$  bit of information about which region contains more inputs reaching the targets. Due to the approximate nature of the counts, we assume that the noisy oracle returns the incorrect answer with probability  $p < 1/2$ . More formally, on input regions  $I_L$  and  $I_R$ , the counting oracle computes the following formula:

$$O(I_L, I_R) = \begin{cases} 1 & \text{if } C(I_L) \geq C(I_R) \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where  $C(I_L), C(I_R)$  denotes the count of inputs that reach the target in the left and right input region, respectively as defined below:

$$C(I) = |\{i \in I \mid \pi_{P(i)} \text{ reaches the target}\}| \quad (2)$$

**Optimal Deterministic Fuzzer.** We present a deterministic fuzzing algorithm that achieves the lower bound on execution complexity (i.e., optimal) in Theorem 2.1 using a noiseless ( $p = 0$ ) counting oracle introduced above. Our fuzzer leverages binary search and splits a given input region into two input regions to use the counting oracle. However, as input regions often span multiple bytes, splitting an input region in half is ambiguous. For example, given a 2 byte input region  $[0, 255] \times [0, 255]$  that can be visualized as a square, we can split the input region in half either vertically or horizontally.

To ensure that the splitting process is unambiguous, we assign a total order to the input space, which conceptually flattens the input space into an array of size  $N = |\mathbb{I}|$ . There are many possible such orderings, but in this paper, we use lexicographic order (unless otherwise specified) which flattens byte 1, then byte 2, and so forth. In the prior example, lexicographic order conceptually flattens the input space into the following form  $[(0, 0), (0, 1), \dots, (0, 255), (1, 0), (1, 1), \dots (255, 255)]$ .

Our fuzzing algorithm iteratively splits an input region in half and queries the counting oracle to pick the half with higher counts, eventually finding inputs that reach the target. At each iteration, this algorithm reduces the input region size by a multiplicative factor of  $\frac{1}{2}$  (i.e.,  $c = 1$  bit of information) and therefore has  $O(\log(N))$  execution complexity, achieving the lower bound in Theorem 2.1. Algorithm 4 in the Appendix illustrates this process.

**Optimal Randomized Fuzzer.** Even though the deterministic algorithm presented earlier achieves the theoretical lower bounds, it is not practical because it depends on a noiseless counting oracle that cannot be efficiently implemented in practice. Precisely, these types of counting problems belong to #P, a complexity class that is at least as hard as NP [33]. Therefore, in practice, one can only hope to implement an oracle that will approximate the underlying counts. Unfortunately, our prior binary search based deterministic fuzzer design cannot work with such approximations as any error can mislead our fuzzer’s selection of the input region with the largest count. We therefore design a new randomized fuzzing algorithm resilient to the noise from approximation error with techniques from the noisy binary search literature [10, 21, 32].

In noisy binary search [10, 21, 32], algorithms perform binary search in a setting where comparisons may be unreliable, a natural fit for us because our noisy counting oracle may not reliably compare the counts between input regions due to the approximate nature of the counts. Since there is always some source of

uncertainty, it is customary to develop randomized algorithms that succeed with high probability and require a small number of oracle queries in expectation (i.e., expected execution complexity), since the exact number may change each time the algorithm runs.

We note that the expectation considers all potential behaviors of the randomized algorithm and makes no assumption about the input distribution, which is a natural fit for fuzzing because we are trying to build a practical randomized fuzzing algorithm that works well on any program. Moreover, any randomized fuzzing algorithm with some probability of success can be re-ran multiple times, so that its probability of failure exponentially decreases with more trials to a negligibly small value. Such repeated runs can also be very easily incorporated in fuzzers that run computations repeatedly in a long-running loop.

To build our noise-resilient fuzzer, we adapt a randomized algorithm proposed in prior work by Ben-Or et al. in noisy binary search [10]. We state the expected execution complexity and optimality of our algorithm below, where we provide proofs in Appendix A.

**THEOREM 2.2 (ALGORITHM 1 EXECUTION COMPLEXITY).** *Given a noisy counting oracle that returns  $c = 1$  bit of information with failure probability  $p < \frac{1}{2}$  per query, our fuzzing algorithm has  $O((1 - \delta) * \frac{\log(N)}{(\frac{1}{2} - p)^2})$  expected execution complexity to find inputs that reach the target with success probability at least  $1 - \delta$ .*

**THEOREM 2.3 (ALGORITHM 1 OPTIMALITY).** *Given any oracle that returns a constant  $c$  bits of information with failure probability  $p < \frac{1}{2}$  per query, any fuzzing algorithm that succeeds with probability at least  $1 - \delta$  has  $\Omega((1 - \delta) * \frac{\log(N)}{(\frac{1}{2} - p)^2})$  expected execution complexity.*

We describe our fuzzer in Algorithm 1. To better understand our fuzzing algorithm’s design, we first compare it to a naive algorithm that makes multiple oracle queries at each splitting point and takes the majority result returned by those queries. Clearly, such an algorithm will be robust to a noisy oracle. However, it has  $O(\frac{\log(N) * \log(\log(N))}{(\frac{1}{2} - p)^2})$  expected execution complexity [32], which is not optimal from Theorem 2.3. By contrast, our algorithm lowers the expected execution complexity by adaptively selecting the splitting point based on each oracle query, so that there are fewer queries in total.

In more detail, our algorithm decides where to split by maintaining a set of weights which represent its belief that an input region contains the target-reaching inputs. Starting from the belief that any input region is equally likely to contain inputs that reach the target, our fuzzing algorithm iteratively increases the weights in promising regions based on each oracle query and prioritizes splitting at points within these promising input regions. We note that the weight update rule can be thought of as updating the Bayesian posterior [10] or using the multiplicative weights algorithmic paradigm [7].

We note that our algorithm has two key desirable properties. First, as the failure probability  $p$  in the noisy counting oracle increases, its performance degrades gracefully with a quadratic, not exponential relationship. Moreover, for a given failure probability  $p$ , it is optimal (i.e., cannot be improved upon) within constant factors as shown in Theorem 2.3.

---

**Algorithm 1** Optimal Randomized Fuzzer.

---

```
Input:  $\mathbb{I} \leftarrow$  Input Space as Array  
 $O \leftarrow$  Noisy Counting Oracle from Equation 1 and Algorithm 3  
1:  $G = \text{WeightGroup}()$   $\triangleright$  Initialize a single weight group: (input region, weight)  
2:  $W = [G]$   $\triangleright$  List of weight groups  
3: while  $\max(W) < \frac{1}{\sqrt{\log(|\mathbb{I}|)}}$  do  
4: /* Find group that best splits total weight in half */  
5: CumulativeWeight = 0  
6: MidIdx = 0;  
7: for group  $\in W$  do  
8: CumulativeWeight += group.weight  
9: if CumulativeWeight  $\geq 0.5$  then  
10: break  
11: MidIdx += 1  
12:  
13: /* Replace W[MidIdx] with two new groups */  
14:  $I_L, I_R = \text{Split input region in half at } W[\text{MidIdx}]$   
15:  $W[\text{MidIdx}] = (I_L, W[\text{MidIdx}].\text{weight})$   
16:  $W.\text{insert}(\text{MidIdx}+1, (I_R, W[\text{MidIdx}].\text{weight}))$   $\triangleright$  Insert group at specific index  
17:  
18: /* Perform Multiplicative Weights Update */  
19: if  $O(I_L, I_R) = 1$  then  
20: /* Left region is more promising */  
21: for idx  $\in \{0, 1, \dots, \text{MidIdx}\}$  do  
22:  $W[\text{idx}] *= (1-p)$   $\triangleright$  Increase left group weights by (1-p)  
23: for idx  $\in \{\text{MidIdx}+1, \dots, |W| - 1\}$  do  
24:  $W[\text{idx}] *= p$   $\triangleright$  Decrease right group weights by (p)  
25: else  
26: /* Right region is more promising */  
27: for idx  $\in \{0, 1, \dots, \text{MidIdx}\}$  do  
28:  $W[\text{idx}] *= p$   $\triangleright$  Decrease left group weights by (p)  
29: for idx  $\in \{\text{MidIdx}+1, \dots, |W| - 1\}$  do  
30:  $W[\text{idx}] *= (1-p)$   $\triangleright$  Increase right group weights by (1-p)  
31:  
32: /* Normalize such that sum of group weights is 1 */  
33: TotalWeight = 0  
34: for group  $\in W$  do  
35: TotalWeight += group.weight  
36: for group  $\in W$  do  
37: group.weight *=  $\frac{1}{\text{TotalWeight}}$   
38:  $G^* = \arg \max_{G \in W} (G.\text{weight})$   $\triangleright$  Select group with largest weight  
39: return random input from  $G^*$ 's input region  $\triangleright$  Inputs in  $G^*$  reach the target
```

---

Furthermore, our algorithm achieves logarithmic storage complexity  $O(\log(N))$  in terms of the size of  $\mathbb{I}$ . A naive implementation would store a distinct weight for each input and therefore require an exponential amount of space  $O(N)$ . Based on the observation that many inputs share the same weight, our algorithm groups input weights together by storing the sum of their individual weights with the corresponding input region in a weight group to avoid redundancy. Each oracle query adds one additional weight group, and since there are logarithmically many oracle queries, the algorithm uses logarithmic amount of space:  $O(\log(N))$  groups in expectation.

## 2.4 Noisy Counting Oracle through Monte Carlo Counting

In this section, we design a noisy counting oracle by approximately counting the number of inputs in a given input region that reach the target. Our oracle builds upon Monte Carlo counting, so we first introduce the intuitions behind this method and explain why directly using it fails in our setting. We next show how we exploit the graph structure in programs to decompose the count into a

summation over individual path counts. We then show how we efficiently approximate this count from individual path counts.

**Monte Carlo Counting.** Suppose we wish to predict the number of votes that a political candidate will receive in some country. Instead of asking every person in the country if they will vote for the candidate, Monte Carlo counting techniques efficiently approximate this count by polling a small number of randomly selected people and multiplying the number of people in the country by the ratio of people who liked the candidate in the poll to the total number of participants. Hence, Monte Carlo counting techniques trade-off accuracy for efficiency (i.e., the more people polled, the more accurate the count). In our context, a naive Monte Carlo counting strategy will be to execute the program on a small number of randomly selected inputs from the input region and multiply the input region size by the ratio of inputs that reached the target during our executions to the total number of tested inputs.

The challenge with such a strategy is that for most input regions, the approximate count will be zero. The main problem is that the input region size  $|I|$ , for most real-world programs, is significantly larger than the count  $C(I)$  of inputs that reach the target in the given input region, so the chance of reaching the target with a randomly selected input can be very small:  $(\frac{C(I)}{|I|} \sim 256^{-d})$ . To approximate this count effectively, the naive Monte Carlo Counting strategy will require a prohibitively large number of program executions,  $\sim \frac{1}{\frac{C(I)}{|I|}}$ .

If the counting oracle estimates the count of inputs that reach the target as zero for most input regions, our fuzzer's performance significantly degrades because it will struggle to identify the input region that contains more inputs reaching the target.

**Exploiting CFG Structure for Counting.** We observe that the graph structure of the CFG enables us to decompose the count  $C(I)$  of inputs that reach the target in an input region  $I$  into a summation of counts along any individual path  $\pi \in \Pi_T$  that reaches the target. More formally:

$$C(I) = \sum_{\pi \in \Pi_T} C_\pi(I) \quad (3)$$

where  $C_\pi(I)$  denote the count of inputs that reach the target along path  $\pi \in \Pi_T$ .

Although it is not feasible to compute this summation exactly because there can potentially be a large number of paths in a real-world program, this observation informs the design of an efficient approximation method: we can use information about how large the count is for individual paths as hints for the approximate count.

In the next section, we describe how we efficiently approximate the count of inputs that reach the target along an individual path, for a given input region. In the subsequent section, we describe how we efficiently approximate this summation by selecting the path with the largest count. We show the entire approximate counting process in Algorithm 3. Although this algorithm does not have guaranteed error bounds for arbitrary programs, our experimental results in Section 4 demonstrate that the method is effective on real-world programs.

**2.4.1 Efficiently Approximating Individual Path Counts.** In this section, we first describe how we use unconstrained counts to efficiently approximate the count of inputs that reach the target along an individual path in a given input region. We then describe two classes

of unconstraint counts that are challenging to compute and then how we address them.

**Approximating Path Counts.** We wish to compute  $C_\pi(I)$  which represents the count of inputs that reach the target  $E_T$  along path  $\pi : E_0 \rightarrow \dots \rightarrow E_T \rightarrow \dots$  in input region  $I$ . We observe the set of inputs that reach the target represents an intersection of multiple sets:  $I_{E_1} \cap I_{E_2} \cap \dots \cap I_{E_T}$ , where  $I_{E_i}$  indicates the set of inputs that satisfy only the single branch constraint at edge  $E_i$  in path  $\pi$ . The count of inputs in this intersection is strictly less than or equal to the count of inputs in any individual set  $I_{E_i}$  because intersections are subsets of individual sets. Therefore, we express the count of inputs that reach the target with the following formula:

$$C_\pi(I) = C(I_{E_1} \cap I_{E_2} \dots I_{E_T}) \leq \min(C(I_{E_1}), C(I_{E_2}), \dots, C(I_{E_T})) \quad (4)$$

where using the minimum count allows us to put an upper bound on the count of inputs along a path.

**Unconstraint Counts.** In the above equation,  $C(I_{E_i})$  represents the count of inputs that satisfy a single branch constraint at edge  $E_i$  in path  $\pi$ , so we call them *unconstraint counts*. Hence, to compute the count of inputs that reach the target along a path, we use the minimum unconstraint count.

We can efficiently approximate unconstraint counts through Monte Carlo counting because individual branch constraints are likely to have a larger count of inputs that satisfy them in contrast to an intersection of multiple branch constraints, which matches empirical evidence from the symbolic execution literature [64]. Therefore, we are more likely to approximate them effectively with a smaller number of program executions. Formally, we use the following approximation formula for unconstraint counts:

$$C(I_{E_i}) = |I| * r_{E_i} \quad (5)$$

where  $r$  is the ratio of inputs that satisfy the branch constraint at edge  $E_i$  along path  $\pi$  in our random subset to the total number of inputs selected uniformly at random with replacement from input region  $I$ .

**Challenges in Approximating Unconstraint Counts.** Even though unconstraint counts are more tractable to be approximated with the naive Monte Carlo counting strategy, there are two classes of unconstraint counts that are difficult to efficiently approximate. First, some individual branch constraints may be evaluated but not satisfied in a small number of program executions (**C1**) and second, some individual branch constraints may not be evaluated at all (e.g., nested branches) in a small number of program executions (**C2**). For these two classes of unconstraint counts, a naive strategy will approximate their counts as zero and hence the minimum unconstraint count that is used to approximate the count of inputs along a path, will be zero. We might choose to increase the number of program executions to handle them, but recall from Section 2.2 that the oracle must internally execute the program a pre-determined constant number of times to avoid brute-forcing. Since we cannot know a priori how many program executions are required to effectively approximate a unconstraint count, we describe how we address these two classes of branches below.

**C1: Handling Evaluated but Unsatisfied Branches.** Although some branches might be evaluated in a small number of executions,

**Algorithm 2** Monte Carlo Executions.

<b>Input:</b> $P$ $\leftarrow$ Program $\pi$ $\leftarrow$ Path reaching the target Inputs $\leftarrow$ Set of inputs
--

```

1: BranchDistances = HashMap() ▷ Tracks branch distances across executions
2: BranchSatisfied = HashMap() ▷ Tracks if branches satisfied
3: for  $i \in$  Inputs do
4:   for  $inst \in P(i)$  do ▷ Execute program  $P$  on input  $i$ 
5:     if  $IsBranch(inst)$  then ▷ Check branch instruction
6:        $dist = GetBranchDistance(inst)$ 
7:        $BranchDistances[inst].Add(dist)$ 
8:        $is\_satisfied = IsBranchSatisfied(inst)$  ▷ If branch satisfied 1 else 0
9:        $BranchSatisfied[inst] += is\_satisfied$ 
10:       $inst.SetBranchDirection(\pi)$  ▷ Enforce runtime control-flows follow  $\pi$ 
11:      if  $inst.RaiseException()$  then ▷ Handle program exceptions
12:        if  $inst.ReadInvalidMem()$  then
13:           $rand = GenerateRandom()$ 
14:           $inst.SetDestination(rand)$ 
15:        continue ▷ Go to next instruction
16:
17:  $ratios = []$  ▷ Compute  $r$  for each branch
18: for  $branch_i \in BranchDistances$  do
19:    $s = BranchSatisfied[branch_i]$ 
20:   if  $s > 0$  then ▷ Branch satisfied at least once
21:      $r_{E_i} = \frac{s}{|inputs|}$ 
22:   else ▷ Probabilistic upper bound
23:      $m = Mean(BranchDistances[branch_i])$ 
24:      $v = Variance(BranchDistances[branch_i])$ 
25:      $r_{E_i} = Chebyshev(m, v, branch_i.predicate)$  ▷ Use Table 1
26:    $ratios.append(r_{E_i})$  ▷ See Equation 5 for interpretation of  $r$ 
27: return  $ratios$ 

```

they might never be satisfied for any tested input, so we will approximate their unconstraint count as zero (i.e.,  $r = 0$ ). We overcome this by computing a probabilistic upper bound on these branches unconstraint counts using a concentration bound called Chebyshev's inequality [48] that sets  $r$ , the likelihood of satisfying the branch constraint, based on the sample mean and variance of observed branch distances [38] during the program executions. Such probabilistic upper bounds are a natural fit in our setting since our unconstraint counts are themselves upper bounds of the counts of all inputs taking a path.

Specifically, we model branch distance  $d(i)$  as a random variable  $X$  with mean  $\mu$  and variance  $\sigma$  to use Chebyshev's inequality. Table 1 shows Chebyshev's inequality for any form of branch constraint. It also models logical operators AND and OR as seen by the equality and inequality. Note that Chebyshev's inequality assumes nothing about the distribution except that the mean and variance are finite, which holds for programs run on finite bit-precision hardware. Therefore, it applies for any program behavior and variable type (floats, integers, etc).

Moreover, we can be confident in our probabilistic upper bounds because the approximation error of the sample mean and variance decreases exponentially fast in the number  $k$  of program executions  $\sim \frac{1}{e^k}$  for any random variable (i.e., Chernoff-Hoeffding inequality [22, 40]), and therefore, we can derive high quality approximations with a small number of executions. In addition, the quality of the approximation error and how likely it is to occur can be quantified and controlled through  $(\delta, \epsilon)$  bounds as shown in the Probably Approximately Correct (P.A.C.) framework [57]. Note that since these upper bounds can potentially result in a large over-approximation error from the true count, we use them only for branches that were never satisfied in our tested inputs. For the rest,

**Table 1: Rules for computing an upper bound on  $r$  from Equation 5. We model the branch distance  $d(i)$  as a random variable  $X$  with mean  $\mu$  and variance  $\sigma$ .  $h$  represents the smallest positive number for the data type of  $d(i)$  (i.e., for integers,  $h = 1$ ).**

Branch constraint	Rule to compute $r$
$d(i) \leq 0$	$Pr(X \leq 0) \leq \frac{\sigma}{\sigma + \mu^2}$
$d(i) < 0$	$Pr(X \leq -h) = Pr(X + h \leq 0)$
$d(i) \geq 0$	$Pr(X \geq 0) \leq \frac{\sigma}{\sigma + \mu^2}$
$d(i) > 0$	$Pr(X \geq h) = Pr(X - h \geq 0)$
$d(i) = 0$	$Pr(X \geq 0 \wedge X \leq 0) = \min(Pr(X \geq 0), Pr(X \leq 0))$
$d(i) \neq 0$	$Pr(X > 0 \vee X < 0) = Pr(X > 0) + Pr(X < 0)$

we use the empirically observed non-zero ratio of the number of inputs satisfying the branch to the total number of tested inputs.

**C2: Handling Unevaluated Nested Branches.** In our small number of program executions, some branches may not be evaluated at all because they are nested and since we have no information about such unevaluated nested branches, we will approximate their unconstraint counts as zero. Inspired by prior work in malware analysis [44, 58, 62], we instead design a new form of execution called Monte Carlo Execution that ensures that any input will visit and evaluate all nested inner branches, even if the prior outer constraints along the way are unsatisfied. Hence, in a single execution, we will visit and evaluate all branches together and therefore effectively approximate the unconstraint counts for all branches, even with a small number of program executions.

Given a path  $\pi \in \Pi_T$  consisting of a set of desired branches reaching the target  $T$ , Monte Carlo Execution modifies control-flows at runtime to always visit these branches, irrespective of the input. Note that Monte Carlo Execution does not necessarily change the original execution path: if an input satisfies all branch constraints in  $\pi$ , Monte Carlo Execution behaves as an original execution. However, it can deviate from the original execution path if the input does not satisfy any one of these branch constraints. Even if Monte Carlo Execution deviates from the original execution path for an input, the input goes through the same computation as if it was a valid input. Hence, Monte Carlo Execution always preserve the sequential ordering of computation.

To ensure that it will always visit the desired set of branches, Monte Carlo Execution must handle program exceptions. For example, the program can make an out of bounds memory access if the input controls the index of an array. We handle them by advancing the instruction pointer and if the program exception was raised by an invalid memory read, we also replace the destination with a uniformly random value to avoid bias in the computed values between individual executions.

This design can increase the set of possible values for the destination, but this is a natural fit since we use upper bounds for counts. Even though this design loses dependencies across memory reads and writes, it has low overhead in contrast to prior work that attempts to preserve these dependencies [44, 58]. In Section 4.3 and Appendix D, we run experiments to better understand this overhead. Algorithm 2 depicts the entire process of Monte Carlo Execution on a set of inputs.

**Algorithm 3** Noisy Counting Oracle.

**Input:**  $I_L \leftarrow$  Left Input Region  
 $I_R \leftarrow$  Right Input Region

```

1: Counts = HashMap()
2: for  $\pi \in \Pi^T$  do
3:   if  $\pi \notin$  PathCache then
4:      $C_\pi =$  ApproxCount( $\mathbb{I}, \pi$ ) ▶ Initialize PathCache with count over  $\mathbb{I}$ 
5:     Insert ( $C_\pi, 1$ ) into PathCache
6:   Lookup ( $C_\pi, T_\pi$ ) in PathCache
7:   Counts[ $\pi$ ] =  $C_\pi + \sqrt{\frac{\log(t)}{T_\pi}}$  ▶ Uncertainty term on  $t$ -th oracle query
8:    $\pi = \arg \max_{i \in \text{Counts}} (\text{Counts}[i])$  ▶ Select path with largest count
9:
10:  $C(I_L), C(I_R) =$  ApproxCount( $I_L, \pi$ ), ApproxCount( $I_R, \pi$ )
11:  $C_\pi = \max(C(I_L), C(I_R))$  ▶ Update count based on latest information
12: Update PathCache entry for  $\pi$  with ( $C_\pi, T_\pi + 1$ )
13: if  $C(I_L) \geq C(I_R)$  then ▶ Send back answer to Algorithm 1
14:   return 1
15: else
16:   return 0
17:
18: procedure APPROXCOUNT( $I, \pi$ ) ▶ Approximate  $C_\pi(I)$ 
19:   Inputs = Select  $k$  uniformly random inputs from  $I$ 
20:   ratios = MonteCarloExecutions(Program,  $\pi$ , Inputs) ▶ Algorithm 2
21:   return  $|I| * \min(\text{ratios})$  ▶ Equation 4
    
```

**2.4.2 Efficiently Approximating The Summation.** The method described in the prior section only deals with a single path reaching the target. To handle multiple paths, we need to sum each path's count to get a total count as mentioned in Equation 3. The challenge, however, is that although we can efficiently approximate counts for a single path, performing this procedure for each path at each oracle query quickly becomes computationally intractable if there are a large number of paths reaching the target.

Instead of computing this sum through contributions from each individual path count at each oracle query, we approximate the sum by only including contributions from a single path with the largest corresponding count. We select the largest-count path as its count best preserves the sum compared to any other single path. However, we do not apriori know which path has the largest count, so we initially spend some computation approximating each path's individual count, amortizing this cost over subsequent oracle queries. Hence, on the fuzzer's first oracle query, we identify the path with the largest count by approximating the count over the input space  $\mathbb{I}$  along each path. However, there will always be uncertainty in this path identification process due to approximation error.

To capture the uncertainty from our approximate counts, we add a correction factor  $\sqrt{\frac{\log(t)}{T_\pi}}$  shown in Algorithm 3 to also explore alternative paths a small number of times, borrowed from the multi-armed bandit literature [55].  $C_\pi$  denotes our latest count of inputs that reach the target along path  $\pi$  and  $T_\pi$  denotes number of times path  $\pi$  has been selected prior to the  $t$ -th oracle query. This correction factor conceptually balances uncertainty because as the algorithm acquires more certainty about a path  $\pi$  by selecting it more, thereby increasing  $T_\pi$ , the correction factor gradually decreases as the term is inversely proportional to  $T_\pi$ . We keep track of the most recent count information per path through a cache data structure called the PathCache.

### 3 IMPLEMENTATION

**Toolchain.** We implement algorithms 1, 2, 3 in C. We use LLVM [35] instrumentation and signal handlers to handle the branch and exception logic, respectively in Algorithm 2. We also incorporate the fork-server optimization used in state-of-the-art fuzzers [13, 17, 43, 65]. As described in Section 2.4, the error of the sample mean and variance drops exponentially fast in the number of program executions  $\sim \frac{1}{e^k}$ , so we set  $k = 5$  in Algorithm 3 such that  $p = 0.01$  in Algorithm 1 because  $\frac{1}{e^5} \leq 0.01$ . To reduce storage overheads, we implement the PathCache as a trie which avoids duplication when paths share edges. Moreover, we do not track  $k$  branch distances per branch in Algorithm 2, but rather compute the sample mean and variance in a streaming setting [60], so that we only store a constant number of values for any number of program executions  $k$ . Such techniques contribute to our minimal performance overheads in Section 4.3.

**Reducing Loop Overheads.** Real-world programs use loops causing the same branch to be visited many times during a program execution. If a single branch is visited a million times per execution, a naive implementation of Algorithm 2 will store a million branch distances for this branch per execution. Instead, we share information across multiple visits to a branch to reduce loop storage overheads. Specifically, in a single Monte Carlo Execution, if a branch is visited multiple times, the branch distance at each visit contributes to the (streaming) mean and variance of the branch. We also enforce control-flows at runtime across multiple visits to a branch by attaching count information to each branch. In addition to the techniques mentioned earlier, these techniques better help us scale to large real-world programs and contribute to our minimal overheads in Section 4.3.

**Assigning A Total Order.** In Section 2.2, we use the lexicographic total order (i.e., flatten first byte, then second byte, and so forth) to unambiguously split the input space. Although noisy binary search is agnostic to the underlying total order, using lexicographic order in real-world programs assumes that any region of the input space is equally likely to change the counts of inputs that reach the target (i.e., all bytes equally contribute to program behavior). However, for many real-world programs, this assumption does not hold as experimental evidence shows that not all bytes equally contribute to program behaviors [8, 25, 52, 53, 61].

Therefore, instead of assigning a total order based on lexicographic order, we assign an order based on the observed program executions in the noisy counting oracle. Specifically, starting with the set of all byte indices, the algorithm partitions the set into two disjoint subsets of equal size, and for each subset, performs Monte Carlo Execution on inputs generated by perturbing byte values whose index belongs to the subset. If the program executions change the approximate count, the algorithm recursively repeats the prior step on the subset. Otherwise, the subset is ignored. The algorithm repeats this process until the only sets that remain are sets with a single byte index. We then assign a total order by prioritizing byte indices from these remaining sets ranked by how much each byte index increases the approximate count. We experimentally demonstrate the effectiveness of this approach in Appendix B.

**Preprocessing.** Existing work in directed greybox fuzzing [13, 16, 30] pre-computes information about a program (e.g., static analysis information or CFG distance) to better guide the directed greybox fuzzer. In our setting, we need to pre-compute the set of all paths that reach the target, a task where algorithms require prohibitively expensive runtimes over large real-world CFGs [24, 50]. Moreover, algorithms that generate a subset of paths [24] generally do not produce paths with repeated edges, and since loops are a common construct in real-world programs, the set of generated paths is unlikely to be realizable in real program executions.

We instead use the initial seed corpus to bootstrap a set of paths. Specifically, we generate a set of paths that reach the target by executing the program on a seed close to the target while randomly inverting the direction of branches along the corresponding execution path, keeping paths based on the program executions after the branches were inverted (e.g., reach the target). Consequently, we use this seed’s length to set the input region size. In Appendix C, we measure our preprocessing time, comparing it to that of directed greybox fuzzers to show that our preprocessing times are similar. We plan to explore better path generation strategies in future work, potentially using ideas from the symbolic execution literature [14, 19, 27, 47, 64].

**Randomly Generating Inputs.** We represent the input region as a  $d$ -dimensional hyperrectangle encoded as  $d$  intervals, where each interval represents upper and lower bounds on input values per dimension. Used in Algorithm 3, we select  $k$  inputs uniformly at random from the hyperrectangle by generating  $d$  integers independently and uniformly at random from each interval, repeating this process  $k$  times for  $k$  inputs of length  $d$ . If the initial seed belongs to a given hyperrectangle (i.e., the seed’s byte values are within the  $d$  intervals), we include it as part of the  $k$  inputs to better utilize initial seed corpus information when applicable.

Note that we do not keep track of a seed corpus. Instead, we keep track of a list of groups as shown in Algorithm 1, where each group corresponds to a tuple: (hyperrectangle, weight) and splitting an input region corresponds to adjusting the hyperrectangle’s per-dimension intervals. To mitigate potential error in the input region weights if the selected path changes during the oracle queries, we also keep track of the groups per path, which does not introduce significant storage overhead as shown in our performance overheads in Section 4.3 since our algorithm uses logarithmic number of groups in expectation with respect to the size of  $\mathbb{I}$ .

### 4 EVALUATION

Our evaluation seeks to answer the following research questions.

- (1) **Comparison against directed greybox fuzzers:** How does MC<sup>2</sup> compare to state-of-the-art directed greybox fuzzers?
- (2) **Bug Finding:** Can MC<sup>2</sup> find new real-world bugs?
- (3) **Performance Overhead:** What is the performance overhead of MC<sup>2</sup>?
- (4) **Design Choices:** Are MC<sup>2</sup>’s design choices justified?

**Compute Infrastructure.** Unless otherwise noted, we ran all experiments on a Ubuntu 18.04 workstation with a Ryzen Threadripper 2970WX 24-Core CPU and 128 GB RAM.

#### 4.1 RQ1: Fuzzers Comparison

**Tested Benchmarks.** To avoid any potential bias while creating our own CVE benchmark in terms of bug class or program type, we use the publicly available Magma benchmark [28], which was specifically curated from a diverse set of CVEs. We also evaluate on a subset of the Fuzzer Test Suite benchmark [2] covered by prior work [16, 43] to enable fair comparison.

**Baseline Fuzzers.** Following prior works in directed greybox fuzzing [16, 30, 36, 70], we primarily compare MC<sup>2</sup> against other directed greybox fuzzers like AFLGo [13]. Other directed greybox fuzzers are either not available in any form (source or binary) [16, 36, 70] or have not made their source code public yet [30] and cannot support our benchmarks (i.e., Magma and Fuzzer Test Suite) without significant modifications. We also reached out to the authors of several of these fuzzers and confirmed that their code is not available for a release at the time of this writing, but they are working on releasing their code soon. Therefore, we could not compare against them on our benchmarks (i.e., Magma and Fuzzer Test Suite).

To compare against alternative designs for directed greybox fuzzing other than AFLGo, we also evaluate MC<sup>2</sup> against ParmeSan [43] which supports a directed greybox fuzzer mode. We contacted the authors of ParmeSan and followed their advice to set it up. Furthermore, as ParmeSan and AFLGo build upon two significantly different regular (i.e., undirected) fuzzers: Angora [17] and AFL [65], respectively, we also include the results of the underlying fuzzer implementations to show the improvement a directed greybox fuzzer has over its undirected counterpart in Appendix Tables 14 and 15.

**Experimental Setup.** We follow the experimental setup based on prior work [13, 16, 30, 36, 43, 70]. We assign each fuzzer a single core and keep 20% of the cores unused to minimize interference. We configure each directed greybox fuzzer to use the default seeds and targets provided by the benchmarks. To avoid potential unfairness or bias in the results arising from how different fuzzing implementations deal with multiple targets, we give fuzzers one target per run to enable a fair comparison in line with prior work [36, 43]. We measure the time it takes to trigger the bug target (for Magma) or reach the target (for Fuzzer Test Suite) with a 6 hour timeout.

We pick 6 hours because it is the arithmetic mean of the times used by Hawkeye [16] and AFLGo [13] evaluations. Since each fuzzer includes some amount of preprocessing (e.g., distance computations), we also separately measure this time in Table 12 in Appendix C. We run with 20 independent trials, using arithmetic mean when reporting results. We note that our Fuzzer Test Suite experiments were performed on a workstation running Ubuntu 18.04 using an Xeon E5-2640 24-Core CPU with 128 GB RAM.

**Magma Results.** Table 2 summarizes the results as well as the result from applying the Mann-Whitney U test between MC<sup>2</sup> and the tested directed greybox fuzzers. We note that although we evaluated over the entire benchmark, not all bugs were triggered, and therefore, for space constraints, we only list the bugs triggered within the time budget in Table 2 following prior work [29].

MC<sup>2</sup> finds bugs 134x faster in arithmetic mean and 38x faster in median compared to the next best fuzzer AFLGo. Moreover, MC<sup>2</sup>'s improvement is statistically significant with a significance level of 0.05 for all bugs except PNG003. MC<sup>2</sup> was also able to find 28 bugs in

**Table 2: Mean time to trigger Magma bugs for each tested fuzzer over 20 trials. We only include the bugs that were triggered within 6 hours for space constraints. (x) refers to the speedup of MC<sup>2</sup> relative to the tested fuzzer. (p) refers to the p-value from the Mann-Whitney U test. Since ParmeSan crashed on php, we write N/A for it. T.O\* indicates 6 hour timeout. We highlight bugs only triggered by MC<sup>2</sup> in blue .**

Bug ID	MC <sup>2</sup>		AFLGo		ParmeSan		
	Time		Time	(x) (p)	Time	(x) (p)	
PDF010	3m15s		4h02m15s	74x <0.01	T.O*	>110x <0.01	
PDF016	3m23s		51m43s	15x <0.01	7m10s	2x <0.01	
PHP004	1m04s		4m09s	3x <0.01	N/A	N/A N/A	
PHP009	1m07s		17m08s	15x <0.01	N/A	N/A N/A	
PHP011	1m01s		15m24s	15x <0.01	N/A	N/A N/A	
PNG003	15s		15s	1x 0.25	1m38s	6x <0.01	
PNG006	1m36s		T.O*	>225x <0.01	2m03s	1x <0.01	
SSL002	1m44s		5m58s	3x <0.01	32m27s	18x <0.01	
SSL003	1m39s		4m30s	2x <0.01	16m27s	9x <0.01	
SSL009	4m59s		T.O*	>72x <0.01	4h51m19s	58x <0.01	
TIF005	9m33s		T.O*	>37x <0.01	3h48m49s	23x <0.01	
TIF006	9m36s		T.O*	>37x <0.01	4h03m29s	25x <0.01	
TIF007	8m18s		1h39m40s	12x <0.01	56m40s	6x <0.01	
TIF012	9m59s		2h46m00s	16x <0.01	3h52m50s	23x <0.01	
TIF014	1m36s		5h49m19s	218x <0.01	T.O*	>225x <0.01	
XML017	16s		1m09s	4x <0.01	23m15s	87x <0.01	
PDF003	1m39s		T.O*	>218x <0.01	T.O*	>218x <0.01	
PDF008	3m21s		T.O*	>107x <0.01	T.O*	>107x <0.01	
PDF011	1m41s		T.O*	>213x <0.01	T.O*	>213x <0.01	
PDF018	1m43s		T.O*	>209x <0.01	T.O*	>209x <0.01	
PDF019	1m37s		T.O*	>216x <0.01	T.O*	>216x <0.01	
PNG001	3m17s		T.O*	>109x <0.01	T.O*	>109x <0.01	
PNG007	3m21s		T.O*	>107x <0.01	T.O*	>107x <0.01	
SSL020	9m16s		T.O*	>38x <0.01	T.O*	>38x <0.01	
TIF001	9m43s		T.O*	>37x <0.01	T.O*	>37x <0.01	
TIF002	9m58s		T.O*	>36x <0.01	T.O*	>36x <0.01	
TIF009	9m49s		T.O*	>36x <0.01	T.O*	>36x <0.01	
XML009	13s		T.O*	>1661x <0.01	T.O*	>1661x <0.01	
<b>Mean speedup</b>				134x		144x	
<b>Median speedup</b>				38x		39x	

total, 16 more than the next-best fuzzer AFLGo, which found only 12 bugs within the time budget. We note that since MC<sup>2</sup> does not generate inputs of different length, we also ran this experiment with variants of AFLGo and ParmeSan that do not change the input length. We found the results to be nearly identical (mean speedup changed by 2%), so we did not insert the full table for space constraints. Overall, our results show the promise of using noisy binary search and approximate counting for directed greybox fuzzing.

**Case Study.** We highlight a particular bug PNG001 in Figure 2 found only by MC<sup>2</sup>. This bug is guarded by constraints and only a single input value width=0x55555555 will cause a divide by zero when row\_factor overflows. We hypothesize that AFLGo did not trigger this bug in the time budget because the chance of producing this specific input value through mutations is small and fuzzer heuristics such as setting values to MAX\_INT also fail. In addition, we hypothesize that ParmeSan did not trigger this bug in the time budget because although it uses gradient descent and taint tracking to narrow down the input space, it cannot effectively reason about nested constraints (Lines 6 and 7). In contrast, MC<sup>2</sup> was able to successfully find this input value through noisy binary search. Moreover, upon manual source code analysis, we found this bug can only be triggered along an execution path that sets channel=3, showing that MC<sup>2</sup> was able to successfully reason across multiple execution paths.

```

1 void png_check_chunk_length() {
2   // set based on input file
3   u32 width, height, colortype;
4
5   /* constraints from libpng_read_fuzzer.cc */
6   if (width < UINT31_MAX) {
7     if (width*height < 10^8) {
8       u32 channels;
9       switch(colortype) {
10        case PALETTE: channels = 1; break
11        case GRAY: channels = 2; break;
12        case RGB: channels = 3; break;
13        case ALPHA: channels = 4; break;
14      }
15
16      u32 row_factor = width * channels + 1;
17      if (row_factor == 0) {
18        // divide-by-zero bug target
19      }
20    }
21  }
22 }

```

Figure 2: Simplified code of Magma PNG001 (CVE-2018-13785).

Table 3: Mean time to reach Fuzzer Test Suite targets for each tested fuzzer over 20 trials. (x) refers to the speedup of MC<sup>2</sup> relative to the tested fuzzer. (p) refers to the p-value from the Mann-Whitney U test. T.O\* indicates 6 hour timeout. We highlight targets only reached by MC<sup>2</sup> in blue .

Bug ID	MC <sup>2</sup> Time	AFLGo			ParmeSan		
		Time	(x)	(p)	Time	(x)	(p)
ttgload.c:1710	1s	1s	1x	0.07	1s	1x	0.07
ttinterp.c:2186	9m57s	T.O*	>36x	<0.01	20m	2x	<0.01
cf2intrap.c:361	58s	23m	23x	<0.01	T.O*	>372x	<0.01
jdmarker.c:659	32s	1h07m	125x	<0.01	5m	9x	<0.01
pngutil.c:139	1s	1s	1x	0.07	1s	1x	0.07
pngutil.c:3182	28s	2m30s	5x	<0.01	1m	2x	<0.01
pngread.c:738	1s	1s	1x	0.07	1s	1x	0.07
pngutil.c:1393	51s	T.O*	>423x	<0.01	T.O*	>423x	<0.01
<b>Mean speedup</b>			77x			102x	
<b>Median speedup</b>			15x			2x	

**Fuzzer Test Suite Results.** Table 3 summarizes the results. MC<sup>2</sup> reaches targets 102x faster in arithmetic mean and 2x faster in median than ParmeSan and 77x faster in arithmetic mean and 15x faster in median than AFLGo, with statistical significance on all targets that were not reached within a few seconds. Moreover, MC<sup>2</sup> reaches 2 more targets compared to either ParmeSan or AFLGo. While cross-comparisons between papers is challenging due to stochasticity in fuzzers and hardware, our results are similar to prior work [16, 43], giving us confidence in our experimental setup of the tested fuzzers.

**Result 1:** Over the Magma benchmark, MC<sup>2</sup> finds bugs 134x faster in arithmetic mean and 38x faster in median compared to the next best fuzzer AFLGo. It also finds 28 bugs in total, 16 more than the next-best fuzzer AFLGo.

## 4.2 RQ2: Bug Finding

For our bug finding experiments, we evaluate over programs based on prior work [31, 37, 39, 45, 53, 59, 63] and Magma listed in Table 4. To find the targets for directed fuzzing, we re-use an idea from prior

Table 4: Tested programs in bug finding experiments.

Library	Program	Version
libpng	libpng_read_fuzzer	Commit a37d483...
poppler	pdf_fuzzer	Commit 1d23101...
binutils	nm -C	2.36
binutils	objdump -xD	2.36
openssl	x509	Commit 3bd5319...
libxml2	xmllint	Commit 07920b4...

work [16, 43] and use Undefined Behavior Sanitizer [4] to identify bug targets. This tool often reports a large number of bug targets, and if all are set as targets, the fuzzer effectively becomes a coverage-guided fuzzer instead of being directed. Instead, we randomly pick one target per function and run each fuzzer with these same targets over a 24 hour run. We start each fuzzer with the initial Magma corpus and a small set of valid ELF files. We report the total number of bugs found, repeating this experiment 10 times to minimize variability.

Table 5: Categorization of new bugs found by each fuzzer.

Bug Type	ParmeSan	AFLGo	MC <sup>2</sup>
divide-by-zero	0	0	1
denial-of-service	3	4	6
stack/heap overflow	10	8	13
integer overflow	21	17	29
Total	34	29	49

In our 24 hour runs, we found previously-unknown real-world bugs in binutils, libxml2, and libpng. Table 5 summarizes the results in terms of bug type. Since counting the number of crashing inputs may inflate the bug count, we take the following approach to better compute the bug count based on prior work [8, 17, 53]. We first use AFL-CMin to filter out duplicate crashing inputs, followed by another deduplication procedure based on unique stack traces. From this reduced set of inputs, we manually review the stack traces and corresponding source code to further deduplicate these inputs. We responsibly disclosed these bugs to the developers and all bugs were confirmed, most of which have been fixed in the latest versions of the programs. Our results show that MC<sup>2</sup> finds 15 more bugs than the next best fuzzer ParmeSan.

**Result 2:** MC<sup>2</sup> finds 49 previously-unknown real world bugs, 15 more than the next best fuzzer ParmeSan.

## 4.3 RQ3: Performance Overhead

Since instrumented target program executions dominate the fuzzing overhead [42], we evaluate the performance overhead of Monte Carlo Execution relative to native (uninstrumented) execution as well as a fuzzer-instrumented execution that tracks edge coverage and distance (i.e., AFLGo). We run the Magma programs over the initial seed corpus inputs and take the arithmetic mean of the results from 10 independent trials. In addition, we measure the total memory footprint of MC<sup>2</sup>'s data structures (e.g., PathCache and weight groups in Algorithm 1) by re-running our Magma evaluation and tracking the total memory consumed in MBs, reporting the arithmetic mean over 10 independent trials.

**Table 6: Monte Carlo Execution Overheads relative to native (uninstrumented) and fuzzer-instrumented execution over Magma.**

Library	MC <sup>2</sup> vs Native		MC <sup>2</sup> vs Fuzzer (AFLGo)	
	Runtime	Memory	Runtime	Memory
libpng	94%	30%	26%	4%
libtiff	78%	2%	16%	1%
libxml2	135%	37%	38%	8%
openssl	117%	15%	42%	6%
php	86%	9%	29%	4%
poppler	87%	10%	25%	3%
sqlite3	136%	7%	34%	4%
Arithmetic mean	105%	16%	30%	4%
Median	94%	10%	29%	4%

**Table 7: MC<sup>2</sup>'s data structures size in MBs over Magma benchmark.**

Library	Data Structures Size (MBs)
libpng	12.1
libtiff	21.8
libxml2	1.6
openssl	59.7
php	1.6
poppler	28.1
sqlite3	20.8
Arithmetic mean	20.9
Median	20.8

Table 6 summarize the performance overheads of Monte Carlo Execution. Monte Carlo Execution adds runtime overheads of 105% in arithmetic mean and 94% in median as well as memory overheads of 16% in arithmetic mean and 10% in median relative to native execution. Relative to a fuzzer-instrumented execution, the overheads are smaller: runtime overheads of 30% in arithmetic mean and 29% in median as well as memory overheads of 4% in arithmetic mean and 4% in median. We attribute the additional memory and runtime overheads to computing the (streaming) mean and variance for each branch, which requires additional memory as well as floating point arithmetic.

We also summarize the memory footprint: 20.9 MBs in arithmetic mean and 20.8 MBs in median (< 1 GB) with full details in Table 7. These results show that the data structures do not consume large amounts of memory. Note that MC<sup>2</sup>, a prototype, still consistently outperforms other fuzzers despite this overhead, showing the promise of our technique. Nonetheless, we believe there are still ways to further cut down our prototype's overhead.

**Result 3:** MC<sup>2</sup> adds 30% runtime and 4% memory overhead in arithmetic mean relative to a fuzzer's instrumentation and 105% runtime and 16% memory overhead in arithmetic mean relative to native execution. In addition, MC<sup>2</sup> data structures consume < 1 GB of memory.

#### 4.4 RQ4: Design Choices

We conduct experiments to measure the effect of three design choices: (i) Chebyshev's inequality for uniconstraint counts, (ii) using the minimum uniconstraint count, and (iii) path selection.

For each design choice experiment, we run MC<sup>2</sup> on a representative subset from the Magma benchmark, repeated 10 times. To form a representative subset, we pick 3 bugs randomly from three categories: bugs found within 60 seconds, bugs found more than 120

**Table 8: Mean time to trigger the bug across various techniques to approximate uniconstraint counts over 10 trials.**

Bug ID	MC <sup>2</sup>	Rule-Of-3	Good-Turing
XML009	13s	<i>T.O*</i>	<i>T.O*</i>
PNG003	15s	<i>T.O*</i>	<i>T.O*</i>
XML017	16s	2m17s	1m54s
PHP004	1m04s	15m8s	10m40s
PDF011	1m41s	<i>T.O*</i>	<i>T.O*</i>
PHP009	1m07s	<i>T.O*</i>	<i>T.O*</i>
SSL020	9m16s	<i>T.O*</i>	<i>T.O*</i>
TIF009	9m49s	<i>T.O*</i>	<i>T.O*</i>
PDF008	3m21s	<i>T.O*</i>	<i>T.O*</i>
Arithmetic mean speedup		427x	426x
Median speedup		107x	107x

**Table 9: Mean time to trigger the bug across various techniques to approximate path counts over 10 trials.**

Bug ID	MC <sup>2</sup>	Multiply Uniconstraint Counts
XML009	13s	5m59s
PNG003	15s	6m15s
XML017	16s	1m31s
PHP004	1m04s	13m01s
PDF011	1m41s	34m31s
PHP009	1m07s	7m16s
SSL020	9m16s	1h10m26s
TIF009	9m49s	19m09s
PDF008	3m21s	56m57s
Arithmetic mean speedup		13x
Median speedup		12x

seconds, and bugs found between these times. Our subset includes at least one bug from each library in Magma. Moreover, it includes bugs that only MC<sup>2</sup> triggers as well as other tested fuzzers trigger. We describe each design choice experiment in more detail below.

**4.4.1 Chebyshev's Inequality for Uniconstraint Counts.** In this experiment, we compare our Chebyshev-based technique to compute probabilistic upper bounds on  $r$  (i.e., see Equation 5 in Section 2.4) against alternate techniques when  $r = 0$  (i.e., zero uniconstraint counts). Specifically, we compare against the Rule-of-3 and Good-Turing techniques from the Natural Language Processing and Biostatistics literature [18, 34], which have also been used in prior work in fuzzing [11, 68]. In contrast to our probabilistic upper bounds which use mean and variance information, these methods upper bound  $r$  by computing  $r = \frac{3}{N}$  (Rule-of-3) or the smallest non-zero  $r$  across all branches (Good-Turing) via  $r = \min(\{r_{E_1}, r_{E_2}, \dots, r_{E_T} \text{ such that } r_{E_i} \neq 0\})$ .

Table 8 summarizes the results. MC<sup>2</sup> improves upon the next-best technique Good-Turing by 426x in arithmetic mean and 107x in median. Our results highlight the importance of probabilistic upper bounds in MC<sup>2</sup>.

**4.4.2 Minimum Uniconstraint Count.** In Section 2.4, we placed an upper bound on the count of inputs that reach the target along an execution path for a given input region using the minimum uniconstraint count. In this experiment, we compare our technique which uses information from a single uniconstraint count with an alternate one that incorporates information from all uniconstraint counts by multiplying them.

Table 9 summarizes the results. MC<sup>2</sup> improves upon the multiply uniconstraint counts technique by 13x in arithmetic mean and 12x

**Table 10: Mean time to trigger the bug across various techniques for path selection over 10 trials.**

Bug ID	MC <sup>2</sup>	Epsilon-greedy	Greedy
XML009	13s	9s	11s
PNG003	15s	38s	23s
XML017	16s	11s	13s
PHP004	1m04s	2m08s	3m12s
PDF011	1m41s	2m06s	1m03s
PHP009	1m07s	2m14s	3m21s
SSL020	9m16s	14m50s	4h38m
TIF009	9m49s	15m42s	4h54m30s
PDF008	3m21s	5m22s	1h40m30s
Arithmetic mean speedup		1.5x	11x
Median speedup		1.6x	3x

in median, showing the utility of approximating the count along an execution path using the minimum unconstraint count. We hypothesize this improvement occurs because multiplying unconstraint counts to approximate the count along a path corresponds to an independence assumption between individual constraints (i.e., the branch constraints share no variables and hence the counts are independent), which is generally not true for most real-world programs, as shown in the symbolic execution literature [14, 19, 26, 27, 56, 64].

**4.4.3 Path Selection.** We discuss in Section 2.4 the importance of selecting alternate paths with large counts due to approximation error, leading us to use the uncertainty term from the multi-armed bandit literature [55] in Algorithm 3. In this experiment, we compare against alternate strategies based on the multi-armed bandit literature. We compare against a strategy that sets the uncertainty term to zero and greedily picks the path with the largest count (Greedy). We also compare against a variant called Epsilon-greedy that also sets the uncertainty term to zero but instead of following Greedy all the time, it randomly selects another path based on a coin flip with bias  $\epsilon$ , set to  $\epsilon = 0.5$  to equally balance the trade-off.

Table 10 summarizes the results. While MC<sup>2</sup> improves upon Greedy by 11x on average and 3x in median, it only improves upon Epsilon-greedy by 1.5x on average and 1.6x in median. Our results show the utility of selecting alternate paths to reflect our uncertainty, but also indicate that simple strategies such as Epsilon-greedy can work as well as more advanced ones that incorporate an uncertainty correction factor.

**Result 4:** Our experimental results justify MC<sup>2</sup>’s design choices with speedups  $\geq 1.5x$  in arithmetic mean and  $\geq 1.6x$  in median.

## 5 RELATED WORK

**Approximate Counting.** Approximate counting has been used in many different contexts including counting the number of solutions to SAT formulas [15, 33], flash memory [20], and database systems [9]. Techniques for approximate counting build upon Monte Carlo counting as well as universal hash functions [15], which provide the property of uniformly partitioning each object to be counted into roughly equally-sized groups. We plan to investigate incorporating such techniques in the future.

Recently, approximate counting was also used in seed scheduling for coverage-guided fuzzing. She et al. approximate the count of

reachable and feasible edges using graph centrality [54]. In contrast, we approximate the count of inputs that reach the target using Monte Carlo counting for directed greybox fuzzing. Generalizing MC<sup>2</sup> from directed greybox fuzzing to the coverage-guided fuzzer setting remains an open question for future work and potentially may involve information entropy from Böhme et al. [12] or abstraction functions from Salls et al. [49].

**Directed Greybox Fuzzing.** Starting with the promising results of AFLGo: finding the HeartBleed vulnerability orders-of-magnitude faster than a directed whitebox fuzzer [13], directed greybox fuzzing has seen multiple research directions. One line of work incorporates additional information into the distance computations such as branch distance [36] or function similarity [16]. In contrast, MC<sup>2</sup> uses noisy binary search and approximate counts, not distance, to guide the fuzzer.

Based on the observation that directed greybox fuzzers consume a lot of time on executions that fail to reach the target, another promising line of work seeks to increase the fuzzer’s efficiency by not executing on inputs that are either unlikely to reach the target [70] or provably cannot [30]. Our approach is complementary to such techniques as we can potentially use them to bias our random input selection process to avoid such inputs. Recent work has also directed a fuzzer with application-specific techniques [41, 43, 67, 69] and incorporating such application-specific techniques is an interesting question for future work.

## 6 CONCLUSION

In this paper, we build an asymptotically optimal directed greybox fuzzer using noisy binary search and a noisy counting oracle. We also empirically show the promise of our fuzzer as it outperforms existing directed greybox fuzzers by up to two orders of magnitude, on average, over Magma and Fuzzer Test Suite.

## ACKNOWLEDGEMENTS

We thank Clayton Sanford, Samuel Deng, Andreas Kellas, Amol Pasarkar, Dennis Roelke, Gabriel Ryan, Zhongtian Chen, Yuhao Li, Ming Yuan, Christian Kroer, and Junfeng Yang for their helpful comments, and the reviewers for their valuable feedback. Peter Coffman helped create tables, improve code quality, and optimize the implementation. Abhishek Shah is supported by an NSF Graduate Fellowship. This work is supported partially by NSF grants CNS-18-42456, CNS-18-01426; a NSF CAREER award; a Google Faculty Fellowship; a JP Morgan Faculty Fellowship; a Capital One Research Grant; and an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT) (No.2020-0-00153). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, NSF, Google, Capital One, J.P. Morgan, or the Korean Government.

## REFERENCES

- [1] 2022. AFLGo Max Input Size. <https://github.com/aflgo/aflgo/blob/b170fad54396f376160befd87adbb28b27c15d9/config.h#L142>.
- [2] 2022. Fuzzer Test Suite. <https://github.com/google/fuzzer-test-suite>.
- [3] 2022. ParmeSan Max Input Size. <https://github.com/vusec/parmesan/blob/fac580130146c07a2a0f82a24dfe0704e1851ab3/common/src/config.rs#L12>.
- [4] 2022. Undefined Behavior Sanitizer. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.

- [5] Scott Aaronson, Andris Ambainis, Andrej Bogdanov, Krishnamoorthy Dinesh, and Cheung Tsun Ming. 2021. On Quantum Versus Classical Query Complexity. *Electron. Colloquium Comput. Complex.* (2021), 115.
- [6] Andris Ambainis. 2018. Understanding Quantum Algorithms via Query Complexity. In *International Congress of Mathematicians: Rio de Janeiro*. World Scientific, 3265–3285.
- [7] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The Multiplicative Weights Update Method: a Meta-Algorithm and Applications. *Theory of Computing* (2012), 121–164.
- [8] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium*. The Internet Society.
- [9] Morton M. Astrahan, Mario Schkolnick, and Kyu-Young Whang. 1987. Approximating the Number of Unique Values of an Attribute Without Sorting. *Inf. Syst.* 12, 1 (1987), 11–15.
- [10] Michael Ben-Or and Avinatan Hassidim. 2008. The Bayesian Learner is Optimal for Noisy Binary Search (and Pretty Good for Quantum as Well). In *49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 221–230.
- [11] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholtz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 230–241.
- [12] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 678–689.
- [13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2329–2344.
- [14] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 209–224.
- [15] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable Approximate Model Counter. In *Principles and Practice of Constraint Programming - 19th International Conference (Lecture Notes in Computer Science, Vol. 8124)*. Springer, 200–216.
- [16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2095–2108.
- [17] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. *IEEE Symposium on Security and Privacy* (2018), 711–725.
- [18] Stanley F. Chen and Joshua Goodman. 1996. An Empirical Study of Smoothing Techniques for Language Modeling. In *34th Annual Meeting of the Association for Computational Linguistics*. Morgan Kaufmann Publishers / ACL, 310–318.
- [19] 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 *IEEE Symposium on Security and Privacy*. IEEE, 1580–1596.
- [20] Jacek Cichon and Wojciech Macyna. 2011. Approximate Counters for Flash Memory. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, 185–189.
- [21] Dariusz Dereniowski, Aleksander Lukaszewicz, and Przemyslaw Uznanski. 2021. An Efficient Noisy Binary Search in Graphs via Median Approximation. In *Combinatorial Algorithms - 32nd International Workshop (Lecture Notes in Computer Science, Vol. 12757)*. Springer, 265–281.
- [22] Devdatt P. Dubhashi and Alessandro Panconesi. 2009. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press.
- [23] Philippe Flajolet. 1985. Approximate Counting: A Detailed Analysis. *BIT* 25, 1 (1985), 113–134.
- [24] Luigi Fratta and Ugo Montanari. 1975. A Vertex Elimination Algorithm for Enumerating all Simple Paths in a Graph. *Networks* 5, 2 (1975), 151–177.
- [25] 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 Association, 2577–2594.
- [26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (2005), 213–223.
- [27] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium*. The Internet Society.
- [28] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *ACM Measurement and Analysis of Computing Systems* (2020), 49:1–49:29. Issue 3.
- [29] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery.
- [30] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed Grey-Box Fuzzing with Provable Path Pruning. In *IEEE Symposium on Security and Privacy*.
- [31] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *Proceedings of the 28th USENIX Security Symposium*.
- [32] Richard M. Karp and Robert Kleinberg. 2007. Noisy Binary Search and its Applications. In *Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 881–890.
- [33] Richard M. Karp, Michael Luby, and Neal Madras. 1989. Monte-Carlo Approximation Algorithms for Enumeration Problems. *J. Algorithms* 10, 3 (1989), 429–448.
- [34] Panagiotis I. Koukos and Nicholas M. Glykos. 2014. On the Application of Good-Turing Statistics to Quantify Convergence of Biomolecular Simulations. *J. Chem. Inf. Model.* 54, 1 (2014), 209–217.
- [35] Chris Latner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, 75–.
- [36] Dwanget Lee, Woohul Shim, and Youngyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium*. USENIX Association, 3559–3576.
- [37] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. In *33rd IEEE/ACM International Conference on Automated Software Engineering*. ACM.
- [38] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering Fitness Gradients for Interprocedural Boolean Flags in Search-based Testing. In *29th International Symposium on Software Testing and Analysis*.
- [39] 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 Association, 1949–1966.
- [40] Andreas Maurer and Massimiliano Pontil. 2009. Empirical Bernstein Bounds and Sample-Variance Penalization. In *The 22nd Conference on Learning Theory*.
- [41] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Finding Counterexamples of Temporal Logic Properties in Software Implementations via Greybox Fuzzing. In *International Conference on Software Engineering*.
- [42] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy*. IEEE, 787–802.
- [43] Sebastian Osterlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *29th USENIX Security Symposium*. USENIX Association, 2289–2306.
- [44] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *23rd USENIX Security Symposium*. USENIX Association, 829–844.
- [45] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 697–710.
- [46] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Trans. Software Eng.* 47, 9 (2021), 1980–1997.
- [47] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *29th USENIX Security Symposium*. USENIX Association, 181–198.
- [48] Napat Rujeerapaiboon, Daniel Kuhn, and Wolfram Wiesemann. 2018. Chebyshev Inequalities for Products of Random Variables. *Math. Oper. Res.* 43, 3 (2018), 887–918.
- [49] Christopher Salls, Aravind Machiry, Adam Doupe, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Exploring Abstraction Functions in Fuzzing. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*.
- [50] Robert Sedgewick. 2002. *Algorithms in C - Part 5: Graph Algorithms*. Addison-Wesley-Longman.
- [51] C.A. Shaffer. 2012. *Data Structures and Algorithm Analysis in C++, Third Edition*. Dover Publications.
- [52] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. 2020. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. In *IEEE Symposium on Security and Privacy*. IEEE, 1527–1543.
- [53] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy*. IEEE, 803–817.
- [54] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *IEEE Symposium on Security and Privacy*.
- [55] Aleksandrs Slivkins. 2019. Introduction to Multi-Armed Bandits. *Found. Trends Mach. Learn.* 12, 1-2 (2019), 1–286.

- [56] 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 *23rd Annual Network and Distributed System Security Symposium*. The Internet Society.
- [57] Leslie G. Valiant. 1984. A Theory of the Learnable. In *16th Annual ACM Symposium on Theory of Computing*. ACM, 436–445.
- [58] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. 2019. Automated Hybrid Analysis of Android Malware through Augmenting Fuzzing with Forced Execution. *IEEE Trans. Mob. Comput.* 18, 12 (2019), 2768–2782.
- [59] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Network and Distributed System Security Symposium*.
- [60] B. P. Welford. 1962. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics* 4, 3 (1962), 419–420.
- [61] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing Without Valid Seed Inputs. In *41st International Conference on Software Engineering*. IEEE / ACM, 712–723.
- [62] Wei You, Zhuo Zhang, Yonghwi Kwon, Youssa Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *IEEE Symposium on Security and Privacy*. IEEE, 1121–1138.
- [63] 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 Association, 2307–2324.
- [64] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium*. USENIX Association, 745–761.
- [65] Michał Zalewski. 2022. American Fuzzy Lop (AFL) README. <http://lcamtuf.coredump.cx/afl/README.txt>.
- [66] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2022. The fuzzing book.
- [67] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. 2022. Exploit the Last Straw That Breaks Android Systems. In *IEEE Symposium on Security and Privacy*.
- [68] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium*. The Internet Society.
- [69] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2169–2182.
- [70] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *29th USENIX Security Symposium*. USENIX Association, 2255–2269.

## A PROOFS

**PROOF OF THEOREM 2.1.** Information theory [23] states that to identify (i.e., encode) a unique element in a set containing  $N$  elements, we require at least  $\log(N)$  bits. Similarly, to identify a target-reaching input in an input space of size  $N$ , any directed fuzzing algorithm requires at least  $\log(N)$  oracle queries, up to constant factors, since each oracle query provides a constant  $c$  bits of information.  $\square$

**PROOF OF THEOREM 2.2.** Ben-Or et al. [10] in Theorem 2.1 prove that their noisy binary search algorithm requires  $(1 - \delta) * \frac{\log(N)}{(\frac{1}{2} - p)^2}$  comparisons in expectation to find the target with success probability at least  $1 - \delta$ . If we map our noisy oracle queries to their noisy comparisons and our input space with a lexicographic total order to their array, our fuzzing algorithm in Algorithm 1 directly translates to their noisy binary search algorithm and therefore inherits the same analysis. The algorithm analysis uses information entropy arguments to show that the expected information gain increases at each query, followed by concentration bounds to show that when the algorithm terminates, the algorithm can identify the region containing the target with high probability. For more details, see Section 2.3 in [10].

**PROOF OF THEOREM 2.3.** Theorem 2.8 by Ben-Or et al. [10] shows that  $\Omega((1 - \delta) * \frac{\log(N)}{(\frac{1}{2} - p)^2})$  is a lower bound for any noisy binary search algorithm (i.e., cannot be improved upon) and therefore implies that Algorithm 1 is optimal. This lower bound results from a reduction to a well-studied problem in information theory where two parties wish to communicate over a noisy channel (i.e., noisy channel coding problem). For exact details, we refer the reader to Section 2.4 in [10]. We note in the special case of a noisy counting oracle that returns  $c = 1$  bit of information without noise ( $p = 0$ ), Algorithm 1 also meets the lower bound, up to constant factors, from Theorem 2.1, so it is optimal in both noisy and noiseless settings.  $\square$

## B TOTAL ORDER ASSIGNMENT

In Section 3, we discussed that although our binary search algorithm is agnostic to the underlying total order, which is necessary to ensure splitting an input region is unambiguous, lexicographic order is a poor choice because it assumes that all bytes equally contribute to the count of inputs reaching the target. Empirical evidence from prior work [8, 25, 52, 53, 61] has shown that not all input bytes equally contribute to program behaviors and therefore such an assumption does not hold for many real-world programs. In this experiment, we show that lexicographic order is poor choice by comparing it with our technique to assign a total order.

Table 11 summarizes the results. MC<sup>2</sup> outperforms the lexicographic ordering by 210x in arithmetic mean and 54x in median, showing that lexicographic ordering is a poor choice. In the future, we plan to investigate what properties constitute an optimal total order assignment.

## C PREPROCESSING TIMES

Existing directed greybox fuzzers use preprocessing to better identify which inputs are more likely to reach the target as described in Section 3. We measure the preprocessing times of the tested fuzzers to see how they compare.

For AFLGo, we measure the time it takes to compute distance over the control-flow graph, which consists of visiting every function, computing intra-function distances, and using callgraphs to compute distances between functions. For ParmeSan, it uses a dynamic

**Table 11: Mean time to trigger the bug with and without lexicographic order across 10 trials.**

Bug ID	MC <sup>2</sup>	Lexicographic Order
XML009	13s	8m14s
PNG003	15s	13m27s
XML017	16s	4m12s
PHP004	1m04s	4h12m21s
PDF011	1m41s	T.O*
PHP009	1m07s	3h30m17s
SSL020	9m16s	20m18s
TIF009	9m49s	16m23s
PDF008	3m21s	T.O*
Arithmetic mean speedup		210x
Median speedup		54x

**Algorithm 4** Optimal Deterministic Fuzzer.

<b>Input:</b> $I \leftarrow$ Input Space as Array $O \leftarrow$ Noiseless ( $p = 0$ ) Counting Oracle from Equation 1	
1: $l = 0; r =  I  - 1$	$\triangleright$ initialize left and right bounds of $I$
2: <b>while</b> $l < r$ <b>do</b>	
3: $m = \lfloor (l + r) / 2 \rfloor$	$\triangleright$ select midpoint input
4: $I_L, I_R =$ left and right input regions of index $m$	
5: <b>if</b> $O(I_L, I_R) = 1$ <b>then</b>	
6: $r = m$	$\triangleright$ select left subregion
7: <b>else</b>	
8: $l = m + 1$	$\triangleright$ select right subregion

**Table 14:** Mean time to trigger Magma bugs for each tested fuzzer’s undirected counterpart over 20 trials. Since Angora crashed on php, we write N/A for it. See Table 2 for the full caption.

Bug ID	MC <sup>2</sup> Time	AFL		Angora	
		Time	(x) (p)	Time	(x) (p)
PDF010	3m15s	4m25s	1x 0.09	$T.O^*$	>111x <0.01
PDF016	3m23s	4m02s	1x 0.11	12m	4x <0.01
PHP004	1m04s	2m39s	2x <0.01	N/A	N/A N/A
PHP009	1m07s	3m38s	3x <0.01	N/A	N/A N/A
PHP011	1m01s	2m29s	2x <0.01	N/A	N/A N/A
PNG003	15s	15s	1x 0.25	59s	4x <0.01
PNG006	1m36s	$T.O^*$	>225x <0.01	2m42s	2x 0.03
SSL002	1m44s	4m06s	2x <0.01	55m53s	32x <0.01
SSL003	1m39s	2m50s	2x <0.01	20m40s	13x <0.01
SSL009	4m59s	$T.O^*$	>72x <0.01	4h56m06s	59x <0.01
TIF005	9m33s	$T.O^*$	>38x <0.01	3h57m57s	25x <0.01
TIF006	9m36s	5h21m19s	33x <0.01	4h40m09s	29x <0.01
TIF007	8m18s	1h13m28s	9x 0.04	1h31m48s	11x <0.01
TIF012	9m59s	1h54m37s	11x <0.01	4h51m22s	29x <0.01
TIF014	1m36s	4h55m49s	185x <0.01	5h38m17s	211x <0.01
XML017	16s	1m23s	5x <0.01	1m57s	7x <0.01
PDF003	1m39s	$T.O^*$	>218x <0.01	$T.O^*$	>218x <0.01
PDF008	3m21s	$T.O^*$	>107x <0.01	$T.O^*$	>107x <0.01
PDF011	1m41s	$T.O^*$	>214x <0.01	$T.O^*$	>214x <0.01
PDF018	1m43s	$T.O^*$	>210x <0.01	$T.O^*$	>210x <0.01
PDF019	1m37s	$T.O^*$	>223x <0.01	$T.O^*$	>223x <0.01
PNG001	3m17s	$T.O^*$	>110x <0.01	$T.O^*$	>110x <0.01
PNG007	3m21s	$T.O^*$	>107x <0.01	$T.O^*$	>107x <0.01
SSL020	9m16s	$T.O^*$	>39x <0.01	$T.O^*$	>39x <0.01
TIF001	9m43s	$T.O^*$	>37x <0.01	$T.O^*$	>37x <0.01
TIF002	9m58s	$T.O^*$	>36x <0.01	$T.O^*$	>36x <0.01
TIF009	9m49s	$T.O^*$	>37x <0.01	$T.O^*$	>37x <0.01
XML009	13s	$T.O^*$	>1662x <0.01	$T.O^*$	>1662x <0.01
<b>Mean speedup</b>			128x		142x
<b>Median speedup</b>			37x		37x

**Table 15:** Mean time to reach Fuzzer Test Suite targets for each tested fuzzer’s undirected counterpart over 20 trials. See Table 3 for the full caption.

Bug ID	MC <sup>2</sup> Time	AFL		Angora	
		Time	(x) (p)	Time	(x) (p)
ttgload.c:1710	1s	1s	1x 0.07	1s	1x 0.07
ttinterp.c:2186	9m57s	$T.O^*$	>36x <0.01	24m	2x <0.01
cf2intrp.c:361	58s	40m	41x <0.01	$T.O^*$	>372x <0.01
jdmarker.c:659	32s	1h10m	131x <0.01	1h15m	141x <0.01
pngrutil.c:139	1s	1s	1x 0.07	1s	1x 0.07
pngrutil.c:3182	28s	3m20s	7x <0.01	1m22s	3x <0.01
pngread.c:738	1s	1s	1x 0.07	1s	1x 0.07
pngrutil.c:1393	51s	$T.O^*$	>424x <0.01	$T.O^*$	>424x <0.01
<b>Mean speedup</b>			80x		118x
<b>Median speedup</b>			22x		3x

**Table 12:** Mean preprocessing times over 10 trials for the Magma and Fuzzer Test Suite benchmarks.

Library	MC <sup>2</sup>	AFLGo	ParmeSan	CFG Nodes
libpng (Magma)	54s	1m52s	32s	6940
libtiff (Magma)	55s	10m39s	33s	15485
libxml2 (Magma)	2m41s	24m08s	8m18s	65735
openssl (Magma)	5m11s	1h31m11s	58m15s	95949
php (Magma)	3m21s	14h20m09s	N/A	371648
poppler (Magma)	3m27s	2h28m09s	2m26s	71591
libjpeg (FTS)	7s	1m45s	32s	11173
libpng (FTS)	38s	54s	31s	5257
freetype2 (FTS)	1m15s	12m07s	38s	28662
Arithmetic mean	2m04s	2h07m53s	8m58s	74716
Median	1m15s	12m07s	35s	28662

**Table 13:** Average proportion of Monte Carlo Executions with Exceptions on the Magma benchmark over 10 trials.

Library	Proportion of Executions with Exceptions (%)
libpng	0.26%
libtiff	2.31%
libxml2	0.84%
openssl	1.22%
php	4.04%
poppler	2.24%
sqlite3	10.9%
Arithmetic mean	3.11%
Median	2.14%

CFG, so it is difficult to accurately measure this time since preprocessing is conflated with runtime. We instead approximate this time by measuring the time it takes to run over only the initial seed corpus, in which the dynamic CFG is constructed and distances are computed. We emailed the authors to ensure our setup was reasonable and they confirmed that our experimental setup is reasonable given the dynamic CFG component. For MC<sup>2</sup>, we measure the time it takes to perform preprocessing as described in Section 3. Table 12 summarizes the results for both Magma and Fuzzer Test Suite for the bugs targets found in Section 4.1.

## D MONTE CARLO EXECUTION EXCEPTIONS

Since handling a large number of program exceptions can potentially incur high overheads (i.e., context switches from signal handling), in this experiment, we investigate how many times Monte Carlo Execution handles program exceptions. Specifically, we measure the ratio between the number of executions which require Monte Carlo Execution to handle program exceptions to the total number of executions in our Magma evaluation, repeated 10 times to reduce variability.

Table 13 summarizes the results, with 3.11% in arithmetic mean and 2.14% in median for the proportion. This experiment shows that many Monte Carlo Executions do not involve program exceptions and therefore incur low overhead, a finding that better helps explain our speedups.