Dependency-aware Residual Risk Analysis

Seongmin Lee
University of California, Los Angeles
Los Angeles, USA
seongminlee@sigsoft.org

Marcel Böhme
Max Planck Institute for Security and Privacy
Bochum, Germany
marcel.boehme@acm.org

Abstract

However much we test a software system, some *residual risk* of undiscovered bugs always remains. If we model test generation as a sampling process, the residual risk can be defined as the probability that the next test input reveals a bug. This risk is upper-bounded by the *discovery probability (DP)*, i.e., the probability that 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 per test input. Prior work introduced the *Good-Turing estimator (GoTu)* to estimate residual risk via coverage rate. However, we find that GoTu substantially overestimates, leading to undue optimism in bug finding because (i) the coverage rate is only a loose upper bound, and (ii) GoTu ignores *dependencies* among coverage elements.

We propose *dependency-aware DP estimation* for residual risk analysis. Our estimator directly estimates DP *and* accounts for dependencies among coverage elements using *Ma and Chao*'s sample coverage estimation. A naive implementation requires space proportional to the number of coverage elements and executions, which can be prohibitively large. To make it practical, we introduce two optimizations: dependency-aware node removal, which reduces the number of coverage elements to observe, and online singleton cluster maintenance, which eliminates the need to record observed coverage elements in each execution.

A comparison of our estimator and GoTu on real-world software from FuzzBench demonstrates a substantial reduction in estimation error. If we stopped the campaign when the estimate of residual risk falls below a certain threshold, GoTu would lead a tester to waste 7× more time than our estimator before deciding to stop. Our estimator achieves a median absolute error of only one-fifth that of GoTu. Finally, our bug-based analysis shows that our estimator achieves one to two orders of magnitude lower error than GoTu in residual risk estimation.

ACM Reference Format:
Seongmin Lee and Marcel Böhme. 2025. Dependency-aware Residual Risk Analysis. In . ACM, New York, NY, USA, 12 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 will exist 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 [2]. 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 [7].

Recently, several studies have employed estimators from ecological biostatistics to estimate an upper bound of the residual risk [2–4]. Böhme [2] 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 from above by the *discovery probability (DP)*, which is the probability of encountering an unseen class in the next sample. If samples are independent and identically distributed (*iid*, e.g., in blackbox fuzzing), previous works have employed the *Good-Turing estimator (GoTu) [12]* to (over-)approximate the DP. GoTu estimates the *coverage rate*, i.e., the expected number of unseen classes in the next sample, which serves as an upper bound on the DP. Developers can safely decide to end test if the estimated residual risk is below a specified threshold.

$$\underbrace{\text{Residual risk} \leq \text{Discovery Probability}}_{\text{Probabilitity of new event}} \leq \underbrace{\text{Coverage rate}}_{\text{Expected \# of new event}}$$

However, the existing residual risk analysis relying on GoTu has two inherent challenges. First, because GoTu measures the *coverage rate*, it introduces two layers of overestimation–from coverage rate to DP and from DP to residual risk–which can result in significant overestimation. Second, GoTu assumes independence between the classes, which does not reflect the inter-dependencies within the 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 *dependency-aware DP estimation* for residual risk analysis. Our estimator directly estimates DP and accounts for dependencies between coverage elements. Rather than relying on explicit dependency analysis, our method leverages only the statistical properties of sampled executions. The key insight is that dependencies among coverage elements are naturally reflected in their co-occurrence within samples. Our estimator builds on Ma and Chao's sample coverage estimation [22], which models class dependencies through their observed co-occurrence, allowing us to capture structural relationships without program analysis. Our estimator is a consistent estimator[1] of the DP.

Conference'17, Washington, DC, USA
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] In statistics, a *consistent estimator* is an estimator where, as the number of data points used increases indefinitely, the resulting sequence of estimates converges in probability to the estimand. Check Section 3.1 for the formal definition.

A naive implementation of dependency-aware DP estimation requires $O(n \cdot b)$ space, where n is the number of executions and b is the number of coverage elements, both of which can be prohibitively large. To make it practical, we introduce two optimizations. The *dependency-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. *Online singleton cluster maintenance* eliminates the need to record the covered elements for each execution, reducing the space complexity of the DP estimation from $O(n \cdot b)$ to $O(b)$.

We evaluate our dependency-aware estimator on real-world software testing using fuzzing on eight benchmark programs from FuzzBench [23]. Our evaluation focuses on two key aspects: the accuracy of DP estimation and the accuracy of stop-time decisions based on a DP threshold. Results show that our estimator significantly outperforms GoTu in DP estimation, achieving a median absolute error of only one-fifth that of GoTu, supported by statistical significance tests. It also provides accurate stop-time estimates, reducing bias by approximately 7× compared to GoTu, meaning only one-seventh of the testing time is wasted relative to the state of the art. Additionally, our node removal mechanism reduces the number of observed coverage elements by 43%. In the estimation of residual risk (i.e., the probability of finding a bug), our estimator yields 1-2 orders of magnitude lower error than GoTu. When applied to greybox fuzzing—where the bug-finding probability changes over time—our estimator achieves up to an order of magnitude lower error in DP estimation compared to GoTu.

The contributions of this paper are summarized as follows:

- We identify key challenges in existing residual risk analysis that rely on GoTu and propose dependency-aware DP estimation to tackle these challenges.
- We establish and evaluate the performance of our estimator. Specifically, we empirically demonstrate that our estimator is 5× more accurate in DP estimation with statistical significance, 7× less testing time is wasted for stop-time decisions, and one to two orders of magnitude more precise in residual risk estimation than GoTu.
- To make DP estimation practical, we design two orthogonal optimization methods: a node removal mechanism, which reduces the number of coverage elements to observe while maintaining estimation accuracy, and online singleton cluster maintenance, which removes the need to record observed coverage elements in each execution.
- We publish implementation, data, and analysis scripts: 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" [7]; 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 [2]. 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, determining the exact residual risk presents a chicken-and-egg problem: to measure it precisely, we would need to know which inputs trigger the unknown bugs—but those bugs are, by definition, unknown, and discovering them is the very purpose of testing.

Software Testing as a Sampling Process. While it is impossible to know the exact residual risk, recent studies have confronted this challenge by estimating an 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 program $\mathcal{P}$. 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 $\mathcal{D}_{\mathcal{P}} : 2^S \to [0, 1]$, i.e., $\mathcal{D}_{\mathcal{P}}(X)$, where $X \subseteq S$, is the probability of the random input exercising exactly X. 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 $r(n)$ is the probability that the next sample X_{n+1} triggers a bug not yet found. Two quantities related to the residual risk are defined: the *discovery probability (DP) m* and the *coverage rate U* (also known as the discovery rate in applied statistics). The DP $m(n)$ is the probability of the next sample X_{n+1} belonging to any of the unseen classes so far. The DP is the *best possible upper bound* on the residual risk when no prior information exists about which classes are bugs. The coverage rate $U(n)$ is the expected number of unseen classes in the next sample. By definition, the coverage rate U upper-bounds the DP m, and the DP m upper-bounds the residual risk r. 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,

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

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

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

where $\mathbb{I}(\cdot)$ is the indicator function that returns 1 if the condition is true and 0 otherwise. For notational simplicity, we omit the argument n in $r(n)$, $m(n)$, and $U(n)$ when it is clear from the context. In a time-dependent context, where the number of samples at time t is n_t, we use $r(t)$, $m(t)$, and $U(t)$ to denote the corresponding quantities evaluated at n_t. Note that if the samples are abundance

[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.

data, i.e., each sample is a single class, then the DP m is the same as the coverage rate U.

Similar to STADS, we initially consider the fixed sampling distribution, i.e., the samples X^n from the distribution $\mathcal{D}_{\mathcal{P}}$ are the collection of the random variables X_i that are i.i.d., during the discussion of the DP estimation (Section 3). Several studies have dealt with the adaptive sampling distribution [4, 20], where the distribution changes as time goes on, while they all relied on the theoretical framework of the fixed sampling distribution. Later, we empirically investigate the performance of our estimator in the presence of an adaptive sampling distribution in Section 5.4.

Good-Turing Estimator for Residual Risk Analysis. The *Good-Turing estimator (GoTu)* [12] is primarily used in software testing to estimate the coverage rate as the upper bound on the DP and, consequently, the residual risk [2, 4]. Given the samples X^n, GoTu 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,

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

GoTu estimates U as $\hat{U}_G = |V_1|/n$, the ratio of the number of singleton classes to the number of executions. GoTu is known to overestimate the coverage rate U [20] and, thus, conservatively overestimates the DP m and the residual risk r. Applying GoTu 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 GoTu for residual risk analysis. 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 GoTu has two inherent challenges:

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

The following example illustrates how GoTu overestimates the DP in software testing due to the challenges. Figure 1a shows a simplified control-flow graph of an imaginary software, where edge probabilities indicate transition likelihoods between coverage elements. In this software, the right subtree, with execution paths of length 20-30, is frequently visited compared to the left subtree, whose paths are significantly longer, around 1,000. The first and the second columns of Figure 1d 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, $\hat{U}_G = 23.0$. As the DP m is a probability and the estimate is larger than 1, the estimate is not useful, at all.
- *Evidence of challenge 2.* Assume that after 100 executions, the left subtree was first visited. Since execution paths in the left subtree are significantly longer due to dependencies between coverage elements, a sharp increase in singleton classes $|V_1|$ is observed; in our example, $|V_1|$ increases from 4 to 1021 when n changes from 99 to 100. This causes a massive surge in the $\hat{U}_G$, demonstrating the unreliability of GoTu as the (upper bound of the) DP m. Ideally, a DP should not experience a significant change due to a single execution in the long run. This sharp increase persists for a while, as the left subtree is rarely visited.

3 Dependency-aware Residual Risk Analysis

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

3.1 Dependency-aware DP Estimation

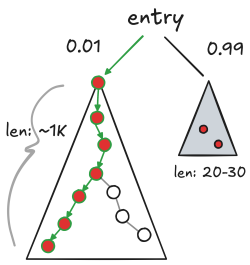
To address the challenges of GoTu, we introduce *dependency-aware DP estimation*. We build and later improve on Ma and Chao's sample coverage estimation [22]. Their approach was inspired by the structure of the 'seven-character quartet' in Chinese poetry, which consists of 7×4 characters, where many characters frequently *co-occur* to maintain rhyme; multiple singleton characters can appear together in a single line, similar to coverage elements in software. Ma and Chao designed an estimation method for sample coverage c, which is the complement of DP ($c = 1 - m$) in abundance data while accounting for class dependencies.

The key approach of Ma and Chao's estimation that makes it applicable regardless of the independence between the classes is to consider a singleton in a *sample-wise manner* rather than a *class-wise manner*. Given the samples X^n of size n, it first identifies the set of singletons $V_1 \subseteq S$. Instead of computing the ratio of the number of singletons $|V_1|$ to the number of samples n as GoTu does, Ma and Chao's estimator identifies the sub-samples $Y_D = \{X_i \mid X_i \in X^n \wedge V_1 \cap X_i \neq \varnothing\}$ that contain at least one singleton and estimates the DP as the ratio of the number of those samples to the number of samples n. Formally, the estimator is computed as

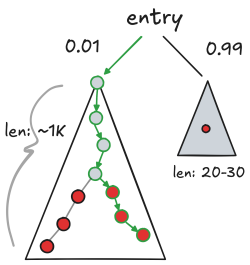
$$\hat{m}_D = \frac{|Y_D|}{n} = \frac{\sum_{i=1}^{n} \mathbb{I}(V_1 \cap X_i \neq \varnothing)}{n}. \tag{5}$$

In their work, Ma and Chao proved that the estimator *consistently estimates* the DP in the abundance data with the dependencies between the classes, i.e., when the number of samples $n \to \infty$, the estimator converges to the DP in probability: $\hat{m}_D \xrightarrow{p} m$.

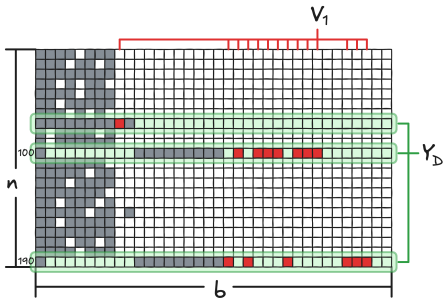
Figure 1 demonstrates how the estimator solves the challenges of GoTu for DP estimation. By definition, the number of sub-samples $|Y_D|$ is always at most the number of samples n. Thus, Ma and Chao's estimator $\hat{m}_D$ is always at most 1 (which is a property we expect from a probability estimator). The set of all new coverage elements appearing together (due to dependencies) in a single execution is regarded as a single execution having a singleton. The fourth column ($|Y_D|$) of Figure 1d, therefore, shows an increase of



(a) Coverage over CFG after 100th execution



(b) Coverage over CFG after 190th execution



(c) Coverage record matrix (n(=190): # samples, b: # coverage elements)

n	$\lvert V_1\rvert$	$\hat{U}_G$	$\lvert Y_D\rvert$	$\hat{m}_D$
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

(d) $\lvert V_1\rvert$, $\hat{U}_G$, $\lvert Y_D\rvert$, and $\hat{m}_D$, given X^n

Figure 1: Example illustrating challenges of GoTu in DP estimation and the effect of the dependency-aware estimator. In (a), (b), and (c), white, gray, and red indicate unvisited, visited, and singleton coverage elements, respectively. In (a) and (b), the green arrow represents the execution path.

1 at the 100th execution, even though many new singletons are observed in the second column ($|V_1|$). Note that the set of singletons evolves as new samples are observed. Figure 1b shows the control-flow graph of the software after the 190th execution, which traversed the left subtree for the second time—overlapping coverage elements are discarded from the singletons, and new ones are introduced. Even though ~500 new singletons are added in the 190th execution (the last row of Figure 1d), the number of sub-samples $|Y_D|$ only increases by 1. The last column of Figure 1d shows Ma and Chao's estimator $\hat{m}_D$ as the number of executions n increases.

Scalability Issue of Ma and Chao's Estimator. Ma and Chao's estimator [22] checks singletons and the executions containing them after all the execution samples are observed. Thus, it must track which coverage elements are observed in each execution throughout testing as the coverage record matrix shown in Figure 1c. Its space complexity is $O(n \cdot b)$, where n is the number of executions and b is the number of coverage elements. This complexity can be a bottleneck for software testing, typically in industry-level software, as b can be excessively large. Moreover, automated software testing, such as fuzzing [10], can execute thousands of tests per second, further exacerbating the space overhead.

3.2 Node Removal Mechanism

To mitigate the space complexity issue and make the DP estimation practical, the first optimization method we suggest is the *dependency-aware node removal mechanism*. This aims to reduce the number of coverage elements b to be observed in advance while preserving the accuracy of the DP estimation. We formally prove which nodes can be safely removed to ensure that the optimization preserves the overapproximation of discovery probability over residual risk. Without this formal justification, arbitrary node removal could reduce $|Y_D|$, leading to an underestimated discovery probability and a false sense of program safety.

Principle of Node Removal. The principle of node removal is to identify the nodes–coverage elements in the control-flow graph–that do not affect the dependency-aware DP estimation even if unobserved; we call these nodes *safely ignorable* from S during observation for the dependency-aware DP estimation.

Definition 3.1 (Safely Ignorable). Let S be the set of coverage elements in program $\mathcal{P}$. We call $s \in S$ as *safely ignorable* if, for all samples $X^n \sim \mathcal{D}_\mathcal{P}$ of some size $n \in \mathbb{N}$, $|Y_D| = |\bar{Y}_D|$, where

- $V_1 \subseteq S$ is the set of all singletons in X^n,
- $Y_D = \{X_i \mid X_i \in X^n \wedge X_i \cap V_1 \neq \emptyset\}$,
- $\bar{X}^n = \{\bar{X}_i \mid X_i \in X^n \wedge \bar{X}_i = X_i \setminus \{s\}\}$,
- $\bar{V}_1 \subseteq S$ is the set of all singletons in $\bar{X}^n$, and
- $\bar{Y}_D = \{\bar{X}_i \mid \bar{X}_i \in \bar{X}^n \wedge \bar{X}_i \cap \bar{V}_1 \neq \emptyset\}$.

The situation where $|Y_D|$ remains unchanged even after removing $s \in S$ from the observation implies that the DP estimate $\hat{m}_D$ is the same, regardless of whether s is observed. The following theorem describes the condition under which a node is safely ignorable.

Theorem 3.2. *Let $s_i \in S$ be a coverage element in program $\mathcal{P}$. For any set of samples $X^n \sim \mathcal{D}_\mathcal{P}$, if s_i appears as a singleton in some sample $X_k \in X^n$, and there always exists $s_j \in S$ such that $s_j \neq s_i$, s_j is also a singleton, and $s_j \in X_k$, then s_i is safely ignorable.*

The proof of Theorem 3.2 follows naturally from the definition of Y_D in Eq. (5). If s_i is always a singleton in X_k whenever s_j is a singleton in X_k, then $X_k \in Y_D$ even after removing s_i from the observation; $|Y_D|$ is still the same, and so is the DP $\hat{m}_D$.

Node Removal Mechanism. Given the principle of node removal, we can identify the safely ignorable nodes based on the *dominance/post-dominance* in the control-flow graph.

Definition 3.3 (Full Dominance, Full Post-dominance). Let $s_i \in S$ be a coverage element in the control-flow graph $G_c = (S, E, s_e, s_x)$ of program $\mathcal{P}$, with a control-flow edge set E, an entry node s_e, and an exit node s_x. Node s_i is a *full dominator* of G_c if it dominates all its direct successors in G_c, i.e., for any $s_j \in \text{succ}(s_i)$, every the paths from s_e to s_j passes through s_i. Node s_i is a *full post-dominator* of G_c if it is a post-dominator of all of its direct predecessors in G_c.

Theorem 3.4. *Let $s_i \in S$ be a coverage element in the control-flow graph G_c of program $\mathcal{P}$. If s_i is a full dominator of G_c, then s_i is safely ignorable.*

Proof. Let s_i be a full dominator with the set of direct successors $\text{succ}(s_i)$. Then, for any set of samples $X^n \sim \mathcal{D}_{\mathcal{P}}$, if s_i is a singleton in some sample $X_i \in X^n$, one of its successors in $\text{succ}(s_i)$ is also executed in X_i for the first time, i.e., it is a singleton in X_i, while all other successors are not executed in X^n. Thus, s_i is safely ignorable based on Theorem 3.2. □

The same argument of Theorem 3.4 holds for the full post-dominator. We implement the node removal mechanism based on Theorem 3.4, where full dominators and full post-dominators are removed from the control-flow graph. The time complexity of the node removal mechanism depends on two steps: $O(e \log b)$ [16], where e and b denote the number of edges and nodes in the control-flow graph, respectively, and $O(b)$ for node removal. Notably, dominance computation can be performed independently for each function in the software, ensuring that the node removal mechanism remains scalable for real-world software testing. The space complexity of the node removal mechanism is $O(e + b)$.

3.3 Online Singleton Cluster Maintenance

The second optimization method is the *online singleton cluster maintenance*, which reduces the space complexity of the DP estimation from $O(n \cdot b)$ to $O(b)$, dropping the need to record the observed coverage elements in each execution. The key idea is to introduce another concept, the *singleton cluster*, which is equivalent to the execution that contains singletons, that are maintainable in $O(b)$ space complexity during the testing process.

Singleton Cluster. The need to maintain the observed coverage elements in each execution arises because Ma and Chao's estimator checks executions with at least one singleton. Yet, *which coverage elements are singletons is unknown in advance* until all samples are observed. To address this, we introduce *singleton clusters*, sets of singletons appearing together in the same sample.

Definition 3.5 (Singleton Cluster). Given the samples X^n of size n and the set of singletons $V_1 \subseteq S$, we define an equivalence relation $\equiv$ over V_1 such that $s_i \equiv s_j$ 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 induced by $\equiv$.

For instance, in Figure 1c, the set of singletons (red cells) in the same row is a singleton cluster. We prove that the number of singleton clusters $|V_1^{\equiv}|$ is the same as the number of executions that contain at least one singleton $|Y_D|$.

Theorem 3.6. *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.,*

$$\sum_{i=1}^{n} \mathbb{I}(V_1 \cap X_i \neq \emptyset) = |V_1^{\equiv}|. \tag{6}$$

Proof. The proof naturally follows by showing that there is a one-to-one correspondence between executions containing 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_{s_i} be the sample containing at least one singleton class $s_j \in V_1$. Then, V_1^i, the singleton cluster that includes the singleton class s_j, is the only singleton cluster that appears in sample X_{s_i}, as all the singletons in the same singleton cluster appear together in the same sample (surjective). □

Algorithm 1: Online singleton cluster maintenance

```
Input: X^n: the stream of samples
Output: V_1^≡: singleton cluster set
S_n ← ∅;  V_1 ← ∅;  V_1^≡ ← ∅
for X_i ∈ X^n do        // iterate over the stream of samples
    B ← X_i ∩ V_1 ;                  // observed class set B
    V_1 ← V_1 \ B ;  // remove classes in B from V_1 and V_1^≡
    for V_1^j ∈ V_1^≡ do
        V_1^j ← V_1^j \ B
    D ← X_i \ S_n ;          // the newly observed classes D
    if D ≠ ∅ then            // Add D to S_n, V_1, and V_1^≡
        S_n ← S_n ∪ D;  V_1 ← V_1 ∪ D;  V_1^≡ ← V_1^≡ ∪ {D}
return V_1^≡
```

Thus, the dependency-aware estimate of the DP in Eq. (5) can be equivalently computed by $|V_1^{\equiv}|/n$, which is the number of singleton clusters divided by the number of executions.

Online Maintenance of Singleton Clusters. While they are identical (Theorem 3.6), the key advantage of singleton cluster-based estimation against the execution-with-singleton approach is that the (number of) singleton clusters can be maintained *online*, i.e., during the testing process, with $O(b)$ space complexity. This is because the change from the singleton clusters in the previous sample to the singleton clusters in the current sample is solely determined by the set of coverage elements in the current sample.

Algorithm 1 presents the algorithm for computing the singleton clusters $V_1^{\equiv}$ from the stream of samples X^n with $O(b)$ space complexity. The algorithm maintains the observed class set S_n, singleton set V_1, and singleton cluster set $V_1^{\equiv}$. While iterating over 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. It 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 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). Given Algorithm 1, dependency-aware DP estimation can be computed with $O(b)$ space complexity, making it practical for real-world software testing.

4 Experimental Design

4.1 Research Questions

We aim to evaluate the effectiveness of the dependency-aware estimation in estimating the discovery probability (DP) for the residual risk analysis. We also aim to evaluate the effectiveness of the optimization methods in reducing the space complexity of

the dependency-aware DP estimation. We use fuzzing as the primary application of the evaluation. To this end, we formulate the following research questions:

RQ1. *To what extent is the dependency-aware estimation more accurate in estimating the DP than the dependency-ignorant estimation?*

The practical use of DP estimation is to obtain an upper bound on the residual risk in SUTs. We evaluate how much the dependency-aware estimator reduces estimation error compared to GoTu. Our evaluation consists of both directly measuring the DP estimation error and assessing its practical impact on the stopping decision of the testing process. Specifically, we consider the most common use case of DP estimation in fuzzing, where the estimation serves as the stopping decision maker: given a DP threshold ϵ, testing stops when the estimated DP reaches this threshold.

RQ2. *How much does the node removal mechanism reduce the space complexity of the dependency-aware DP estimation?*

We propose two orthogonal optimization methods for dependency-aware DP estimation: online singleton cluster maintenance and dependency-aware node removal. The second research question evaluates how many coverage elements are safely ignorable by the node removal mechanism. Reducing the number of coverage elements proportionally decreases the overhead of observing residual risk. Not that online singleton cluster maintenance is directly incorporated into the dependency-aware estimator and is not separately evaluated, as it is a theoretically proven optimization that reduces the space complexity from $O(n \cdot b)$ to $O(b)$, eliminating dependence on the number of executions n. Instead, its impact is reflected in the number of executions during fuzzing.

RQ3. *How accurate is the dependency-aware estimation in predicting residual risk, i.e., the probability of uncovering unseen bugs?*

Here, we directly evaluate the DP estimates in the context of the residual risk, i.e., the probability of finding an undiscovered bug. Naturally, the DP would overestimate the bug-finding probability. However, we argue that accounting for dependencies in the estimation yields a more accurate bug-finding probability estimate.

RQ4. *Does the dependency-aware estimation provide a better estimate than the state-of-the-art estimators for the greybox fuzzing?*

So far, we have focused on blackbox fuzzing, where the test cases are independent and identically (*iid*) distributed samples from a fixed distribution. In this research question, we evaluate the effectiveness of the dependency-aware estimation in greybox fuzzing, where the distribution shifts with each coverage-increasing seed added to the corpus. We modify the GoTu-based greybox fuzzing DP estimator [4] by substituting the number of singleton clusters for the number of singletons in the formula and evaluate its performance against the original.

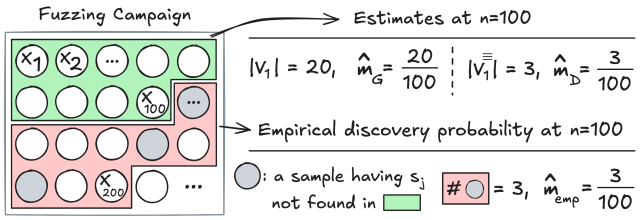


Figure 2: Illustration of DP estimation and ground truth computation. The green area represents samples up to time t used for estimation ($n_t = 100$), while the red area shows auxiliary executions for empirical DP calculation.

4.2 Metrics and Subjects

For **RQ1**, we first measure the accuracy of the estimators by their *absolute errors* compared to the ground truth DP during the blackbox fuzzing; at a particular time point t during the fuzzing, we compute the DP estimation $\hat{m}$ using the dependency-aware estimator ($\hat{m}_D$) and GoTu ($\hat{m}_G = \hat{U}_G$, regarding GoTu as the dependency-ignorant DP estimator) and compare them to the ground truth DP.

Since the true DP for an arbitrary software is unknown, we use the *empirical DP* 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_{n_t}, respectively, we compute the empirical DP $\hat{m}_{emp}(t)$ by countering the number of executions that cover the coverage element that is not in S_{n_t} from n_t additional auxiliary executions, $\{X_{n_t+1}, X_{n_t+2}, \ldots, X_{2n_t}\}$:

$$\hat{m}_{emp}(t) = \frac{\sum_{i=1}^{n_t} \mathbb{I}(X_{n_t+i} \setminus S_{n_t} \neq \emptyset)}{n_t}. \tag{7}$$

Figure 2 illustrates how the estimates and the ground truth DP are computed. The absolute error is then computed as $AE_{\{esti\}}(t) = |\hat{m}_{\{esti\}}(t) - \hat{m}_{emp}(t)|$. For the compatible comparison, we measure the *relative absolute error* as $RE_{\{esti\}}(t) = AE_{\{esti\}}(t)/\hat{m}_{emp}(t)$. The evaluation over the time interval $[0, T]$ is conducted by averaging the error over the time points t: $\int_0^T \{error\}\, dt$. We also perform a statistical significance test on the absolute errors using the two-sided Wilcoxon signed-rank test [28] and rank-biserial correlation [6] to assess the difference between the estimators.

To evaluate the stopping decision, we compare T_{emp}, the time when the empirical DP first reaches $\hat{m}_{thres}$, with T_G and T_D, the times when GoTu and the dependency-aware estimator reach $\hat{m}_{thres}$, respectively. The $\hat{m}_{thres}$ candidates are chosen from 10^{-k} for $k \geq 3$; for each subject, we select those reached at least once in all repeated fuzzing runs. Similar to DP estimation evaluation, we measure the absolute error of the stopping decision as $AE^T_{esti} = |T_{esti} - T_{emp}|$ and the relative absolute error as $RE^T_{esti} = AE^T_{esti}/T_{emp}$.

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 $|S_{rm}|/|S|$ ($|S| = b$), where $S_{rm} \subseteq S$ is the reduced set of coverage elements after the node removal mechanism. We also measure the time the node removal mechanism takes to compute the reduced set of coverage elements. Finally, we compare the fuzzing performance of the fuzzers with and without the node removal mechanism in terms of the number of executions per time. Although reducing the cost of coverage checks

during fuzzing was not our primary goal, the node removal mechanism inherently reduces this overhead, contributing to improved timewise efficiency in fuzzing.

For **RQ3**, we compare the DP estimates with the residual risk, i.e., the probability of finding a bug that has not yet been found. Similar to **RQ1**, we use the empirical residual risk as the ground truth. Let each unique bug $c_i \in C$ $(1 \leq i \leq m)$ be first discovered at the n_i-th execution. The empirical probability of finding bug c_i is then $\hat{\Pr}(c_i) = n_i/n$. The empirical residual risk $\hat{r}_{emp}$ at time point t with n_t executions is given by the sum of the empirical probabilities of finding bugs that have not yet been discovered: $\hat{r}_{emp} = \sum_{i=1}^{m} \mathbb{I}(n_i > n_t) \cdot \hat{\Pr}(c_i)$. We then compute the relative absolute error for the residual risk estimation as $RE^r_{\{esti\}} = |\hat{m}_{\{esti\}} - \hat{r}_{emp}|/\hat{r}_{emp}$.

Böhme et al. [4] suggested the various DP estimators for greybox fuzzing, where the adaptive bias keeps changing the sampling distribution; thus, the existing estimator, GoTu, underestimates the DP. They adjusted GoTu to design the new estimator while handling the adaptive bias. Yet, as dependency ignorance remains in the estimator, we propose the dependency-aware estimator for the greybox fuzzing. We use both estimators suggested by Böhme et al. [4], the Reset estimator and the Mean Local estimator, as baselines for the comparison. Leveraging the fact that the species distribution changes only upon the addition of a new seed, the α-reset estimator considers incidence (i.e., coverage) data only from the point when the α-th last seed was added—effectively using a moving window. A greybox campaign $\mathcal{F}$ is a sequence of blackbox campaigns, $\mathcal{F} = \langle F_1, F_2, \ldots, F_k \rangle$, where k denotes the number of new seeds that have been added to the initial seed corpus throughout the campaign. Consequently, the set of samples X^n generated by $\mathcal{Z}$ is partitioned into k segments, $\langle Z_1, Z_2, \ldots, Z_k \rangle$, where each segment $Z_i = (X_{n_{i-1}+1}, X_{n_{i-1}+2}, \ldots, X_{n_i})$ (with $n_0 = 0$, $n_k = n$) contains the samples generated between the addition of the $(i-1)$-th and i-th seeds (i.e., the sub-campaign F_i). The reset estimator is computed as:

$$\hat{U}_{Reset}(n) = \frac{f_{1,\alpha}}{n}, \tag{8}$$

where $f_{1,\alpha} = |\{s \mid s \in S, \sum_{i=k-\alpha+1}^{k} \sum_{X \in Z_i} \mathbb{I}(s \in X) = 1\}|$ is the number of singletons in the last α segments. The key of the *Mean Local estimator* is to manage the coverage record per seed, estimating the DP if the seed is selected to mutate and then aggregating the estimations to get the final DP.

$$\hat{U}_{MLG}(n) = \sum_{t \in C_n} q_t \cdot \begin{cases} 1 & \text{if } n_t = 0 \\ \frac{1}{n_t+2} & \text{if } |V_1^t| = 0 \\ \frac{|V_1^t|}{n^t} & \text{otherwise.} \end{cases} \tag{9}$$

Eq. (9) shows the Mean Local estimator based on GoTu, where C_n is the set of seeds in the corpus at the point where n executions are done, q_t is the probability of selecting the seed t to mutate, n_t is the number of times the seed t has been selected to mutate, and V_1^t is the set of singletons when the seed t is selected to mutate. In their experiments, Böhme et al. [4] showed that the Mean Local estimator performs best in greybox fuzzing, whereas the Reset estimator at times underestimates the DP. Against these baselines, we design the *dependency-aware Mean Local estimator* by substituting the number of singletons $|V_1^t|$ for the number of singleton clusters $|V_1^{\equiv}|$ in the Mean Local estimator (Eq. (9)). In **RQ4**, we evaluate the dependency-aware Mean Local estimator against the Reset estimator and the Mean Local estimator based on GoTu in the greybox fuzzing, similar to the evaluation in **RQ1**.

For **RQ1, RQ2, and RQ4**, we use benchmark subjects from FuzzBench [23], a widely used fuzzing benchmarking framework that provides diverse real-world benchmarks at an industrial scale. We gather six subjects from FuzzBench previously used in a recent residual risk analysis study [4]: Freetype2, Jsoncpp, Libpcap, Libpng, Libxml2, and Zlib. These are security-critical programs that did not crash within the first 10^9 generated test inputs, ensuring that residual risk analysis remains meaningful. The sizes of these subjects in terms of the number of basic blocks are 1.8K, 8.8K, 10.5K, 14.4K, 46.1K, and 93.9K. As this set contains only two large programs, we decided to add two more subjects Libjpeg and Sqlite3, which have 36.8K and 58.3K basic blocks, respectively, to increase the diversity of our subjects in terms of size. The statistics of the size (b) of these subjects are presented in Table 4. For the bug-based evaluation in **RQ3**, we consider subjects from both FuzzBench and previous studies [4, 30], selecting those where bugs can be found within 24 hours of blackbox fuzzing. In total, we identify four subjects, Assimp (commit ID: 4d451fe), File (2d5f858), Harfbuzz (17863bd), and Libxml2 (99a864a), for the bug-based evaluation.

4.3 Implementation and Setup

We modify the AFL++ [10] fuzzer to implement the blackbox fuzzing. We disable the coverage-guidance features of AFL++ by defining the macro `IGNORE_FINDS`. We also discard its deterministic stages, which mutate inputs in a predefined order and thus violate the *iid* sampling assumption. By default, we use both the node removal mechanism and the online singleton cluster maintenance mechanism (Algorithm 1).

While the expected value of our discovery probability (DP) estimator (Eq. (5)), as well as that of GoTu, is strictly greater than zero for $n > 0$, the corresponding random variables may still evaluate to zero. This is because the number of singletons ($|V_1|$) or singleton clusters ($|V_1^{\equiv}|$) may become zero during fuzzing due to its stochastic nature-an outcome that is not meaningful in our context. To prevent this, we conservatively set the number of singletons ($|V_1|$) or singleton clusters ($|V_1^{\equiv}|$) to 1 when they are zero, respectively, thereby avoiding a zero DP estimate. Similarly, to avoid assigning zero probability in empirical DP, we replace it with the minimum non-zero empirical DP observed during the entire fuzzing campaign whenever it evaluates to zero. As the baseline for greybox fuzzing, we set the hyperparameter α of the Reset estimator to 1, which falls within the recommended range from Böhme et al. [4].

We evaluate the estimation performance on a 24-hour fuzzing campaign; to do so, we run the fuzzing for 48 hours to compute the empirical DP and empirical residual risk. Experiments are conducted in a Docker container with 64 cores of AMD EPYC 7713P @ 2.0 GHz and 251 GB of memory. To avoid selection bias, we repeat fuzzing with the same configuration 20 times per subject for the DP estimation research question (**RQ1**, **RQ4**) and 5 times per subject for other research questions.

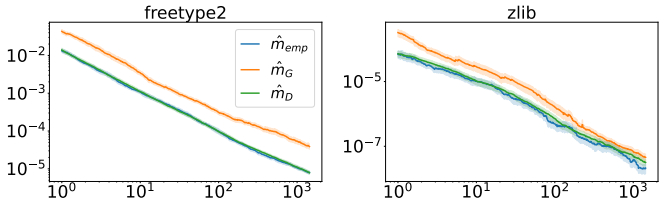


Figure 3: DP estimation for Freetype2 and Zlib. The x-axis (time in minutes) and y-axis are in log scale. Lines represent averages over 20 repetitions, with shaded areas showing 95% confidence intervals.

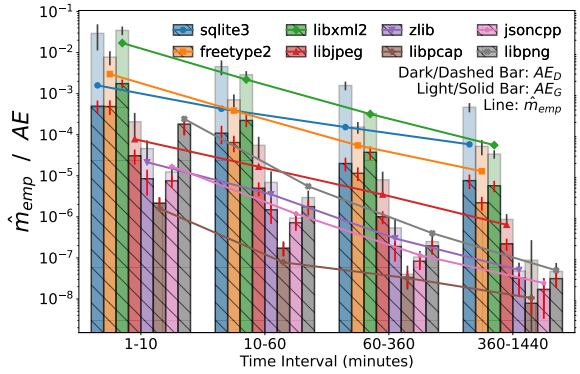


Figure 4: Absolute error of DP estimation over time for FuzzBench subjects. The x-axis (time in minutes) and y-axis are in log scale. Bars represent the mean absolute error per interval, with error bars showing 95% confidence intervals.

5 Results

5.1 RQ1: Discovery Probability Estimation

DP Estimation Accuracy. Figure 3 shows the discovery probability (DP) estimation result of the dependency-aware estimator ($\hat{m}_D$, green line) and GoTu ($\hat{m}_G$, orange line), along with the empirical DP ($\hat{m}_{emp}$, blue line), the ground truth of our experiment for two of the FuzzBench subjects: Freetype2 and Zlib.[4] The dependency-aware estimator consistently provides an accurate estimation of the DP compared to GoTu. In most cases, the area of the 95% confidence interval of the dependency-aware estimator substantially overlaps with the ground truth DP, i.e., what is empirically observed in the future fuzzing process. In contrast, in most cases, GoTu overestimates the DP compared to the ground truth DP. Such overestimation demonstrates the challenge of using GoTu for the DP estimation we have discussed in Section 2. Figure 4 shows the absolute error of DP estimation over time for all the subjects. The error bars represent the *AE* per time interval, with the darker dashed bars and the lighter solid bars representing AE_D and AE_G, respectively. The lines show the average empirical DP at each time interval. The figure demonstrates that the error gap between the dependency-aware estimator and GoTu is relatively consistent over time, except for Libpcap, where the difference becomes significant after six hours.

[4]The results for the other subjects are provided in the supplementary material.

Table 1: DP estimation error (*AE*) and relative error (*RE*) averaged over the fuzzing campaign. The table shows mean errors across 20 repetitions per subject, with the last two columns reporting statistical significance (p) and effect size (δ).

Subject	$\hat{m}_{emp}$	AE_G	AE_D	RE_G	RE_D	$\frac{AE_D}{AE_G}$	p	δ
Sqlite3	1.01e-04	1.02e-03	1.69e-05	8.50	0.14	0.02	2e-06	1.00
Freetype2	5.38e-05	1.45e-04	9.41e-06	3.85	0.20	0.07	2e-06	1.00
Libxml2	2.95e-04	4.13e-04	3.11e-05	0.71	0.11	0.08	2e-06	1.00
Libjpeg	2.29e-06	5.56e-06	7.45e-07	2.09	0.46	0.13	2e-06	1.00
Zlib	3.61e-07	6.80e-07	1.70e-07	2.55	1.48	0.25	2e-06	1.00
Libpcap	2.76e-08	9.38e-08	3.25e-08	15.87	4.71	0.35	2e-04	0.99
Jsoncpp	1.91e-07	1.45e-07	1.03e-07	1.26	0.94	0.71	1e-05	0.99
Libpng	1.84e-06	1.33e-06	1.27e-06	1.62	1.10	0.95	4e-04	0.92

Table 2: Records from a single fuzzing run on the Freetype2 and Libpng subjects. '#New' denotes the number of executions covering elements not in S_{n_t} among $\{X_{n_t+1}, \cdots, X_{2n_t}\}$.

Freetype2					Libpng				
t	S_{n_t}	$\lvert V_1\rvert$	$\lvert V_1^{\equiv}\rvert$	#New	t	S_{n_t}	$\lvert V_1\rvert$	$\lvert V_1^{\equiv}\rvert$	#New
1m	3655	183	71	68	1m	1501	59	44	27
10m	4273	207	55	52	10m	1603	31	26	21
1h	4545	200	57	61	1h	1644	9	8	10
6h	4860	254	51	47	6h	1659	9	9	1
24h	5234	425	59	58	24h	1667	6	6	3

Table 1 presents the error statistics of DP estimation, averaged over the entire fuzzing campaign, to assess estimator accuracy. The table reports the mean error across 20 repetitions per subject. As shown, the absolute error of GoTu often exceeds the empirical DP, resulting in a relative absolute error greater than 1.0 in seven out of eight subjects, reaching up to 16. In contrast, the absolute error of the dependency-aware estimator is significantly lower: five and seven subjects exhibit a relative absolute error below 1.0 and 2.0, respectively, resulting in a median error across subjects that is one-fourth that of GoTu. A statistical significance test on the absolute errors across 20 repetitions confirms that the dependency-aware estimator significantly outperforms GoTu (p-value < 0.001, effect size $\delta > 0.9$). For Libpcap, although the dependency-aware estimator outperforms GoTu, both exhibit high relative absolute error. This stems from blackbox fuzzing's limited effectiveness, which restricts coverage discovery. The low empirical DP makes accurate estimation challenging for both estimators.

We further analyze when and how the dependency-aware estimator outperforms GoTu by examining fuzzing records. Table 2 presents records from a single fuzzing run on the Freetype2 and Libpng subjects, reporting the number of discovered coverage elements (S_{n_t}), singletons ($|V_1|$), singleton clusters ($|V_1^{\equiv}|$), and '#New,' which counts executions covering previously undiscovered elements. Since #New serves as the numerator in empirical discovery probability computation, it provides a reference for estimator accuracy—similar to $|V_1|$ and $|V_1^{\equiv}|$, which act as numerators for their respective estimators, all sharing the denominator n. Thus, an estimator's accuracy is reflected in its closeness to #New. The Freetype2 record reveals a large gap between the number of singletons and singleton clusters, a trend seen in many subjects. This

Table 3: Stop time estimation, error (AE), and relative error (RE). m_{thres} denotes the threshold for the stopping criterion. The table shows mean values across 20 repetitions per subject.

Subject	m_{thres}	T_{emp}	T_G	T_D	AE_G^T	AE_D^T	RE_G^T	RE_D^T	$\frac{AE_D^T}{AE_G^T}$
Sqlite3	10^{-3}	6.4	355.9	7.2	349.47	0.79	54.18	0.12	0.00
Sqlite3	10^{-4}	328.5	1440.0	348.2	1111.46	19.69	3.38	0.06	0.02
Freetype2	10^{-3}	9.8	26.8	10.7	16.99	0.94	1.74	0.10	0.06
Freetype2	10^{-4}	92.6	354.6	92.2	262.07	0.34	2.83	0.00	0.00
Libxml2	10^{-3}	59.2	111.0	57.3	51.89	1.91	0.88	0.03	0.04
Libxml2	10^{-4}	444.4	673.0	467.6	228.55	23.23	0.51	0.05	0.10
Libjpeg	10^{-4}	2.1	9.3	2.9	7.16	0.83	3.38	0.39	0.12
Libjpeg	10^{-5}	42.3	160.7	51.0	118.36	8.64	2.80	0.20	0.07
Libjpeg	10^{-6}	469.4	1064.5	571.7	595.19	102.31	1.27	0.22	0.17
Zlib	10^{-5}	8.2	23.4	11.6	15.19	3.36	1.85	0.41	0.22
Zlib	10^{-6}	65.9	118.6	73.6	52.75	7.75	0.80	0.12	0.15
Zlib	10^{-7}	240.5	692.9	551.2	452.42	310.77	1.88	1.29	0.69
Libpcap	10^{-8}	1.6	780.7	567.2	779.05	565.51	474.83	344.68	0.73
Jsoncpp	10^{-5}	5.3	7.5	6.2	2.19	0.90	0.41	0.17	0.41
Jsoncpp	10^{-6}	21.1	40.5	34.6	19.40	13.53	0.92	0.64	0.70
Jsoncpp	10^{-7}	81.7	189.6	163.3	107.98	81.68	1.32	1.00	0.76
Libpng	10^{-4}	2.9	3.6	2.9	0.68	0.04	0.23	0.01	0.06
Libpng	10^{-5}	16.4	23.3	18.6	6.94	2.23	0.42	0.14	0.32
Libpng	10^{-6}	63.7	95.7	87.0	31.93	23.30	0.50	0.37	0.73
Libpng	10^{-7}	291.7	565.1	539.3	273.41	247.65	0.94	0.85	0.91

explains the significant discrepancy in discovery probability estimates between the estimators and highlights the accuracy of the dependency-aware estimator, as its estimates align closely with #New. In contrast, Libpng shows a minor difference between the two, leading to a smaller accuracy gain for the dependency-aware estimator over GoTu. This is likely due to Libpng's smaller $|S|$, which results in shorter execution traces that capture fewer coverage dependencies. Additionally, its coverage discovery saturates more quickly than in other subjects, further reducing the distinction between singletons and singleton clusters.

Stop-Time Estimation Accuracy. Table 3 presents stop-time estimation results for each subject and threshold m_{thres}, chosen based on empirical DP. The table reports mean values across 20 repetitions per $\langle$subject, $m_{thres}\rangle$ pair. As expected from the DP estimation results, the dependency-aware estimator provides more accurate stop-time estimates than GoTu. Its relative error is significantly lower across all subjects and thresholds, except for Libpcap, where both estimators perform similarly. Excluding Libpcap, all subjects have a relative error below 0.9 for the dependency-aware estimator, with a median RE_D^T of 0.17, meaning fuzzing stops less than 17% later than necessary (mean: 0.32). In contrast, for over half of the $\langle$subject, $m_{thres}\rangle$ pairs, GoTu's relative error exceeds 1.0, reaching up to 54 (median: 1.27, mean: 4.22), leading to significant resource waste. Notably, for $\langle$Sqlite3, $10^{-4}\rangle$, GoTu never reached the threshold within the 24-hour fuzzing campaign in any of the 20 repetitions. The dependency-aware estimator achieves an absolute error 7× lower than GoTu, demonstrating its superior performance in stop-time estimation.

Table 4: The impact of the node removal mechanism.

Subject	$\lvert S\rvert$	$\lvert S_{rm}\rvert$	$\frac{\lvert S_{rm}\rvert}{\lvert S\rvert}$	$T_{o.h.}$	$\mu(eps)$	$\mu(eps_{rm})$	$\mu\left(\frac{eps_{rm}}{eps}\right)$
Sqlite3	58,253	32,849	0.56	-82.6s (-29%)	21.93	31.43	1.43
Freetype2	46,051	26,553	0.58	-7.6s (-9%)	51.06	104.05	2.04
Libxml2	93,858	50,573	0.54	-5.7s (-11%)	18.82	50.17	2.67
Libjpeg	36,840	21,788	0.59	-9.9s (-18%)	290.40	1061.58	3.66
Zlib	1,775	986	0.56	-0.6s (-13%)	7043.66	7683.91	1.09
Libpcap	14,355	7,815	0.54	-8.9s (-15%)	5415.95	5664.24	1.05
Jsoncpp	8,780	5,631	0.64	-3.5s (-9%)	2686.26	3370.08	1.25
Libpng	10,463	6,077	0.58	-2.7s (-10%)	2210.39	3140.78	1.42
Avg.			0.57	-15.2s (-14%)			1.83

Answer to RQ1: Our dependency-aware estimator significantly outperforms the Good-Turing estimator in discovery probability estimation. Five and seven subjects show a relative absolute error below 1.0 and 2.0, respectively, with a median error one-fifth that of the Good-Turing estimator. It also provides accurate stop time estimates, with an absolute error 7× lower than that of the Good-Turing estimator.

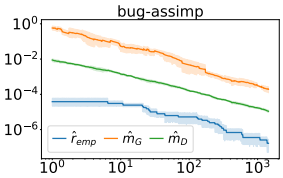
5.2 RQ2: Effect of Node Removal Mechanism

Table 4 presents the results of the node removal mechanism, reporting the number of nodes in the original ($|S|$) and reduced control-flow graphs ($|S_{rm}|$) along with the proportion of nodes remaining after removal. Results show a significant reduction in control-flow graph size, decreasing node count by 36-46% (average: 43%).

Notably, node removal also reduces compilation time overhead. $T_{o.h.}$ represents this overhead, computed as the difference in compilation time with node removal minus that without it, measured in seconds (s) and percentage (%). The results show an average reduction of 15.2s (14%), a consistent trend across subjects with negligible correlation to subject size. This negative overhead suggests that the time saved from reducing the number of basic blocks to instrument outweighs the cost of node removal.

$\mu(eps)$ and $\mu(eps_{rm})$ denote the average executions per second (eps) during fuzzing with and without node removal. Their ratios range from 1.05 to 3.66, averaging 1.83, indicating improved fuzzing efficiency. Compared to Table 1, the dependency-aware estimator generally shows greater DP estimation accuracy improvement when eps is lower. This aligns with the intuition that when many discoveries remain, the difference between singleton and singleton cluster sizes is larger. Additionally, longer execution traces (reflected in lower eps) suggest more dependencies, reinforcing the mechanism's impact.

Answer to RQ2: The node removal mechanism significantly reduces control-flow graph size, decreasing nodes by 36-46% (average: 43%). The mechanism also reduces compilation time overhead by 15.2s (14%) on average, indicating that the time saved from reducing the number of basic blocks to instrument outweighs the cost of node removal.



Subject	$\|C\|$	$\hat{r}_{emp}^{t=0}$	RE_{GoTu}^{r}	RE_{St}^{r}
Assimp	5.6	3.8e-05	1090.6	53.4
File	4.4	3.1e-05	124.3	38.5
Harfbuzz	4.2	1.9e-05	3557.3	347.7
Libxml2	6.0	6.7e-06	1190.2	107.6

Figure 5: *Left*: Residual risk estimation for Assimp (x-axis: time (min.), y-axis: prob.). *Right*: Result summary for the bug-based evaluation. $|C|$ is the number of unique crashes found during the fuzzing, and $\hat{r}_{emp}^{t=0}$ is the initial residual risk. The table reports the mean values across 5 repetitions.

5.3 RQ3: Bug-based Residual Risk Estimation

In this experiment, we evaluate residual risk estimation using DP estimators. Since not all coverage elements are bugs, DP estimation is expected to overapproximate residual risk. The left side of Figure 5 shows the empirical residual risk $\hat{r}_{emp}$ and DP estimation for Assimp (4d451fe). As expected, DP estimation overapproximates residual risk, but the dependency-aware estimator provides a closer estimate than GoTu, aligning with our study's goal.

The right side of Figure 5 summarizes results for bug-based residual risk estimation. The table reports the number of unique crashes found during fuzzing ($|C|$), the initial residual risk $\hat{r}_{emp}^{t=0}$, and the relative error of residual risk estimation for both estimators. The dependency-aware estimator achieves an error one to two orders of magnitude lower than GoTu. This result indicates that the dependency-aware estimator mitigates two layers of overestimation in residual risk analysis, providing a more accurate estimate.

Answer to RQ3: The dependency-aware estimator mitigates two layers of overestimation in residual risk analysis by directly estimating the discovery probability, resulting in an error one to two orders of magnitude lower than the Good-Turing estimator.

5.4 RQ4: Estimation over Greybox Fuzzing

Presentation. In this section, we evaluate the effectiveness of our dependency-aware estimator in the context of *greybox fuzzing*, which is subject to adaptive bias. In greybox fuzzing, generated inputs that increase coverage are added to the seed corpus, altering the discovery probability each time a new seed is introduced. Existing estimators for greybox fuzzing that are not dependency-aware are the reset estimator (Reset; Eq. (8)) and the mean local estimator (MLG; Eq. (9)). We adapt our dependency-aware estimation methodology to greybox fuzzing by substituting the number of singletons in the mean local estimator with the number of singleton clusters (see Section 4.2), and refer to the resulting variant as the dependency-aware mean local estimator (MLD).

Since discovery probability approaches 0 only in the limit, a negatively biased estimator may appear to have a low absolute error while still differing from the true value by several orders of magnitude. To capture this discrepancy, we report the logarithmic error (LE) in addition to the absolute error (AE) and relative error (RE). The logarithmic error is defined as $LE_{esti} = \log_{10}(\hat{m}_{esti}) - \log_{10}(\hat{m}_{emp})$, where $\hat{m}_{esti}$ is the estimate produced by the estimator.

Results. Table 5 presents the performance of two existing estimators for greybox campaigns—Reset and MLG—alongside our dependency-aware extension of MLG, termed MLD, evaluated over 24-hour campaigns. Additional results are provided in the supplementary material. We observe that MLD outperforms both existing estimators in the greybox setting, confirming the finding of **RQ1** that dependency-aware estimation improves DP estimator performance.

Between the two Mean Local estimators, MLD consistently achieves a lower error than the original MLG across all subjects. For four subjects—Sqlite3, Freetype2, Libxml2, and Libjpeg —where our dependency-aware estimator demonstrated dominant performance over GoTu in blackbox fuzzing in terms of absolute error ($\frac{AE_D}{AE_G} < 0.2$), MLD also outperforms MLG in greybox fuzzing, achieving a relative error below 0.45 in all cases. The last two columns of Table 5, which report the statistical significance (p) and effect size (δ) of the difference between the absolute errors of MLD (AE_{MLD}) and MLG (AE_{MLG}), statistically confirm this outperformance: the p-value is less than 10^{-6} and the effect size δ is approximately 1.0.

While still outperforming GoTu, the dependency-aware estimator shows less improvement in certain subjects. This includes not only cases where the difference between singleton and singleton-cluster sizes is small (Libpcap) but also subjects where the mean local estimator itself is particularly inaccurate (Zlib, Jsoncpp, and Libpng). One common cause is the frequent addition of new seeds, which have not yet been chosen for mutation, to the corpus; their local DP estimate remains 0.5 (Eq. (9)), which can significantly contribute to DP overestimation. This suggests that the poor accuracy of the mean local estimator itself may be the primary factor limiting the performance of the dependency-aware approach in these cases.

The Reset estimator underestimates the empirical DP by at least two orders of magnitude across all subjects, with an average logarithmic error of -3.42 (while MLG (1.85) and MLD (1.52) exhibit a bias of between one and two orders of magnitude). Underestimating residual risk by so many orders of magnitude is problematic, as it may lead a security researcher to believe that the likelihood of discovering a vulnerability is substantially lower than it actually is.

Answer to RQ4: The dependency-aware estimator improves upon state-of-the-art discovery probability estimation in greybox fuzzing. Its effectiveness is greater in subjects where the mean local estimation approach accurately accounts for adaptive bias in the sampling distribution.

6 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 generalize to other subjects and tools. While we do not claim our findings apply to all software, we aim to minimize this threat by selecting subjects from FuzzBench [23], a widely recognized fuzzer benchmark. We based our selection on well-defined criteria from prior work [4], including all software used in previous studies while expanding the selection to cover a broader range of sizes and characteristics. Another concern is *internal validity* or the degree to which our study controls systematic error. To minimize random variation and enhance statistical power,

Table 5: Summary of results for the greybox fuzzing evaluation. Mean values over 20 repetitions are reported. *LE* denotes the estimator's error in logarithmic scale, defined as $LE_{esti} = \log_{10}(\hat{m}_{esti}) - \log_{10}(\hat{m}_{emp})$, where $\hat{m}_{esti}$ is the estimate of the DP. *AE* and *RE* denote absolute and relative errors, respectively. The last two columns report the statistical significance (p) and effect size (δ) of the difference between the absolute errors of the original Mean Local estimator (AE_{MLG}) and the dependency-aware estimator (AE_{MLD}).

Subject	$\hat{m}_{emp}$	LE_{Reset}	LE_{MLG}	LE_{MLD}	AE_{Reset}	AE_{MLG}	AE_{MLD}	RE_{Reset}	RE_{MLG}	RE_{MLD}	$\frac{AE_{MLD}}{AE_{MLG}}$	p	δ
Sqlite3	9.57e-02	-2.26	1.44	0.36	9.49e-02	4.60e+00	1.34e-01	0.92	45.95	1.35	0.03	2e-6	1.00
Freetype2	3.07e-02	-2.49	1.48	0.98	3.04e-02	1.43e+00	2.87e-01	0.92	49.50	10.54	0.20	2e-6	1.00
Libxml2	1.85e-02	-2.01	1.88	1.18	1.80e-02	2.60e+00	3.17e-01	0.91	120.25	16.63	0.12	2e-6	1.00
Libjpeg	1.24e-02	-3.87	1.58	1.34	1.23e-02	3.53e-01	1.60e-01	0.93	350.16	245.86	0.45	2e-6	1.00
Zlib	3.58e-03	-3.93	2.78	2.78	3.58e-03	2.17e-01	2.17e-01	0.93	7364.87	7354.14	1.00	2e-6	1.00
Libpcap	1.95e-02	-5.29	1.10	1.06	1.95e-02	2.22e-01	1.92e-01	0.93	17.20	15.42	0.87	2e-6	1.00
Jsoncpp	1.12e-03	-3.09	2.87	2.83	1.12e-03	2.82e-01	2.18e-01	0.93	3570.90	3258.92	0.77	2e-6	1.00
Libpng	6.59e-03	-4.46	1.65	1.62	6.58e-03	2.58e-01	2.27e-01	0.93	454.99	390.82	0.88	2e-6	1.00

we ran each experiment 20 times for **RQ1** and 5 times for other **RQs**, reporting statistical outcomes where applicable. Finally, there is a risk of errors in our evaluation. We have made all scripts and data publicly available to ensure transparency and reproducibility.

7 Related Work

Residual Risk Analysis and Reliability of Software Systems. Recently, methods for measuring residual risk in software testing have been actively explored using various approaches. Most employ biostatistical methods [14], such as Good-Turing and Laplace [2, 4, 18, 24, 26, 29], or machine-learning techniques [27] to estimate the probability of discovering new bugs. However, while statistical estimators assess residual risk without analyzing program semantics, they overlook dependencies between coverage elements, often leading to inaccurate results. In contrast, our dependency-aware discovery probability estimation leverages program dependencies, yielding 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 [8, 9, 11]. However, model counting is computationally intensive and may not scale well to large software systems. In contrast, our dependency-aware discovery probability estimation is lightweight and scalable for large-scale software.

Our primary focus is on the residual risk of undetected bugs during an ongoing greybox testing campaign. These methods may help allocate testing resources efficiently and refine testing strategies [25]. Meanwhile, extensive research has examined methods for quantifying overall software reliability [21]. However, as Filieri et al. [8] 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. [19] estimated *reachable coverage*—the number of coverage elements a fuzzer can potentially reach—using statistical methods. Another approach is *extrapolating the coverage rate* [2, 20], which estimates potential additional coverage within a future time frame. Statistical analysis also extends beyond general progress predictions; for example, Lee and Böhme [13] estimated the probability of reaching specific program states that remain unreached. Such predictions also aid in information leakage analysis [15], where statistical estimations quantify information leakage in software systems.

8 Discussion

In this work, we proposed dependency-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, significantly overestimates residual risk. Our dependency-aware discovery probability estimation accounts for program dependency, providing more accurate estimates of the discovery probability. Theoretically, our estimator is grounded in the incidence data model and guarantees tighter—or at least equally tight—bounds than the Good-Turing estimator; it achieves equality only in the hypothetical case where coverage elements are entirely independent—a condition that virtually never holds in real software. Empirical evaluations using FuzzBench subjects show that our estimator reliably yields accurate discovery probability estimates. Two orthogonal optimizations further demonstrate the practicality of our estimator for real-world software testing with large coverage sets and extensive execution traces. Our online singleton cluster maintenance mechanism enables efficient computation of the estimator, with the same space complexity as the Good-Turing estimator. In the absence of dependencies, our approach would incur a slightly higher memory footprint. However, in practice, the node-removal mechanism eliminates approximately 43% of the nodes, which are not recorded, thereby reducing memory overhead (see Table 4 for details).

Our dependency-aware discovery probability estimation extends beyond residual risk analysis; it provides a framework for estimating the probability of observing new behaviors across a range of empirical program analyses. Its key advantage is 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 [13] 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, as they frequently generate new program variants and observe their behaviors. Depending on the behavior space's semantics (e.g., the RIPR model [1, 17] for mutation testing or precondition violation levels in program repair), these can be multidimensional and modeled as incidence data. We anticipate dependency-aware discovery probability estimation extending to other sampling-based methodologies that handle incidence data or class dependencies.

9 Data Availability

All codes and data used in the paper are available at

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

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and for helping us improve this paper. This research is funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is supported by ERC grant (Project AT_SCALE, 101179366).

References

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

[2] 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

[3] Marcel Böhme. 2022. Statistical Reasoning about Programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '22)*. Association for Computing Machinery, New York, NY, USA, 76–80. https://doi.org/10.1145/3510455.3512796

[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] Edward E. Cureton. 1956. Rank-Biserial Correlation. *Psychometrika* 21, 3 (Sept. 1956), 287–290. https://doi.org/10.1007/BF02289138

[7] Edsger W. Dijkstra. 1972. Chapter I: Notes on Structured Programming. In *Structured Programming*. Academic Press Ltd., GBR, 1–82.

[8] 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

[9] 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

[10] 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)*.

[11] 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

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

[13] Seongmin Lee and Marcel Böhme. 2023. Statistical Reachability Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23) (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 326–337. https://doi.org/10.1145/3611643.3616268

[14] Seongmin Lee and Marcel Böhme. 2025. How Much Is Unseen Depends Chiefly on Information About the Seen. In *Proceedings of the 13th International Conference on Learning Representations (ICLR'25)*.

[15] Seongmin Lee, Shreyas Minocha, and Marcel Böhme. 2025. Accounting for Missing Events in Statistical Information Leakage Analysis. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE'25)*. Association for Computing Machinery.

[16] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141. https://doi.org/10.1145/357062.357071

[17] 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

[18] 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

[19] 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

[20] Danushka Liyanage, Seongmin Lee, Chakkrit Tantithamthavorn, and Marcel Böhme. 2024. Extrapolating Coverage Rate in Greybox Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24) (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3597503.3639198

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

[22] 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

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

[24] 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.

[25] Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2025. Novelty Not Found: Exploring Input Shadowing in Fuzzing through Adaptive Fuzzer Restarts. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). https://doi.org/10.1145/3712186

[26] 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

[27] Neil Walkinshaw, Michael Foster, José Miguel Rojas, and Robert M. Hierons. 2024. Bounding Random Test Set Size with Computational Learning Theory. *Proc. ACM Softw. Eng.* 1, FSE (July 2024), 112:2538–112:2560. https://doi.org/10.1145/3660819

[28] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. https://doi.org/10.2307/3001968 jstor:3001968

[29] 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

[30] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. https://doi.org/10.1145/3460120.3484596