Structure-aware Residual Risk Analysis

ANONYMOUS AUTHOR(S)

Software testing can never be *exhaustive*; hence, there is always some *residual risk* that an unseen bug exists even after extensive testing. If we model software test generation as a sampling process, we can define the residual risk as the probability that the next generated test input reveals a bug. The residual risk is upper-bounded by the *discovery probability*, i.e., the probability the next test input covers new code, which itself is upper-bounded by the *coverage rate*, i.e., the expected number of new coverage elements that are covered by the next generated input. Previous work introduced the *Good-Turing estimator* as an estimator of coverage rate to overestimate residual risk. However, the Good-Turing approach induces a strong positive bias, which may lead to undue optimism for bug finding, because (i) the coverage rate is only a loose upper bound and (ii) Good-Turing does not account for the dependencies among coverage elements.

In this work, we propose *structure-aware discovery probability estimation* for residual risk analysis, along with two orthogonal optimization methods that enhance its practical applicability. Our estimator addresses prior challenges by directly estimating the discovery probability while accounting for dependencies between coverage elements. We further prove that our estimator is a *consistent* estimator of the discovery probability. A naive implementation of our estimator requires $O(n \cdot b)$ space complexity, where n is the number of executions and b is the number of coverage elements. Our online singleton cluster maintenance optimization reduces this complexity to $O(b)$ while our node removal mechanism further reduces the number of coverage elements b to be tracked without loss of estimation accuracy.

We evaluated the performance of our structure-aware estimation on real-world data. Results demonstrate that our estimator achieves roughly an order of magnitude greater accuracy than the Good-Turing estimator in most cases. For the task of estimating the expected time to achieve new coverage, our estimator maintains an error of less than 500 seconds within the first 12 hours of fuzzing. Furthermore, the node removal mechanism reduces the number of observed nodes by approximately three-quarters, without compromising estimation accuracy.

ACM Reference Format:
Anonymous Author(s). 2024. Structure-aware Residual Risk Analysis. 1, 1 (December 2024), 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software testing can never be *exhaustive*. Hence there is always some *residual risk* that an unseen bug exists even after extensive testing. Specifically, in a testing campaign where no bugs have been found, residual risk refers to the probability that the next test input triggers a bug. In essence, residual risk represents the risk that persists due to the inherent incompleteness of testing. If residual risk is high, testing should continue to uncover hidden bugs. If residual risk is low, continued testing may be inefficient, and resources could be redirected to other tasks, such as initiating a new fuzzing campaign with different seeds or configurations or employing alternative testing techniques. However, it is impossible to know the exact residual risk as the underlying distribution of the testing process is unknown.

Recently, several studies have employed estimators from ecological biostatistics to estimate an upper bound of the residual risk [3**?** , 4]. Böhme [3] introduced the statistical framework STADS, which models software testing as a sampling process from an unknown Bernoulli Product distribution. In this framework, classes represent coverage elements, such as basic blocks or paths, and each sample corresponds to a set of these coverage elements exercised during execution with a generated test input. Böhme demonstrated that the residual risk is bounded by the *discovery probability*, which is the probability of encountering an unseen class in the next sample. For the

2024. ACM XXXX-XXXX/2024/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fixed sampling distribution (e.g., a black-box fuzzing), studies including Böhme's have employed the *Good-Turing* estimator [10] to estimate the discovery probability. The Good-Turing estimator estimates the *coverage rate*, i.e., the expected number of unseen classes in the next sample, which serves as an upper bound on the discovery probability. Developers can safely decide to end testing if the estimated residual risk is below a specified threshold.

However, the existing residual risk analysis relying on the Good-Turing estimator has two inherent challenges. First, because the Good-Turing estimator measures the *coverage rate*, it introduces two layers of overestimation–from coverage rate to discovery probability, and from discovery probability to residual risk–which often result in significant overestimation. Second, the Good-Turing estimator assumes independence between the classes, which does not reflect the interdependencies within software. Certain coverage elements or bugs are only reachable if other specific blocks are reached first. These limitations may lead to overly optimistic estimates about discovering new bugs, potentially resulting in substantial resource waste.

In this work, we propose structure-aware discovery probability estimation for residual risk analysis, along with two orthogonal optimization methods that enable practical use of the estimator. Recognizing structural aspects of the software, we introduce a new estimator that directly estimates the discovery probability. Our estimator accounts for the dependencies between the coverage elements as indicated by their co-occurrence within the same sample. We further prove that our estimator is a consistent estimator[1] of the discovery probability. To make the discovery probability estimation practical, we design two orthogonal optimizations. Online singleton cluster maintenance eliminates the need to record the covered elements for each execution, reducing the space complexity of the discovery probability estimation from $O(n \cdot b)$ to $O(b)$, where n is the number of executions and b is the number of coverage elements. The structure-aware node removal mechanism reduces the number of coverage elements b to be observed in advance using the control-flow graph of the software, while preserving the accuracy of the estimator.

We evaluate our structure-aware estimation on real-world data, which demonstrates the effectiveness of our approach. We first assess the accuracy of the structure-aware estimator in estimating the discovery probability, comparing it to the Good-Turing estimator, a structure-ignorant alternative, using fuzzing data from benchmark software provided by FuzzBench [17]. Compared to the empirically computed discovery probability values (used as ground truth), our estimator shows approximately an order of magnitude greater accuracy than the Good-Turing estimator for the majority of the cases. In terms of estimating the expected time to achieve more coverage, our estimator has an error under 500 seconds within the first 12 hours of the fuzzing. Subsequently, we evaluate how much the node removal mechanism can reduce the cost of the discovery probability estimation. On the Google workload traces data [1], which has 100K to several million nodes, we show that the node removal mechanism can reduce roughly three-quarters of the nodes to be observed while preserving the accuracy of discovery probability estimation. The node removal mechanism is also efficient, requiring only about half an hour to process control-flow graphs with up to 2 million nodes.

The contributions of this paper are summarized as follows:

- We identify key challenges in existing residual risk analysis that rely on the Good-Turing estimator and propose structure-aware discovery probability estimation to tackle these challenges.
- We establish and evaluate the performance of our estimator. Specifically, we prove that the estimate approaches the estimand as the number of samples increases (consistency) and

[1] In statistics, a consistent estimator is an estimator that, as the number of data points used increases indefinitely, the resulting sequence of estimates converges in probability to the estimand. Check Sec. 3.2 for the formal definition.

empirically find that our estimator is approximately an order of magnitude more accurate than the Good-Turing estimator in most cases.
- To make discovery probability estimation practical, we design two orthogonal optimization methods: online singleton cluster maintenance, which removes the need to record observed coverage elements in each execution, and a node removal mechanism, which reduces the number of coverage elements to observe while maintaining estimation accuracy.
- We publish tools, data, and analysis scripts.

The paper is organized as follows. In Sec. 2, we provide the background of the residual risk analysis, the Good-Turing estimator used in the existing residual risk analysis, and the challenges caused by the structure-ignorant estimation. Motivated by the evidence of the challenges in the software testing, we suggest the structure-aware discovery probability estimation in Sec. 3, along with the proof of the estimator as the discovery probability estimator. In Sec. 4, we suggest the optimization methods for the structure-aware discovery probability estimation. For the evaluation, we design the experiments in Sec. 5 and present the results in Sec. 6. Following the threats to validity in Sec. 7 and the related work in Sec. 8, we conclude the paper with the discussion in Sec. 9.

Every code and data used or reproduced in this paper is available in

https://anonymous.4open.science/r/struct-disc-prob-7795.

2 Background: Extrapolation of Software Testing and Residual Risk Analysis

As Dijkstra famously said, "Testing shows the presence, not the absence, of bugs"; software testing can, therefore, never be exhaustive. Software testing is a matter of *trade-off*: the more testing is done, the more bugs are found, but at the cost of increased resource consumption. Thus, questions like "*How much can the software be tested?*," "*Have we reached the limits of what testing can achieve?*," and "*How quickly are we approaching those limits?*" are fundamental to the testing process. Among these questions, *residual risk analysis* seeks to estimate the probability that the next input[2] will trigger a bug that has not yet been found. If residual risk is high, testing should continue to uncover hidden bugs. Conversely, if residual risk is low, further testing may be inefficient, and resources might be better allocated elsewhere. Yet, exactly knowing the residual risk is, unfortunately, a chicken-and-egg problem: it can only be known if we know which inputs will trigger the unknown bugs, which is why we are testing the software in the first place.

Software Testing as a Sampling Process. While it is impossible to know the exact residual risk, recent studies have confronted this challenge by estimating the upper bound of the residual risk, and the key to this is modeling software testing as a sampling process. STADS [4], the underpinning framework of recent studies, defines the testing target as the set of classes[3] $S = \{s_i\}_{1 \le i \le b}$ ($b = |S|$) that is the union of the set of coverage elements and bugs in the software $\mathcal{P}$. The result of the test execution $X = run(\mathcal{P}, i) \subseteq S$ with the input i is the set of coverage elements covered by the execution and the bug if triggered. Software testing is then the sampling process of executions $X^n = \{X_1, X_2, \ldots, X_n\}$ from the unknown distribution $\mathcal{D}_\mathcal{P} : 2^S \to \mathbb{R}$, i.e., $\mathcal{D}_\mathcal{P}(X)$, where $X \subseteq S$, is the probability of the random input exercising exactly X.[4] In statistics terms, software testing is the sampling process of the *incidence data*, where each sample X is the subset of the set of classes S, while if the sample is a single class, it is the *abundance data* [5]. Given n sample test executions X^n, the residual risk r is the probability of the next sample X_{n+1} triggering a bug that has not yet been

[2]Here, *input* is used interchangeably with *test case*.
[3]This paper uses the terms 'class' (regarding the context of the statistics), 'coverage element' (regarding the software testing context), and 'node' (regarding the graph theory for the control-flow graph) interchangeably.
[4]In this work, we consider the fixed sampling distribution. In other words, the samples X^n from the distribution are the collection of the random variables X_i that are i.i.d.

found. Two quantities related to the residual risk are defined: the *discovery probability* m and the *coverage rate* U (also known as the discovery rate in applied statistics). The discovery probability m is the probability of the next sample X_{n+1} belonging to any of the unseen classes so far, and the coverage rate U is the expected number of unseen classes in the next sample. Formally, let $S_n \subseteq S$ be the set of classes observed in the n samples, $S_{bug} \subset S$ be the set of bugs, $p_X = \mathcal{D}_{\mathcal{P}}(X)$, and $\mathcal{X}$ be the set of all possible samples X, i.e., $\mathcal{X} = \{X | X \subseteq S, p_X > 0\}$. Then,

$$r = \sum_{X \in \mathcal{X}} p_X \cdot \mathbb{I}(X \setminus S_n \neq \emptyset \wedge (X \setminus S_n) \cap S_{bug} \neq \emptyset), \tag{1}$$

$$m = \sum_{X \in \mathcal{X}} p_X \cdot \mathbb{I}(X \setminus S_n \neq \emptyset), \tag{2}$$

$$U = \sum_{X \in \mathcal{X}} p_X \cdot |X \setminus S_n|, \tag{3}$$

where $\mathbb{I}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise. By definition, the coverage rate U upper-bounds the discovery probability m, and the discovery probability m upper-bounds the residual risk r. Note that if the samples are abundance data, i.e., each sample is a single class, then the discovery probability m is the same as the coverage rate U.

Good-Turing Estimator for Residual Risk Analysis. The *Good-Turing* estimator [10] is primarily used in software testing to estimate the upper bound of the residual risk [3, 4]. Given the samples X^n, the Good-Turing estimator estimates the coverage rate U based on the frequency of the observed classes. To be more specific, let V_1 be the set of singleton classes, i.e., the classes observed only once in X^n,

$$V_1 = \left\{ s_i \;\middle|\; s_i \in S, \sum_{X \in X^n} \mathbb{I}(s_i \in X) = 1 \right\}. \tag{4}$$

The Good-Turing estimator is

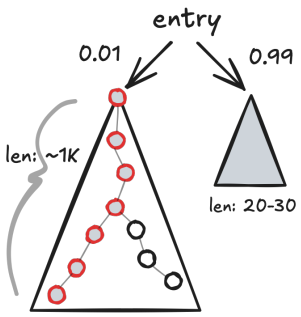
$$\hat{U}_G = |V_1|/n, \tag{5}$$

the ratio of the number of singleton classes to the number of executions. The Good-Turing estimator is known to overestimate the coverage rate U [**?**] and, thus, conservatively overestimates the discovery probability m and the residual risk r. Applying the Good-Turing estimator gives, for the first time, the non-trivial upper bound of the residual risk in the software testing, which can be used to inform the decision-making process in the software testing.

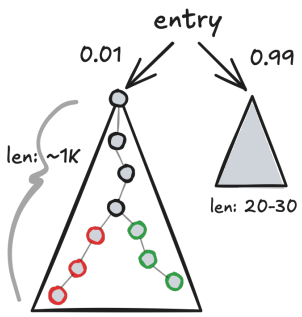
Hereafter, we assume no bug is found within the current testing campaign (X^n), which is the typical assumption in software testing; if the bug is found, the testing campaign will be terminated, and the residual risk is no longer meaningful. We also refer to S as the set of coverage elements regarding statistical estimation purposes as no bug has been found in $X_i \in X^n, X_i \subseteq S$ so far. Similarly, $b = |S|$ is the number of coverage elements in the software.

Challenges of the Good-Turing Estimator as the Discovery Probability Estimator. While they opened the door to foresee the future of software testing and to inform the decision-making process, the current residual risk analysis relying on the Good-Turing estimator has two inherent challenges:

- **Challenge 1.** The Good-Turing estimator estimates the coverage rate U, not the discovery probability m. By nature, two layers of overestimation, U to m and m to r, are inevitable when using the Good-Turing estimate for the residual risk analysis, which may lead to excessive optimism about finding a new bug.
- **Challenge 2.** The Good-Turing estimator assumes the independence between the classes, which is far from the reality of the software. Some coverage elements or bugs are only reachable if another coverage element is reached.



(a) CFG with 100th execution path



(b) CFG with 190th execution path

n	$\lvert V_1\rvert$	$\hat{U}_G$	$\lvert V_1^{=}\rvert$	$\hat{m}_s$
1	23	23.0	1	1.0
...	...		...	
99	4	0.04	2	0.02
100	1021	10.21	3	0.03
101	1021	10.11	3	0.03
...	...		...	
189	1019	5.39	2	0.01
190	1533	8.07	3	0.02

(c) $\lvert V_1\rvert$, $\lvert V_1^{=}\rvert$, $\hat{U}_G$, and $\hat{m}_s$, given X^n

Fig. 1. The motivating example demonstrating the challenges of the Good-Turing estimator for the discovery probability estimation and the effect of the singleton clustering in the structure-aware estimator. The grey nodes are the coverage elements that have been observed, and the red and green nodes denote the singleton classes in the 100th and 190th executions.

The following example illustrates how the Good-Turing estimator overestimates the discovery probability in the software testing due to the challenges. Figure 1a shows the simplified control-flow graph of an imaginary software; the probabilities on the edges show the transition probability between the coverage elements. In this software, the right subtree, whose length of the execution paths is around 20-30, is frequently visited compared to the left subtree, whose length is significantly longer, around 1,000. The first and the second columns of Figure 1c show the hypothetical number of singleton classes $|V_1|$ as the number of executions n increases.

- *Evidence of challenge 1.* Given the single execution, the number of visited coverage elements is 23, which is the number of singletons. Therefore, the Good-Turing estimate $\hat{U}_G = 23.0$, being larger than 1, which is useless for the upper bound of the discovery probability m, the probability.
- *Evidence of challenge 2.* Assume that after 100 executions, the left subtree was first visited. Because the execution paths of the left subtree are significantly longer due to the dependencies between the coverage elements, a significant increase in the number of singleton classes $|V_1|$ is observed; in our example, $|V_1|$ increases from 4 to 1021 when n changes from 99 to 100. It makes a massive jump in the Good-Turing estimate $\hat{U}_G$, which demonstrates the unreliability of the Good-Turing estimate as the (upper bound of the) discovery probability m. Ideally, a discovery probability should not experience a significant change due to a single execution in the long run. Such a vast jump will stay for a while as the left subtree is rarely visited.

3 Structure-aware Residual Risk Analysis

In this work, we suggest a structure-aware residual risk analysis that considers the structural aspect of the software to tackle the challenges of the current residual risk analysis.

3.1 Motivation of the Structure-aware Estimator

To address the challenges mentioned in Sec. 2 of using the Good-Turing estimator for the discovery probability estimation, we suggest the structure-aware estimator that considers the dependencies between the coverage elements. The key insight of our approach is that the Good-Turing estimator becomes the discovery probability estimator for the abundance data, i.e., only a single class is observed in each sample. Instead of considering each singleton class independently, our approach

considers the set of singletons that appear together in the same sample, which we call *singleton clusters*. Singleton clusters are defined over the set of samples and are formally defined as follows:

Definition 3.1 (Singleton Cluster). Given the samples X^n of size n and the set of singletons $V_1 \subseteq S$, the equivalent relation $\equiv$ is defined over V_1 as $s_i \equiv s_j$ if two singletons s_i and s_j appear together in the same sample $X \in X^n$. The singleton clusters $V_1^{\equiv} = \{V_1^1, V_1^2, \ldots, V_1^k\}$, where k is the number of singleton clusters, are the equivalence classes of the equivalent relation $\equiv$.

The set of all new coverage elements appearing together (because of the dependency) in the newly discovered execution path, which are, therefore, the set of all new singleton classes, is regarded as a single singleton cluster in our approach. The fourth column of Figure 1c shows the number of singleton clusters $|V_1^{\equiv}|$ as the number of executions n increases. Here, we can see that, at the 100th execution, the number of singleton clusters $|V_1^{\equiv}|$ only increases by 1, even though many new singleton classes are observed. Note that a singleton cluster may change as the new samples are observed. Figure 1b shows the control-flow graph of the software after the 190th execution, which went through the left subtree for the second time. There is an overlap between the 100th and 190th execution paths. Thus, the singleton cluster that appeared in the 100th execution (red in Figure 1a) becomes smaller after the 190th execution (red in Figure 1b), and the new singleton cluster (green in Figure 1b) is formed. The result shown in the last row of Figure 1c is that the number of singleton clusters $|V_1^{\equiv}|$ increases by 1. In fact, the number of singleton clusters $|V_1^{\equiv}|$ is always is always less than or equal to the number of executions n, as $|V_1^{\equiv}|$ can increase by at most 1 in a single execution. Thus, the structure-aware estimator we suggest,

$$\hat{m}_s = \frac{|V_1^{\equiv}|}{n}, \tag{6}$$

is always less than or equal to 1, which is the property that the discovery probability estimator should have. The last column of Figure 1c shows the structure-aware estimator $\hat{m}_s$ as the number of executions n increases.

3.2 Proof of the Structure-aware Estimator as the Discovery Probability Estimator

In the previous section, we have shown that the structure-aware estimator $\hat{m}_s$ solves the challenges of the current residual risk analysis using the Good-Turing estimator. Yet, we haven't proven that the structure-aware estimator $\hat{m}_s$ is a discovery probability estimator. In this section, we prove that the structure-aware estimator $\hat{m}_s$ is indeed an appropriate estimator for the discovery probability m in terms of its complementary relation to the *generalized sample coverage estimator* [16].

Ma and Chao [16] introduced the *sample coverage* estimator that estimates the proportion of the probability mass of the observed classes. Given the samples X^n of size n, the sample coverage C is the probability of observing only the seen classes in the next sample X_{n+1}, i.e., $C = 1 - m$ by definition. Inspired by the 'seven-character quartet' Chinese poem style, consisting of 7×4 characters and many of the characters frequently *co-occurring* for 'rhyming,' Ma and Chao [16]'s estimator is designed to be applicable in general regardless of the independence between the classes. The sample coverage estimator $\hat{C}$ is defined as follows: Given the samples X^n of size n and the set of singletons $V_1 \subseteq S$,

$$\hat{C} = 1 - \frac{\sum_{i=1}^{n} \mathbb{I}(V_1 \cap X_i \neq \emptyset)}{n}, \tag{7}$$

i.e., the complement of the ratio of the number of executions that contains at least one singleton class to the number of executions. In their work, they proved that the sample coverage estimator $\hat{C}$ is the consistent estimator of the sample coverage C, i.e., when the number of samples $n \longrightarrow \infty$, the sample coverage estimator $\hat{C}$ converges to the sample coverage C in probability: $\hat{C} \xrightarrow{p} C$.

Now, we prove that the structure-aware estimator $\hat{m}_s$ is identical to the complement of Ma and Chao's sample coverage estimator $\hat{C}$, i.e., $\hat{m}_s = 1 - \hat{C}$, which is equivalent to the following theorem.

Theorem 3.2. *Given the samples X^n of size n and the set of singletons $V_1 \subseteq S$, the number of executions that contain at least one singleton class is equal to the number of singleton clusters, i.e.,*

$$\sum_{i=1}^{n} \mathbb{I}(V_1 \cap X_i \neq \emptyset) = |V_1^{\equiv}|. \tag{8}$$

Proof. The proof naturally follows by showing that there is a one-to-one correspondence between the executions that contain at least one singleton class and the singleton clusters. Every singleton in the same singleton cluster appears together in the same sample X as they can appear in precisely one sample (injective). Let X_i be the i-th sample that contains at least one singleton class $s_j \in V_1$. Then, V_1^j, the singleton cluster that includes the singleton class s_j, is the only singleton cluster that appears in sample X_i, as all the singletons in the same singleton cluster appear together in the same sample (surjective). □

Theorem 3.2 thus proves that $\hat{m}_s$ is the consistent estimator of the discovery probability.

4 Optimization of the Structure-aware Estimator

To estimate the discovery probability regarding Ma and Chao [16]'s definition based on the number of executions that contain at least one singleton class (Equ. 7), one needs to maintain which coverage elements are observed in each execution during the whole testing process to check, after all the executions, what are the singletons and which executions contain them. The space complexity for it is, then, $O(n \cdot b)$, where n is the number of executions and b is the number of coverage elements. Such complexity could be a bottleneck for software testing as, typically, an enormous number of executions are performed during the software testing; in the case of the fuzzing [8], the number of executions can be thousands per second. Also, the number of coverage elements b could be excessively large, especially in the industry-level software, which makes the space complexity even worse.

In this section, we suggest two orthogonal optimization methods for the structure-aware discovery probability estimation to make it practical: online singleton cluster maintenance and structure-aware node removal. *Online singleton cluster maintenance* reduces the space complexity of the discovery probability estimation from $O(n \cdot b)$ to $O(b)$, dropping the need for recording the observed coverage elements in each execution. The structure-aware node removal reduces the number of coverage elements to be observed in advance, given the control-flow graph of the software, while preserving the accuracy of the discovery probability estimation.

4.1 Online Singleton Cluster Maintenance

The key to avoiding the need to track observed coverage elements in each execution for counting is to to use the alternative definition of the estimator based on the number of singleton clusters (Equ. 6). Because those two are identical (as we have proven in Theorem 3.2), we could only maintain the singleton clusters to compute the discovery probability estimate.

Alg. 1 shows the algorithm that computes the singleton clusters $V_1^{\equiv}$ from the stream of samples X^n with $O(b)$ space complexity. The algorithm maintains the set of observed classes S_n, the set of singletons V_1, and the set of singleton clusters $V_1^{\equiv}$. While iterating over the stream of samples X^n, the algorithm computes the set of outdated (observed more than once) singletons B and the set of newly observed classes D in each sample X_i. The algorithm then updates the sets S_n, V_1, and $V_1^{\equiv}$ accordingly by removing the outdated singletons from V_1 and $V_1^{\equiv}$ and adding the newly observed

Algorithm 1: Structure-aware discovery probability estimation with $O(b)$ space complexity

Input: X^n: the stream of samples
Output: $V_1^{\equiv}$: the set of singleton clusters

$S_n \leftarrow \emptyset$; $V_1 \leftarrow \emptyset$; $V_1^{\equiv} \leftarrow \emptyset$
for $X_i \in X^n$ **do** /* iterate over the stream of samples */
  $B \leftarrow X_i \cap V_1$; /* the set of observed singletons B */
  $V_1 \leftarrow V_1 \setminus B$; /* remove classes in B from V_1 and $V_1^{\equiv}$ */
  for $V_1^j \in V_1^{\equiv}$ **do**
    $V_1^j \leftarrow V_1^j \setminus B$
  $D \leftarrow X_i \setminus S_n$; /* the set of newly observed classes D */
  if $D \neq \emptyset$ **then**
    $S_n \leftarrow S_n \cup D$; $V_1 \leftarrow V_1 \cup D$; $V_1^{\equiv} \leftarrow V_1^{\equiv} \cup \{D\}$; /* add D to S_n, V_1, $V_1^{\equiv}$ */
return $V_1^{\equiv}$

classes to S_n, V_1, and $V_1^{\equiv}$. The algorithm discards previous samples after each update, so its space complexity depends only on the sizes of the sets $V_1^{\equiv}$, V_1, and S_n, which are bounded by $O(b)$, as each set's maximum size is b (the number of classes).

4.2 Node Removal Mechanism

4.2.1 Principle of Node Removal. To mitigate the cost of observing all the coverage elements in the software, which is the prerequisite for the discovery probability estimation, we suggest a node removal mechanism that can reduce the number of coverage elements to be observed while preserving the accuracy of the structure-aware discovery probability estimation. The principle of node removal is to identify the nodes–coverage elements in the control-flow graph–that can be safely ignored in the observation process for structure-aware discovery probability estimation. For this, we prove the two following theorems that help us to identify the nodes that can be safely ignored.

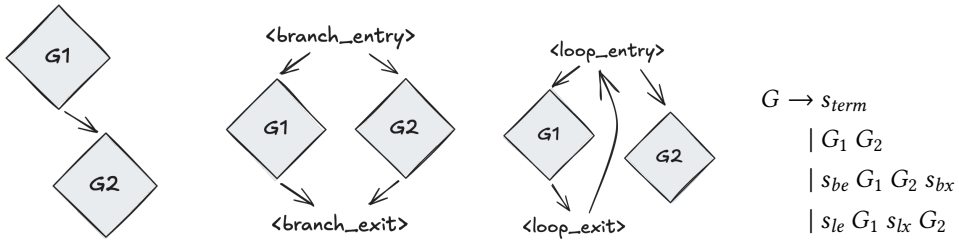
Theorem 4.1. *Let $s_i \in S$ be a coverage element in the control-flow graph G_c. If there exists a non-empty set of coverage elements $T_i \subseteq S, s_i \notin T_i$ such that for any set of samples X^n, if s_i is a singleton from the sample X_i, there is another singleton $s_j \in T_i$ from the sample X_i, then s_i can be safely ignored from the observation for the structure-aware discovery probability estimation.*

The proof of Theorem 4.1 is natural from the definition of T_i; the number of singleton clusters of the samples is the same even after removing s_i since the singleton s_i is always accompanied by another singleton $s_j \in T_i$. The following is the extension of the notion of *dominance/post-dominance* in the control-flow graph and its use for node removal.

Definition 4.2 (Overlaying Set, Post-overlaying Set). Let $s_i \in S$ be a coverage element in the control-flow graph G_c. An *overlaying set* $L_i \subseteq S, s_i \notin L_i$ is the set of coverage elements such that

(1) (set-based dominance) for any path from the entry node to s_i, there is a coverage element $s_j \in L_i$ that appears in the path,
(2) (post-dominance) s_i post-dominates all the coverage elements in L_i.

A *post-overlaying set* $PL_i \subseteq S, s_i \notin PL_i$ is the set of coverage elements such that 1) PL_i is set-based post-dominating s_j, and 2) s_j dominates all the coverage elements in PL_i.



(a) Sequence (b) Branch (c) Loop (d) Grammar

Fig. 2. The components of the control flow of the software and the context-free grammar $\mathcal{M}$ for the language. The subscript in the grammar is a label for the convenience of the explanation.

Theorem 4.3. *Let $s_i \in S$ be a coverage element in the control-flow graph G_c. If there exists an overlaying set or a post-overlaying set of s_i, then s_i can be safely ignored from the observation for the structure-aware discovery probability estimation.*

Proof. Let L_i be the overlaying set of s_i. Then, for any set of samples X^n, if s_i is a singleton, where $X_i \in X^n$ is the sample including s_i, one of the coverage elements in L_i is also executed in the X_i and all the other coverage elements in L_i are not executed in X^n. Thus, the number of singleton clusters of the samples is the same even after removing s_i from the observation. The same argument holds for the post-overlaying set PL_i. □

4.2.2 Recursive Node Removal Mechanism. Based on the principle of node removal, we can identify the nodes that can be ignored from the observation, and still, the accuracy of the discovery probability estimation is preserved. Based on the following proposition, one can retrieve the minimal set of coverage elements required for the structure-aware discovery probability estimation by recursively excluding nodes that are safely ignorable from the observation.

Proposition 4.4. *Given the control-flow graph G and the subgraph $H \subseteq G$, if $s \in H$ is removable from the observation for H, then s is also removable from the observation for G.*

Proof. Let $s \in H$ be removable from the observation. Then, there exists a set $T \subseteq H$ satisfying the conditions in Theorem 4.1. As $T \subseteq H \subseteq G$, s is also removable from the observation for G. □

Here, we propose a recursive mechanism, grounded in the structure of the control-flow graph of the software,[5] to systematically remove these ignorable nodes. Our mechanism accounts for the composition of the control-flow graph through three fundamental control-flow structures: sequence, branch, and loop, as illustrated in Figure 2, along with context-free grammar $\mathcal{M}$ for language $\mathcal{L}_{\mathcal{M}}$ of the control-flow graph. Below, we describe the recursive node removal mechanism for each of the rules in $\mathcal{M}$ based on Theorem 4.3: *if there exists an overlaying set or a post-overlaying set of a node, then the node is removed from the observation.* Let *obs* be the function implementing the node removal mechanism for control-flow graph $G \in \mathcal{L}_{\mathcal{M}}$.

- *Terminal* $\langle G = s_{term} \rangle$: Simply $obs(G) = \{s_{term}\}$.
- *Sequence* $\langle G = G_1\ G_2 \rangle$: If either $obs(G_1)$ or $obs(G_2)$ is a single element set, then $obs(G) = obs(G_2)$ or $obs(G) = obs(G_1)$, respectively. This is because if, without loss of generality,

[5]Note that we only consider the structure of the control-flow graph, not the context or semantics of the software. Based on the semantics of the software, some of the nodes can be additionally safely ignored based on Theorem 4.1. Yet, analyzing the semantics need another level of complexity and overhead, which we would like to avoid for the practicality of the method.

$obs(G_1) = \{s\}$, which means that observing s is sufficient to count the number of singleton clusters in G_1, then s can be safely ignored from the observation as G_2 is the post-overlaying set of s. Otherwise, $obs(G) = obs(G_1) \cup obs(G_2)$.

- *Branch* $\langle G = s_{be}\ G_1\ G_2\ s_{bx} \rangle$, where s_{be} and s_{bx} are the entry and exit nodes of the branch, respectively: $obs(G) = obs(G_1) \cup obs(G_2)$. s_{be} and s_{bx} can be safely ignored from the observation as $G_1 \cup G_2$ is the post-overlaying and the overlaying set of s_{be} and s_{bx}, respectively.
- *Loop* $\langle G = s_{le}\ G_1\ s_{lx}\ G_2 \rangle$, where s_{le} and s_{lx} are the entry and exit nodes of the loop, respectively: $obs(G) = obs(G_1) \cup obs(G_2)$. s_{le} and s_{lx} can be safely ignored from the observation as $G_1 \cup G_2$ and G_1 is the post-overlaying and overlaying set of s_{le} and s_{lx}, respectively. If $obs(G_2) = \{s\}$ is a single element set, then $obs(G) = obs(G_1)$ as G_1 is the overlaying set of s.

We investigate the effectiveness of the recursive node removal mechanism in empirical evaluation in Sec. 5. While the components in the language $L_{\mathcal{M}}$, the sequence, branch, and loop, structure the basic control flow of software, there are cases where the practical control flow of software operates beyond the $L_{\mathcal{M}}$, which we will discuss also in Sec. 5.3.

5 Experimental Design

5.1 Research Questions

We aim to evaluate the effectiveness of the structure-aware estimation in estimating the discovery probability for the residual risk analysis. We also aim to evaluate the effectiveness of the optimization methods in reducing the space complexity of the structure-aware discovery probability estimation. To this end, we formulate the following research questions:

RQ1. *To what extent is the structure-aware estimation more accurate in estimating the discovery probability compared to the structure-ignorant estimation?*

The Good-Turing estimator, the structure-ignorant estimator, is known to overestimate the discovery probability for the abundance data and the classes with dependencies. In contrast, our structure-aware estimator is designed as a consistent estimate of the discovery probability regardless of the independence between the classes. We evaluate how much the structure-aware estimator reduces the estimation error compared to the Good-Turing estimator.

RQ2. *How much does the node removal mechanism reduce the space complexity of the structure-aware discovery probability estimation?*

We suggest two orthogonal optimization methods for the structure-aware discovery probability estimation: online singleton cluster maintenance and structure-aware node removal. The second research question evaluates how many coverage elements can be safely ignored from the observation strictly from the structure of the control-flow graph of the software by the node removal mechanism. The reduction ratio in the number of coverage elements will proportionally affect the overhead of observing residual risk.

Note that the online singleton cluster maintenance is not evaluated in the experiment as it is a (theoretically proved) fundamental optimization that reduces the space complexity from $O(n \cdot b)$ to $O(b)$, dropping the dimension for the number of executions n; the structure-aware estimator will be directly implemented in the online singleton cluster maintenance mechanism.

Table 1. Statistics of the Subjects from FuzzBench

Subject	Size (MB)	#Nodes	Description
Sqlite3	7.9	45,136	Lightweight DB engine
Freetype2	6.76	19,056	Library used to render text onto bitmaps
Libxml2	8.23	50,461	Library for parsing XML documents
Libjpeg	6.4	9,586	Library for handling the JPEG image data format
Jsoncpp	5.96	5,536	Library for manipulating JSON file format

5.2 Metrics and Subjects

For **RQ1**, we measure the accuracy of the estimators by their *absolute errors* compared to the ground truth discovery probability during the black-box fuzzing; at a particular time point t during the fuzzing, we compute the discovery probability estimation $\hat{m}$ using the structure-aware estimator ($\hat{m}_s$) and the Good-Turing estimator ($\hat{m}_G = \hat{U}_G$) and compare them to the ground truth discovery probability. Since the true discovery probability for an arbitrary software is unknown, we use the *empirical discovery probability* as the ground truth: at the time point t, where the number of executions (i.e., samples) and the set of observed coverage elements so far are n_t and S_t, respectively, we compute the empirical discovery probability $\hat{m}_{emp}$ by countering the number of executions that cover the coverage element that is not in S_t from n_t additional auxiliary executions:

$$\hat{m}_{emp} = \frac{\sum_{i=1}^{n_t} \mathbb{I}(Y_i \setminus S_t \neq \emptyset)}{n_t}, \tag{9}$$

where $Y_i \subseteq S$ $(1 \leq i \leq n_t)$ is the set of coverage elements that are covered by the i-th auxiliary execution. The absolute error is then computed as $|\hat{m} - \hat{m}_{emp}|$. For the compatible comparison, we measure the *relative absolute error* as $|\hat{m}_{\{esti\}} - \hat{m}_{emp}|/\hat{m}_{emp}$. In addition, we also measure the *expected time for a new discovery*, which is $\hat{T} = 1/\hat{m}/epm$, where epm is the executions per minute. The expected time for a new discovery will give us insight into the practical utility of the estimators in real-world fuzzing: "*How much time will it take to find a new bug/a new coverage?*"

We used the benchmark subjects from FuzzBench [17], a famous fuzzing benchmarking framework that provides a wide variety of real-world benchmarks at an industrial scale, for **RQ1**. We randomly selected five subjects from FuzzBench: Sqlite3, Freetype2, Libxml2, Libjpeg, and Jsoncpp. The statistics of the subjects are shown in Table 1.

For **RQ2**, we compute the reduction ratio of the number of coverage elements to be observed using the node removal mechanism. The reduction ratio is computed as $|obs(G)|/|S|$, where $|S|$ is the number of coverage elements in the control-flow graph G, and obs is the function implementing the node removal mechanism (Sec. 4.2.2). In addition, we also measure the time the node removal mechanism takes to compute the reduced set of coverage elements.

We used the branch trace data from Google Workload Traces (GWTs) [1] for **RQ2**. GWTs are data logs from workloads running on Google's data centers, shared to support research. These traces provide detailed records of instructions and memory accesses, captured per thread using DynamoRIO's drmemtrace tool. Branch traces are records that capture the behavior of branch instructions (like conditional branches, calls, and jumps) during program execution; each trace contains the sequence of the instruction addresses visited during the program execution. We used GWTs instead of the subjects from FuzzBench for **RQ2** regarding the size of the subjects. The space complexity $O(b)$ matters when the number of nodes in the control-flow graph b is large. While the maximum number of nodes in the subjects from FuzzBench is ~120K, the number of coverage

Table 2. Statistics of the Subjects from Google Workload Traces

Subject	#Trace	#Unique Trace	#Addrs	Trace length (mean [min–max])
Charlie	3,149	2,645	1,085,246	13,733 [5–729,506]
Delta	217	197	135,755	9,939 [5– 75,848]
Merced	5,086	510	646,723	12,609 [5–370,512]
Whiskey	1,266	858	2,112,312	18,745 [5–493,070]

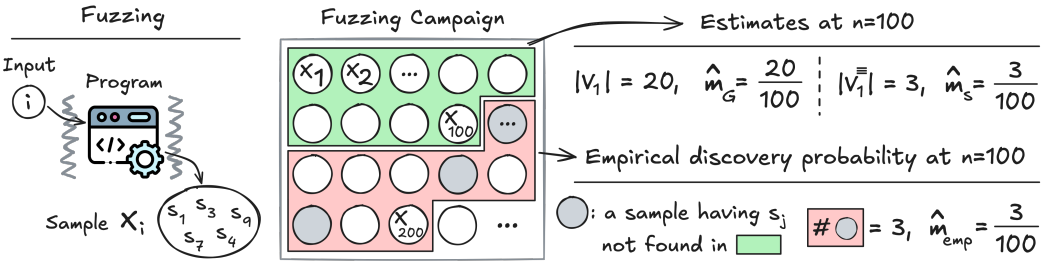


Fig. 3. Figure illustrating how the estimates and the ground truth discovery probability are computed. The green area shows the time point t where the estimates are computed ($n_t = 100$). The red area shows the additional auxiliary executions used to compute the empirical discovery probability.

elements in the branch traces from GWTs is up to ~2M, which is more suitable for evaluating the space complexity reduction. The statistics of the subjects are shown in Table 2.

5.3 Implementation and Setup

We modify the AFL++ [8] fuzzer to implement the blackbox fuzzing. We turned off the coverage-guidance feature of AFL++ by defining the macro `IGNORE_FINDS`. We also discarded the deterministic stages of AFL++, which deterministically attempt to mutate the input in a predefined order, and the position, thus, violates the i.i.d. sampling assumption.

We implement the structure-aware discovery probability estimation and the Good-Turing estimation for **RQ1** on the modified AFL++. During the fuzzing, the number of singletons and singleton clusters is computed using the online singleton cluster maintenance mechanism (Alg. 1). To avoid the underestimation of the discovery probability to the zero probability by the Good-Turing estimator and the structure-aware estimator when either the number of singletons or the number of singleton clusters is zero, respectively, we set the number of singletons and the number of singleton clusters to 1 if they are zero. To compute the empirical discovery probability at the time point t with n_t executions, the next n_t executions are used as the auxiliary executions to count the number of executions that cover the coverage element that is not in the observed set until the time point t. Similarly, to avoid the zero probability in the empirical discovery probability, we set the empirical discovery probability to the minimum non-zero empirical discovery probability estimated during the entire fuzzing campaign if the empirical discovery probability is zero. Figure 3 illustrates how the estimates and the ground truth discovery probability are computed. The experiments for **RQ1** are conducted on a docker container with 64 cores of AMD EPYC 7713P @ 2.0GHz and 251GB memory. To avoid selection bias, we repeat the fuzzing with the same configuration 20 times for each subject.

In **RQ2**, we imagine the traces from GWTs represent the traces from the real-world usage profile of the software; thus, they can be regarded as samples from the software testing under the in-production environment. We approximate the true sampling distribution of the software testing as the empirical probability distribution of the branch traces from GWTs. While GWTs do not provide the control-flow graph of the software, we construct the control-flow graph from the branch traces; unique instruction addresses are regarded as the coverage elements, and the transition between the instruction addresses in the branch traces are regarded as edges between the coverage elements. The node removal mechanism is then applied to the constructed control-flow graph to compute the reduced set of coverage elements. To avoid the significant gap between the approximated sampling distribution and the true sampling distribution, we limit the number of samples from the approximated sampling distribution up to the number of given branch traces.

Several heterogeneous control-flow structures could lie beyond the language $\mathcal{L}_{\mathcal{M}}$ we defined in Sec. 4.2.2. Those structures include multiple entry and exit points, function calls, and structure-breaking jumps. We adapt the node removal mechanism while implementing it to handle those structures. For instance, the multiple entry and exit points can be handled by adding the meta-entry and meta-exit that connect all the entry and exit points, respectively. The function calls and inter-procedural edges can be treated as the loop structure since the body of the function is encapsulated as the loop body; therefore, the same mechanism can be applied. When the structure-breaking jumps are identified, we conservatively maintain the observable coverage elements in the observation set to avoid losing the information on singleton clusters. The node removal mechanism experiments are conducted on the server with the same spec as the experiments for **RQ1** with a single core.

6 Results

6.1 RQ1: Accuracy of the Structure-aware Estimation

The upper part of Figure 4 shows the discovery probability estimation result of the structure-aware estimator (green line) and the Good-Turing estimator (orange line), along with the empirical discovery probability (blue line), the ground truth of our experiment. The structure-aware estimator consistently provides an accurate estimation of the discovery probability compared to the Good-Turing estimator. In most cases, the area of the 95% confidence interval of the structure-aware estimator substantially overlaps with the ground truth discovery probability, i.e., what is empirically observed in the future fuzzing process. In contrast, it is clearly shown that the Good-Turing estimator overestimates the discovery probability compared to the ground truth discovery probability. Such overestimation demonstrates the challenge of using the Good-Turing estimator for the discovery probability estimation we have discussed in Sec. 2.

To assess estimator accuracy over the entire fuzzing campaign, we compute the average relative absolute error across all time points. For each subject–Sqlite3, Freetype2, Libxml2, Libjpeg, and Jsoncpp–the mean relative absolute error (averaged over 20 repetitions) of the structure-aware estimator is 0.23, 0.20, 0.41, 0.12, and 3.87, respectively. In comparison, the Good-Turing estimator yields a mean relative absolute error of 10.01, 2.63, 3.56, 1.82, and 5.87, showing a significant improvement for the structure-aware estimator. In general, the difference tends to be more pronounced for larger subjects. Except for Jsoncpp, the structure-aware estimator achieves roughly an order of magnitude lower relative absolute error than the Good-Turing estimator. Both estimators perform worse for Jsoncpp than for the other subjects, as will be discussed later in this section.

Table 3 presents records from a single fuzzing run on the Libxml2 and Jsoncpp subjects. For each time point t, it shows the number of discovered coverage elements S_t, singletons $|V_1|$, singleton clusters $|V_1^{\equiv}|$, and #New Disc., which represents the number of executions covering coverage elements not in S_t among the auxiliary executions. #New Disc. is the numerator in the empirical

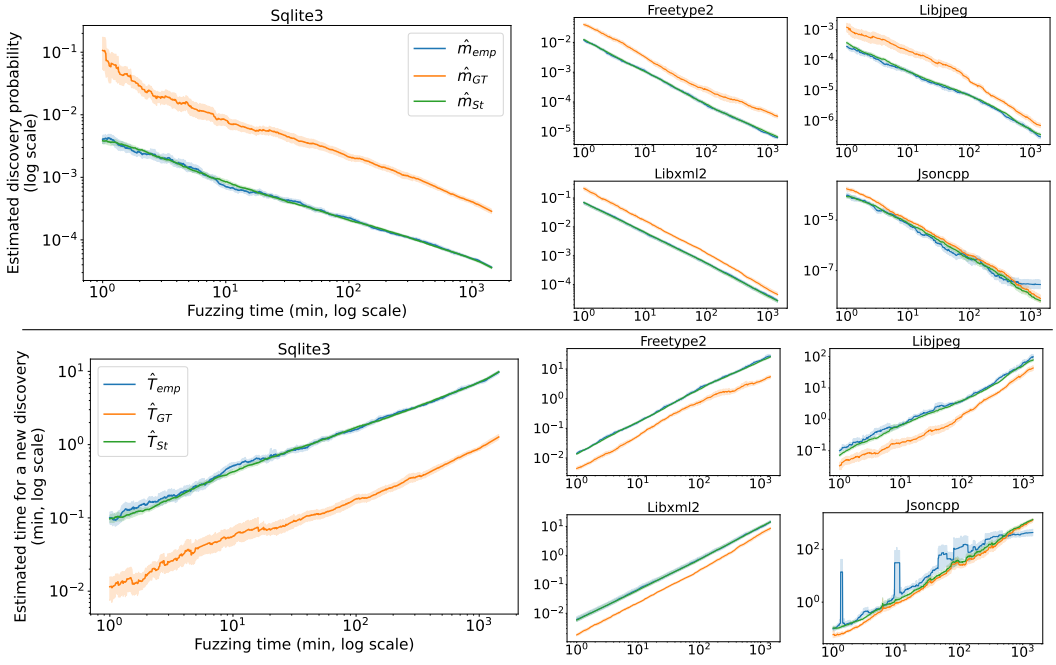


Fig. 4. Figures showing the estimation of discovery probability (above) and expected time for a new discovery (below) for FuzzBench subjects. $\hat{m}_s$ ($\hat{T}_s$) and $\hat{m}_G$ ($\hat{T}_G$) represent the estimates of discovery probability (expected time) from the structure-aware and Good-Turing estimators, respectively, while $\hat{m}_{emp}$ ($\hat{T}_{emp}$) represents the empirical values, used as ground truth, computed from auxiliary executions. The x-axis shows time points during fuzzing, measured in minutes on a logarithmic scale. Lines show averages over 20 repetitions, with shaded areas indicating 95% confidence intervals.

Table 3. The records from a single fuzzing run on the Libxml2 and Jsoncpp subjects. For each time point t, it shows the number of discovered coverage elements S_t, singletons $|V_1|$, singleton clusters $|V_1^{\equiv}|$, and #New Disc., which represents the number of executions with coverage elements not in S_t among the auxiliary executions. (Left: Libxml2, Right: Jsoncpp)

t	S_n	$\lvert V_1\rvert$	$\lvert V_1^{\equiv}\rvert$	#New Disc.
1m	6356	511	195	164
10m	7654	452	192	136
1h	8319	325	152	153
6h	8861	243	129	136
12h	9048	235	133	100

t	S_n	$\lvert V_1\rvert$	$\lvert V_1^{\equiv}\rvert$	#New Disc.
1m	1131	8	5	25
10m	1197	11	9	5
1h	1211	4	3	4
6h	1221	3	2	2
12h	1224	1	1	2

discovery probability computation, similar to $|V_1|$ and $|V_1^{\equiv}|$, which serve as numerators in each estimator, all sharing the same denominator n, the number of executions. Thus, the closeness to #New Disc. indicates estimator accuracy. The record of Libxml2 shows the big difference between the number of singletons and the number of singleton clusters, explaining the significant difference in the estimation of the discovery probability between the estimators and the closeness between the number of singleton clusters and #New Disc., illustrating the accuracy of the structure-aware estimator.

Table 4. The absolute error of the discovery probability estimation for the subjects from FuzzBench across different time intervals during fuzzing. $\hat{T}_{emp}$ represents the mean expected time for a new discovery within each interval, measured in minutes. *ErrG* and *Errs* denote the mean absolute error, defined as $Err_{\{esti\}} = |\hat{T}_{emp} - \hat{T}_{\{esti\}}|$, for the expected time for a new discovery estimated by the Good-Turing estimator and the structure-aware estimator, respectively. The values in parentheses indicate the mean relative absolute error, calculated as $Err_{\{esti\}}/\hat{T}_{emp}$. All values are averages from 20 repetitions.

Duration	Sqlite3			Freetype2			Jsoncpp		
	$\hat{T}_{emp}$	Err_G	Err_s	$\hat{T}_{emp}$	Err_G	Err_s	$\hat{T}_{emp}$	Err_G	Err_s
(0m, 10m)	0.23	0.20 (.86)	0.07 (.34)	0.06	0.04 (.70)	0.01 (.18)	1.75	1.54 (.50)	1.53 (.53)
(10m, 1h)	0.79	0.71 (.88)	0.16 (.21)	0.53	0.31 (.59)	0.09 (.17)	17.38	13.86 (.50)	13.49 (.49)
(1h, 6h)	2.22	1.99 (.89)	0.28 (.14)	3.61	2.45 (.65)	0.76 (.20)	135.22	110.77 (.88)	106.30 (.97)
(6h, 12h)	4.49	3.97 (.88)	0.53 (.12)	10.50	7.90 (.74)	2.24 (.21)	329.26	202.91 (.74)	196.70 (.80)
(12h, 24h)	7.30	6.37 (.87)	0.87 (.12)	20.98	16.84 (.80)	4.49 (.20)	391.22	504.39 (2.10)	549.57 (2.27)

Duration	Libxml2			Libjpeg		
	$\hat{T}_{emp}$	Err_G	Err_s	$\hat{T}_{emp}$	Err_G	Err_s
(0m, 10m)	0.03	0.02 (.63)	0.00 (.11)	0.33	0.24 (.69)	0.13 (.39)
(10m, 1h)	0.19	0.12 (.59)	0.02 (.10)	1.69	1.34 (.76)	0.58 (.32)
(1h, 6h)	1.31	0.73 (.53)	0.17 (.12)	6.96	4.30 (.63)	1.56 (.23)
(6h, 12h)	4.69	2.27 (.45)	0.57 (.12)	27.23	15.96 (.56)	7.56 (.26)
(12h, 24h)	9.75	3.85 (.36)	1.51 (.16)	63.52	34.56 (.49)	20.28 (.30)

As mentioned earlier, while the structure-aware estimator outperforms the Good-Turing estimator significantly for most subjects, there is little difference between the two for Jsoncpp. This is likely because Jsoncpp has the smallest number of nodes, resulting in shorter execution traces that capture fewer dependencies between coverage elements. Consequently, the numbers of singletons and singleton clusters are similar, as shown on the right side of Table 3. Additionally, the Jsoncpp subject reaches saturation in discovered coverage elements faster than the other subjects (illustrated by the plateau in the rightmost part of the Jsoncpp plot in Figure 4), which further minimizes the difference between singletons and singleton clusters.

The expected time for a new discovery, as shown in the lower part of Figure 4, also reflects the accuracy of the estimators: the structure-aware estimator is significantly closer to the ground truth discovery probability than the Good-Turing estimator. This result highlights the practical utility of these estimators in real-world fuzzing scenarios. Suppose developers decide to stop fuzzing when the expected time for a new discovery exceeds a certain threshold. With the Good-Turing estimate, developers may waste resources by running the fuzzing process longer than necessary due to overconfidence in the fuzzing progress (i.e., in discovering new behaviors or bugs). In contrast, the structure-aware estimator provides a more accurate measure of fuzzing progress, enabling developers to stop fuzzing at an optimal time, thereby minimizing resource waste.

The results for the expected time of a new discovery are further analyzed in Table 4. This table shows the absolute error of the discovery probability estimation for FuzzBench subjects at different time intervals during fuzzing. The expected time for a new discovery, $\hat{T}_{emp}$ represents the average expected discovery time across each interval, measured in minutes. Err_G and Err_s denote the mean absolute error, calculated as $Err_{\{esti\}} = |\hat{T}_{emp} - \hat{T}_{\{esti\}}|$, for estimates by the Good-Turing and structure-aware estimators, respectively. Values in parentheses indicate the mean relative absolute error, $Err_{\{esti\}}/\hat{T}_{emp}$. All values are averaged over 20 repetitions.

The table shows that, except for Jsoncpp, the structure-aware estimator maintains an error of less than 500 seconds in the expected time for a new discovery within the first 12 hours of fuzzing.

Table 5. Results of the node removal mechanism for subjects from GWTs. The table displays the number of nodes in the original control-flow graph, the number of nodes in the reduced control-flow graph, the proportion of nodes remaining, the time taken by the node removal mechanism to compute the reduced set of coverage elements, and the processing speed of the node removal mechanism.

Subject	#Nodes (original)	#Nodes (reduced)	Ratio	Time (minute)	Speed (node/sec)
Charlie	1,085,246	264,084	24%	13.53	1,336.84
Delta	135,755	29,939	22%	0.31	7,298.66
Merced	646,723	191,233	30%	10.02	1,075.72
Whiskey	2,112,312	491,261	23%	31.40	1,121.18

The structure-aware estimator's relative absolute error remains around 20% (under 40% in the worst case, excluding Jsoncpp) of the empirical discovery time and is consistent throughout the fuzzing duration. In contrast, the Good-Turing estimator has a relative absolute error exceeding 50% in most cases. For both estimators, the error increases in the (12h, 24h) interval for Jsoncpp, likely due to the saturation of fuzzing progress.

Below is the summary of the results of **RQ1**:

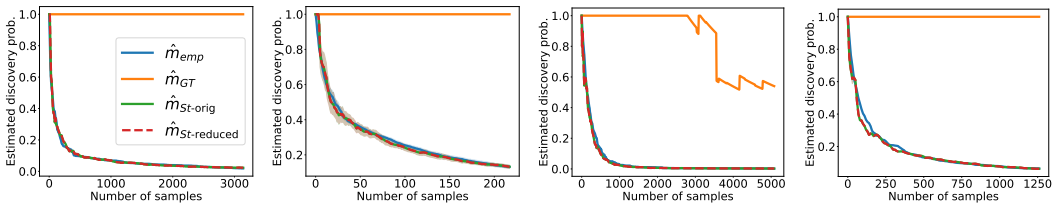
Answer to RQ1: The structure-aware discovery probability estimation provides an accurate assessment of the discovery probability: except for the small-sized subject, the Good-Turing estimator significantly overestimates the discovery probability, whereas the structure-aware estimator achieves a relative absolute error approximately an order of magnitude lower. For the expected time to a new discovery, the structure-aware estimator shows a relative absolute error of around 20% compared to the empirical discovery time for most subjects, underscoring the practical utility of these estimators in real-world fuzzing.

6.2 RQ2: Effect of the Optimization

Table 5 presents the results of the node removal mechanism for the subjects from GWTs. The table includes the number of nodes in the original and reduced control-flow graphs, the proportion of nodes remaining after node removal, and the time taken to compute the reduced set of coverage elements.

Results indicate that the node removal mechanism significantly reduces the control-flow graph size, decreasing nodes by 70–78%, with an average reduction of 75%. The mechanism is also efficient, processing up to 2 million nodes in roughly half an hour, equivalent to a speed of about 1,100 nodes per second.

To ensure the safety of the node removal mechanism, we conducted discovery probability estimations for the GWT subjects using both the full and reduced node sets. Execution samples follow the empirical distribution of branch trace frequencies in the GWTs. To avoid sampling bias while considering the empirical distribution as the approximation of the true distribution, we matched the number of samples to those given by GWTs. For completeness, we also conduct the discovery probability estimation with the Good-Turing estimator. Each experiment was repeated 20 times, and results are shown in Figure 5. Findings confirm that the singleton clusters computed from the reduced node set is consistent with with those from the full node set. The result also displays that the Good-Turing estimator hardly provides the reasonable estimation of the discovery probability (often reporting values near 1.0), reflecting a situation where the singleton count exceeds sample size—a frequent occurrence in large software with extensive coverage elements.



(a) Charlie (b) Delta (c) Merced (d) Whiskey

Fig. 5. The discovery probability estimation for the subjects from GWTs. The x-axis is the number of samples, and the y-axis is the discovery probability. $\hat{m}_{emp}$ (blue line) represents the empirical discovery probability, used as the ground truth, computed from the auxiliary executions. $\hat{m}_{GT}$ (orange line), $\hat{m}_{St\text{-}orig}$ (green line), and $\hat{m}_{St\text{-}reduced}$ (red dashed line) represent the discovery probability estimated by the Good-Turing estimator, the structure-aware estimator with the original node set, and the structure-aware estimator with the reduced node set, respectively. The lines are the average of the 20 repetitions, and the shaded areas are the 95% confidence intervals.

Below is the summary of the results of **RQ2**:

Answer to RQ2: The node removal mechanism significantly reduces the control-flow graph size by 70–78% in GWT subjects. It is efficient, requiring about half an hour to handle control-flow graphs with up to 2 million nodes. The discovery probability estimation from the reduced node set is consistent with estimations from the full set, validating the safety and efficacy of the node removal mechanism.

7 Threats to Validity

As with any empirical study, our results and conclusions face several threats to validity. A primary concern is *external validity*—the extent to which our findings can generalize to other subjects and tools. To address this, we selected subjects from a widely recognized fuzzer benchmark, FuzzBench [17], which includes a diverse set of real-world software for **RQ1**. To ensure scalability of the node removal mechanism, we chose subjects from the Google Workload Trace [1], which includes traces from real-world software with a large number of coverage elements, for **RQ2**.

Another concern is *internal validity*, or the degree to which our study controls systematic error. To reduce random variation from the fuzzers and enhance statistical power, we ran each experiment 20 times and report the statistical outcomes. To minimize bias between the approximated sampling distribution (derived from the empirical distribution) and the true distribution in **RQ2**, we limited the sample count to match that of the GWTs. Finally, there is a risk of errors in our evaluation scripts. To promote transparency and reproducibility, we have made our scripts and data publicly available.

8 Related Work

Residual Risk Analysis and Reliability of Software Systems. Recently, methods for measuring residual risk have been actively investigated across various approaches to software testing. In blackbox fuzzing, statistical estimators such as Good-Turing and Laplace have been proposed to quantify residual risk [3, 13, 18–20]. Böhme et al. [4] extended these statistical estimations to greybox fuzzing, where adaptive biases shift the sampling distribution during testing. However, while statistical estimators assess residual risk without detailed analysis of program semantics, they do not account for the structure of the program or dependencies between coverage elements, often

leading to inaccurate estimations. In contrast, our structure-aware discovery probability estimation leverages program structure, providing more accurate residual risk estimations.

Whitebox testing uses symbolic execution to systematically explore program paths. For residual risk assessment in whitebox testing, model counting has been proposed to evaluate path conditions in traversed paths [6, 7, 9]. However, model counting can be computationally intensive and may not scale well to large software systems. In comparison, our structure-aware discovery probability estimation is lightweight and scalable for large-scale software.

Our primary focus is on the residual risk of undetected bugs in a program implementation during an ongoing greybox testing campaign. Meanwhile, extensive research has examined methods for quantifying overall software reliability [15]. However, as Filieri et al. [6] observed, these approaches are often defined at the design and architectural levels rather than at the program level.

Other Predictive Analyses in Software Testing. Framing software testing as a statistical problem opens up diverse predictive analyses. Beyond residual risk, Liyanage et al. [14] estimated *reachable coverage*—the number of coverage elements a fuzzer can potentially reach—using statistical methods. Another predictive analysis involves the *extrapolation of the coverage rate*[3**?**], which estimates potential additional coverage within a future time frame. Statistical analysis also extends beyond general progress predictions; for example, **?**] estimated the probability of reaching specific program states that remain unreached. Such predictions also aid in information leakage analysis[11], where statistical estimations quantify information leakage in software systems.

9 Discussion

In this work, we proposed structure-aware discovery probability estimation to provide a better upper-bound estimate of residual risk in software testing. Since execution samples inherently form incidence data—where multiple dependent coverage elements appear together in a single execution—the Good-Turing estimator, which assumes independence between coverage elements, overestimates the discovery probability. Our structure-aware discovery probability estimation accounts for program structure, providing more accurate estimates of the discovery probability. Evaluations with FuzzBench subjects show that the structure-aware estimator reliably produces accurate discovery probability estimates. The two orthogonal optimizations further demonstrate that the estimator is practical for real-world software testing involving large numbers of coverage elements and extensive executions.

Our structure-aware discovery probability estimation is not limited to residual risk analysis; it provides a framework for estimating the probability of observing new behaviors across a range of empirical program analyses. Its main advantage lies in handling incidence data, where multiple classes appear together in a sample due to dependency relations. Beyond residual risk analysis, potential applications include reachability analysis: while current methods [**?**] consider binary reachability of specific program states, single program executions transition through multiple states, creating incidence data. Mutation testing and automated program repair could also benefit from this approach, as they frequently generate new program variants and observe their behaviors. Depending on the behavior space's semantics (e.g., the RIPR model [2, 12] for mutation testing or precondition violation levels in program repair), these can be multidimensional and modeled as incidence data. We anticipate that structure-aware discovery probability estimation could extend to other sampling-based methodologies that handle incidence data or class dependencies.

10 Data Availability

Every code and data used or reproduced in this paper is available in

https://anonymous.4open.science/r/struct-disc-prob-7795.

References

[1] 2024. Google Workload Traces. https://dynamorio.org/google_workload_traces.html.

[2] P. Ammann and J. Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press.

[3] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Trans. Softw. Eng. Methodol.* 27, 2 (June 2018), 7:1–7:52. https://doi.org/10.1145/3210309

[4] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 230–241. https://doi.org/10.1145/3468264.3468570

[5] Anne Chao and Robert K Colwell. 2017. Thirty Years of Progeny from Chao's Inequality: Estimating and Comparing Richness with Incidence Data and Incomplete Sampling. *SORT-Statistics and Operations Research Transactions* (2017), 3–54.

[6] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic PathFinder. In *2013 35th International Conference on Software Engineering (ICSE)*. 622–631. https://doi.org/10.1109/ICSE.2013.6606608

[7] Antonio Filieri, Corina S. Păsăreanu, Willem Visser, and Jaco Geldenhuys. 2014. Statistical Symbolic Execution with Informed Sampling. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 437–448. https://doi.org/10.1145/2635868.2635899

[8] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[9] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/2338965.2336773

[10] I. J. Good. 1953. The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrika* 40, 3/4 (1953), 237–264. jstor:2333344

[11] Seongmin Lee, Shreyas Minocha, and Marcel Böhme. 2024. Accounting for Missing Events in Statistical Information Leakage Analysis. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 217–228.

[12] Nan Li and Jeff Offutt. 2017. Test Oracle Strategies for Model-Based Testing. *IEEE Transactions on Software Engineering* 43, 4 (2017), 372–395. https://doi.org/10.1109/TSE.2016.2597136

[13] B. Littlewood and D. Wright. 1997. Some Conservative Stopping Rules for the Operational Testing of Safety Critical Software. *IEEE Transactions on Software Engineering* 23, 11 (Nov. 1997), 673–683. https://doi.org/10.1109/32.637384

[14] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 371–383. https://doi.org/10.1109/ICSE48619.2023.00042

[15] Michael R Lyu et al. 1996. *Handbook of Software Reliability Engineering*. Vol. 222. IEEE computer society press Los Alamitos.

[16] M.-C. Ma and Anne Chao. 1993. GENERALIZED SAMPLE COVERAGE WITH AN APPLICATION TO CHINESE POEMS. *Statistica Sinica* 3, 1 (1993), 19–34. jstor:24304935

[17] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Esec/Fse 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[18] Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M. Nicol, Branson W. Murrill, and M Voas. 1992. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE transactions on Software Engineering* 18, 1 (1992), 33.

[19] Mariëlle Stoelinga and Mark Timmer. 2009. Interpreting a Successful Testing Process: Risk and Actual Coverage. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. 251–258. https://doi.org/10.1109/TASE.2009.26

[20] Xingyu Zhao, Bev Littlewood, Andrey Povyakalo, and David Wright. 2015. Conservative Claims about the Probability of Perfection of Software-Based Systems. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 130–140. https://doi.org/10.1109/ISSRE.2015.7381807