Reachable Coverage: Estimating Saturation in Fuzzing

Danushka Liyanage[†] Marcel Böhme[‡†] Chakkrit Tantithamthavorn[†] Stephan Lipp[*]
[†]Monash University, Australia [‡]MPI-SP, Germany [*]TU Munich, Germany

***Abstract*—Reachable coverage is the number of code elements in the search space of a fuzzer (i.e., an automatic software testing tool). A fuzzer cannot find bugs in code that is unreachable. Hence, reachable coverage quantifies *fuzzer effectiveness*. Using static program analysis, we can compute an upper bound on the number of reachable coverage elements, e.g., by extracting the call graph. However, we cannot decide whether a coverage element is reachable in general. If we could precisely determine reachable coverage efficiently, we would have solved the software verification problem. Unfortunately, we cannot approach a given degree of accuracy for the static approximation, either.**

In this paper, we advocate a *statistical* perspective on the approximation of the number of elements in the fuzzer's search space, where accuracy *does* improve as a function of the analysis runtime. In applied statistics, corresponding estimators have been developed and well established for more than a quarter century. These estimators hold an exciting promise to finally tackle the long-standing challenge of counting reachability. In this paper, we explore the utility of these estimators in the context of fuzzing. Estimates of reachable coverage can be used to measure (a) the amount of untested code, (b) the effectiveness of the testing technique, and (c) the completeness of the ongoing fuzzing campaign (w.r.t. the asymptotic max. achievable coverage). We make all data and our analysis publicly available.

I. Introduction

Fuzzing has become one of the most successful automatic bug finding techniques in practice. For instance, in the five years since its launch, OSS-Fuzz alone has found about 41,000 bugs in over 650 open source projects—only by fuzzing [35]. Ever since, many new fuzzers have been released, some more effective than others.

What does it mean for a fuzzer to be effective? Given a program, we consider a fuzzer which has the *capability* of exposing more bugs as more effective. Since bugs are scarce, we measure the amount of "reachable code". A fuzzer cannot find bugs in unreachable code. Such bugs are outside of the fuzzer's search space. With "fuzzer" we mean the *automatic testing tool* that generates the test inputs, the *test harness* that passes the test inputs to the program, and the *initial corpus* of seed inputs (if one is used). The effectiveness of a fuzzer can be increased by improving the test harnesses, by changing the seed corpus, or by using a different testing tool.

Evaluating fuzzer effectiveness is important. Together with the fuzzing research community, Google has been developing a fuzzer benchmarking platform, called Fuzzbench [45], to identify the most effective fuzzer for each program and across all programs in one of the largest benchmarks. The most effective fuzzers are given higher priority when run within the ClusterFuzz project. To measure the fuzzer effectiveness, Fuzzbench follows the standard recommendations from the fuzzing community [42], [5], [13] and measures the coverage achieved (and #bugs found) *within a given time budget*.

However, the *number of covered elements* $S(n)$ in a fixed-length fuzzing campaign is *not* a reliable indicator of fuzzing effectiveness. Firstly, the chosen time budget is an arbitrary convention. For instance, in the fuzzing literature, a typical recommendation is to run the fuzzer for one day [42], [45]. In the search-based testing literature, time budgets vary between 3 and 15 minutes [4], [32]. Secondly, for all practical purposes $S(n)$ does not even seem to asymptotically approach S as the number of generated test inputs n increases. We checked the coverage achieved by 11 fuzzers run 20 times on 32 programs *for 7 days* and have been unable to identify the asymptotic, maximum achievable coverage in log-scale time for all but one program (cf. Section II-A). In fact, Böhme et al. [13] observed that there is only a moderate agreement between benchmarking results given a 1-hour versus a 23-hour time budget. The best fuzzer after one hour might be the worst fuzzer after one day. So then, what is a reasonable time budget? Fortunately, the total number of reachable coverage elements S—as our measure of fuzzing effectiveness—is *not* parametric in n.

Given a fuzzer, *reachable coverage* S measures the amount of code that the fuzzer has the capability of reaching, i.e., the maximum achievable coverage. Conversely, reachable coverage also measures the amount of currently *untestable code*. Some code might only be testable when the test harness is improved or better seeds are provided. Fuzzer effectiveness is maximized when the amount of untestable code is minimized. In an ongoing campaign of length n, the currently achieved coverage $S(n)$ provides a trivial lower bound on the reachable coverage. The proportion $S(n)/S$ of reachable coverage currently achieved provides a measure of progress of the ongoing fuzzing campaign towards completion (i.e., when to stop).

Using static program analysis, we can upper-bound the number of reachable coverage elements. For instance, OSS-Fuzz uses a static analyzer called Fuzz-Introspector [16] which analyzes the call graph to approximate reachable coverage and assess the quality of the fuzzer integration into any of the 650 enrolled projects. However, static analysis suffers from over- and undercounting (Section II-B) which cannot be remedied by a longer running time. In fact, we cannot decide whether there exists an input that executes a given coverage element in general. If we could precisely determine reachable coverage efficiently, we would be proving the *absence* of errors not their presence. To see this, let us encode the verification problem as a reachability problem. We let each `assert(`φ`)`-statement be transformed into `if(`$\neg\varphi$`) assert(false)`. We would prove that the program is correct w.r.t. these assertions if none of these `assert(false)`-statements is reachable. So how can we efficiently approximate the effectiveness of a fuzzer to a given degree of accuracy?

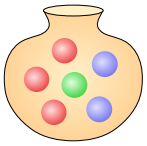


Fig. 1: We can model the estimation of reachable coverage as an estimation of the number of colors in an urn full of colored balls (Illustration by Quartl; CC-BY-SA 3.0).

In this paper, we propose a *statistical approach* for the approximation of fuzzer effectiveness which may allow us to tackle the undecidability challenge of the reachability problem *empirically* by increasing the analysis runtime until achieving a required degree of accuracy. This approach is inspired by Böhme [8] who proposed applying techniques from biostatistics such as estimators and extrapolators to assess fuzzing. Prior work has successfully used and extended this link between fuzzing and biostatistics to assess other aspects in fuzzing such as efficiency [7],[1] scalability [9], behavioral diversity [49] and residual risk [10]. Here, we use STADS to study the fuzzer's *effectiveness*.

As illustrated in Figure 1, given an urn with colored balls from which n balls are sampled with replacement and $S(n)$ colors are observed, *how many colors S are in this urn?* Variants of this question appear in many disciplines, such that applied statisticians have developed a number of estimators over the past thirty years [22], [24], [48]. In the following, we explore—in the context of fuzzing—the utility of the most widely used estimators to quantify the asymptotic total number of coverage elements in a fuzzer's search space.

We empirically evaluate the accuracy of the approximations of reachable coverage produced by seven state-of-the-art estimators [19], [21], [27], [43], [37], [15] and two static analysis tools [51], [16] in a context where the ground-truth reachable coverage is known and in a context where it is not known. For *static analysis tools*, we find that they over- and under-count reachable coverage. The static approximation is inaccurate, but it is difficult to know exactly how inaccurate the approximation is or how to improve its accuracy. For the *statistical method*, we find that the Jackknife estimators [15] exhibit the smallest magnitude in bias and variance compared to the other estimators. All estimators are asymptotically consistent, meaning that their accuracy improves with campaign length. However, in the context where the reachable coverage is empirically unknown, we find that all estimators predict *false peaks*: More coverage can be achieved upon reaching the predicted reachable coverage.

In summary, our paper makes the following contributions:

- An empirical investigation of the size of a fuzzer's search space. Particularly, we study the asymptotic behavior of the coverage achieved over time $S(n)$ as n grows large (7 days). For 31 out of 32 programs, we are *unable* to identify the asymptote S in log-scale time which motivates our work on better estimates of reachable coverage S.
- An in-depth discussion of over- and under-counting in *static analysis* designed to provide an upper bound on reachable coverage S (state-of-the-art used in OSS-Fuzz [16], [3]).
- A survey and empirical evaluation of *statistical estimators* of reachable coverage S which have been developed in applied statistics and widely used in other disciplines. We find that existing estimators under-estimate fuzzer effectiveness and motivate the development of novel estimators specifically for the domain of fuzzing.

[1]Efficiency is the rate at which new elements are reached.

II. Challenges of Existing Approximations of S

Reachable coverage quantifies the number of elements in the fuzzer's search space and thus a fuzzer's effectiveness. In the following, we shed some light on the challenges of existing approximations of reachable coverage. Firstly, we attempt to determine the total number of coverage elements in the fuzzer's search space by observing the asymptotic behavior of the coverage achieved $S(n)$ as n grows very large. Secondly, we discuss the challenges of over- and undercounting in static call graph reachability analysis, as it is used at OSS-Fuzz [16] to analyze fuzzer effectiveness on more than 300 OSS projects.

A. Coverage Achieved: No Indication of an Asymptote

Setup. *How often can we observe the asymptotic, maximum achievable coverage in a week-long fuzzing campaign*? Coverage achieved within a given time budget is the standard evaluation of automatic testing effectiveness [5], [42], [13], [45]. But is it really indicative of the maximum achievable (i.e., reachable) coverage? We plot coverage achieved over time for *all* week-long fuzzing campaigns in Fuzzbench [45] that were *not* terminated prematurely.[2] Together they involve 32 widely-used open-source programs, 11 fuzzers (incl. AFL++), and more than one CPU-*century* worth of fuzzing campaigns. The experiments were run in default configuration using fuzz harnesses, initial seeds, and dictionaries provided by the open-source community. Fuzzbench records coverage information in fifteen-minute intervals using the coverage-increasing inputs that are being added to the current seed corpus.

Results. Figure 2 shows the coverage achieved on a log-scale for time. Apart from one program (`jsoncpp`) there is no evidence that the fuzzer reaches an asymptote for any of these 32 programs. For 22 programs, the coverage achieved over time comes out as a straight line in log-time while for the remaining seven (7) programs, the straight line is clearly identifiable after the first hour. We should point out that every tick on the y-axis increases the length of the fuzzing campaign by a factor of 10x. The next tick at the 1000-hour-mark would require 30 repetitions of 1.5-**month**-long fuzzing campaigns.

The linear shape in log-scale time might be explained by Böhme's empirical law which hypothesizes an exponential cost for fuzzing [9]. However, the absence of a discernible asymptote in log-scale time comes as a surprise—given that there always seems to be an asymptote in linear-scale time.[3]

[2]The Experiment IDs are `2021-04-11-7d-paper`, `2021-04-23-7d-paper`, `2021-07-10-redo-7d`, and `2021-08-19-crash-s-7d`.

[3]We confirmed the absence of an asymptote for Fuzztastic (Figure 8), too, where we measure basic block coverage over time in week-long campaigns.

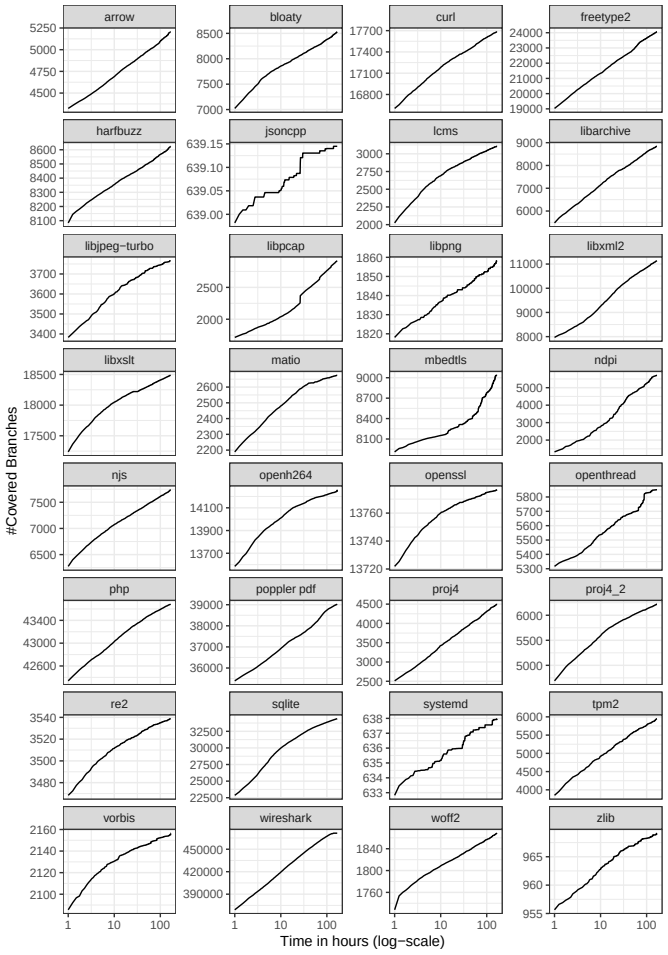


Fig. 2: No discernible asymptote when plotting the average number of branches covered over time in 30x 7-day campaigns on the log-time scale (**116.9 CPU years**).

In practice, we *cannot* precisely determine the reachable coverage S of a fuzzer by observing the achieved coverage $S(n)$ in a very long fuzzing campaign (as n gets larger).

This result motivates the invention and study of better estimators of reachable coverage to measure fuzzing effectiveness.

B. Static Call Graph Reachability: Inaccurate Approximation

Setup. *How well does static analysis approximate reachable coverage?* In OSS-Fuzz [3], the quality of the fuzzer integration into each of the 650+ OSS projects is evaluated using reachable coverage. The compliment, unreachable code identifies *untestable code*, beyond the *currently* untest*ed*, uncovered code. Specifically, OSS-Fuzz uses the Fuzz-Introspector (FI) static analysis tool to compute the number of reachable and the number of covered functions [16]. FI implements an LLVM link time optimization (LTO) pass to extract the program's entire call graph when the C project is built. During link time, all code is available and can be analyzed. FI uses several state-of-the-art heuristics to improve the completeness of the extracted call graph. FI finds reachable functions (i) by finding destinations of normal function calls, (ii) by finding destinations of register-indirect calls in the vtable (dynamic dispatch via virtual method table), and (iii) by finding assignments of function pointers to variables or function call parameters.[4]

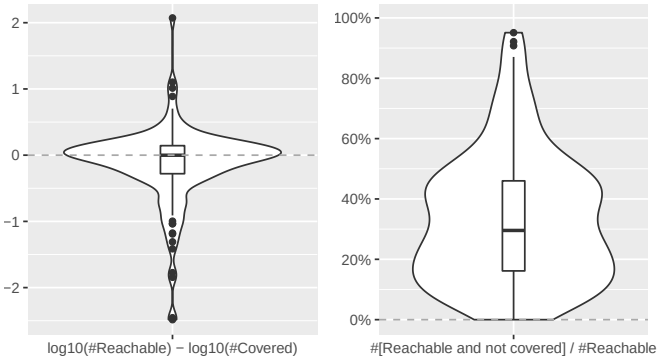


Fig. 3: Under-counting and over-counting in Fuzz-Introspector static analyzer of reachable coverage in OSS-Fuzz [16], [3].

Figure 3 illustrates under- and over-counting of the FI static analysis. We extracted the number of functions reported as reachable (#reachable) and the number of functions that were covered (#covered) from the FI reports generated of all 354 open source programs that were analyzed on 15 August 2022.[5] To mitigate threats to validity, we removed 202 programs with multiple fuzz harnesses (which were not properly handled in FI, yet) and 14 programs where no function was reported as covered (due to a bug in the coverage instrumentation), leaving us with the current FI reports for 138 programs.

Under-counting. Figure 3.left shows the difference in the logarithm of the number of functions considered reachable versus actually covered. We use a log-transformation because there are many programs with a few functions but a few programs with many functions (showing a long-tail distribution).

For the majority of projects (i.e., on the median), there are more functions covered than considered reachable.

Over-counting. Figure 3.right shows the proportion of reachable functions *not* covered. For the majority of projects about 30+% of the reachable functions are not covered, suggesting a large potential to improve the function coverage. However, if we compare the number of functions covered today with those covered five months prior, coverage does *not* increase for 72% of projects, despite continuous fuzzing. The corpus is saturated. For the remaining 28% of programs, the median coverage increase is just over 2%. Given these numbers, the 30+% of uncovered but reachable functions that FI predicts seems unrealistic. Unfortunately, there is no way to "non-technically" improve the accuracy of this approximation. Generally, the question of whether there exists an input that

[4]https://github.com/ossf/fuzz-introspector/blob/main/frontends/llvm/lib/Transforms/FuzzIntrospector/FuzzIntrospector.cpp#L812-L850

[5]List of analyzed projects: https://oss-fuzz-introspector.storage.googleapis.com. Project-specific FI reports: https://storage.googleapis.com/oss-fuzz-introspector/[project]/inspector-report/20220816/fuzz_report.html

reaches a function is undecidable. As mentioned earlier, if a practical decision procedure existed, we could consider the software verification problem as solved.

For the majority of projects, the predicted number of reachable but uncovered functions seems unrealistic.

III. Estimation of Reachable Coverage as Measure of Fuzzer Effectiveness

We wish to measure reachable coverage as the number of coverage elements in the fuzzer's search space (i.e., #elements that the fuzzer has the capability of covering). This measure of fuzzer effectiveness does *not* depend on the generated number n of test inputs. Instead of computing reachable coverage *statically before* running a fuzzing campaign, we propose to estimate it *dynamically during* the campaign. In contrast to static analysis, the estimation allows us to approximate reachable coverage to a given degree of accuracy. The hope is to overcome the undecidability challenge of the reachability problem that afflicts any static reachability analysis.

Using statistical estimation to approximate the effectiveness of a fuzzer seems particularly promising given the previous results of estimating (i) the efficiency of a fuzzer [11], (ii) the residual risk of an ongoing campaign [10], and (iii) the diversity of the generated inputs [49].

A. Statistical Model

The simple urn model illustrated in Figure 1 can be extended by allowing each ball to have multiple colors. In the following, we recall the standard definitions from the *Bernoulli Product model* of the STADS statistical framework [8]. The Bernoulli Product model associates each input (i.e., ball) with the exercised coverage elements (i.e., colors or "species").

Terminology. Let $\mathcal{P}$ be the program we wish to fuzz and $\mathcal{D}$ the set of all inputs that $\mathcal{P}$ can exercise. A fuzzing campaign is a stochastic process

$$\mathcal{F} = \{X_n \mid X_n \in \mathcal{D}\}_{n=1}^{N}$$

where N input are sampled with replacement from $\mathcal{D}$. Suppose, we can divide the search space $\mathcal{D}$ into S individual subdomains $\{\mathcal{D}_i\}_{i=1}^{S}$ called *coverage elements* (or species). An input $X_n \in \mathcal{F}$ is said to *newly cover* element $\mathcal{D}_i$ if $X_n \in \mathcal{D}_i$ and there does not exist a previously sampled input $X_m \in \mathcal{F}$ such that $m < n$ and $X_m \in \mathcal{D}_i$ (i.e., $\mathcal{D}_i$ is covered for the first time). In this paper, we are interested in estimating the total number of coverage elements S that the fuzzer can cover. We let $\{p_i\}_{i=1}^{S}$ be the probability that a fuzzer-generated input $X_n \in \mathcal{D}_i$ where $i : 1 \leq i \leq S$. We call $\mathcal{F}$ as the fuzzing campaign of a non-deterministic blackbox fuzzer.

Bernoulli Product model. If we define branches as our coverage elements of interest, then the coverage of the new element corresponds to an increase in branch coverage. In the Bernoulli product model [8], [24], an input can cover one or more elements. Specifically, for a fuzzing campaign of length N, we let the incidence matrix $W_{S\times N}$ be defined as

$$W_{S\times N} = \{W_{ij} \mid i = 1, 2, ..., S \wedge j = 1, 2, \ldots, N\}$$

where $W_{ij} = 1$ if input X_j covers $\mathcal{D}_i$ and $W_{ij} = 0$ otherwise. We further define the *incidence frequency counts* f_k, where $0 \leq k \leq N$, as the number of elements covered by exactly k generated inputs. It is assumed that each element W_{ij} is a Bernoulli random variable with probability p_i [24], such that the probability distribution for the incidence matrix can be expressed as the probability for all $i : 1 \leq i \leq S$ and $j : 1 \leq i \leq N$ that we have $W_{ij} = w_{ij}$.

$$P\left(\forall(i,j).W_{ij} = w_{ij}\right) = \prod_{j=1}^{N}\prod_{i=1}^{S} p_i^{w_{ij}}(1-p_i)^{1-w_{ij}} \quad (1)$$

The marginal for Y_i follows a binomial distribution characterized by campaign length N and coverage probability p_i,

$$P\left(\forall i.Y_i = y_i\right) = \prod_{i=1}^{S} p_i^{y_i}(1-p_i)^{N-y_i} \quad (2)$$

Hence, the incidence frequency counts can be derived as

$$f_k(n) = E\left[\sum_{i=1}^{S} I(Y_i = k)\right] = \sum_{i=1}^{S}\binom{n}{k}p_i^k(1-p_i)^{n-k} \quad (3)$$

Specifically, $S(n) = S - f_0(n)$ where $f_0(n) = \sum_{i=1}^{S}(1-p_i)^n$.

B. Estimators

The class of estimators that have been developed to estimate the total number of colors in an urn full of colored balls (cf. Figure 1) and those for the Bernoulli Product model is called *species richness estimators*. Figure 4 shows seven (7) *state-of-the-art* [39], [14], [50] species richness estimators defined for the Bernoulli Product model [8]. All listed species richness estimators are *non-parametric*. This means, they can be used for any type of coverage criterion (statement, branch, MCDC, mutation coverage, etc.), for any fuzzer, and for any program irrespective of the underlying distribution $\{p_i\}_{i=1}^{S}$. We do not consider parametric estimators that assume specific distributions [29]. All estimators shown in Figure 4 are also made available via the `Spade` R package which implements the state-of-the-art in species richness estimation [17].

Chao-type estimators. In 1984, Anne Chao suggested that it is unreasonable to provide a point estimate of the total number of species $S = S(n) + f_0(n)$. In applied statistics, the sample coverage $C(n)$ is a property of a sample that quantifies the probability that the next sampled individual $(n + 1)$ belongs to a species that is already present in the sample. Chao's position is that the complement of the sample coverage (i.e., the discovery probability $1 - C(n)$ [10]) is always non-zero, and that we can always add arbitrarily many species f_0 into this "missing" probability mass. Instead, Chao proposed an estimator of a *lower bound* on S [18], which for the Bernoulli Product model is called the *Chao2* estimator. It can be theoretically shown that *Chao2* performs best for the uniform distribution where species are equally likely. In addition to *Chao2*, we also evaluate a *bias-corrected* variant which tackles systematic bias (*Chao2-bc* [21]) plus an *improved* variant which also uses f_3 and f_4 (*iChao2* [27]).

Estimator	Description and Formula
Chao2 [19]	*Chao estimator (1984)* $\hat{S}_{Chao2} = \begin{cases} S(n) + \frac{(n-1)}{n}\frac{f_1(n)^2}{2f_2(n)} & \text{if } f_2(n) > 0 \\ S(n) + \frac{(n-1)}{n} f_1(n)\frac{f_1(n)-1}{2} & \text{if } f_2(n) = 0 \end{cases}$
Chao2-bc [21]	*Bias corrected Chao estimator (2005)* $\hat{S}_{Chao2(bc)} = S(n) + f_1(n)\left[\frac{f_1(n)-1}{2(f_2(n)+1)}\right]$
iChao2 [27]	*Improved Chao estimator (2014)* $\hat{S}_{iChao2} = \hat{S}_{Chao2} + \frac{(n-3)}{n}\frac{f_3(n)}{4f_4(n)} \cdot \max\left(f_1(n) - \frac{n-3}{n-1}\frac{f_2(n)f_3(n)}{2f_4(n)}, 0\right)$
ICE [43]	*Incidence-based coverage estimator (1994)* $\hat{S}_{ICE} = S(n)_{freq} + \frac{S(n)_{infr}}{\hat{C}(n)_{ICE}}\hat{\gamma}^2$ where $\hat{\gamma}^2 = \max\left(\frac{S(n)_{infr}}{\hat{C}(n)_{ICE}}\frac{n_{infr}}{-1+n_{infr}}\frac{\sum_{i=1}^{k} i(i-1)f_i(n)}{(n_{infr})^2} - 1, 0\right)$ $S(n)_{infr} = \sum_{i=1}^{k} f_i(n)$, $S(n)_{freq} = S(n) - S(n)_{infr}$, $n_{infr} = \sum_{i=1}^{k} i f_i(n)$, $\hat{C}(n)_{ICE} = 1 - f_1(n)/n_{infr}$, k is a constant (often, $k = 10$), and *infr* or *freq* denote "rare" and "frequent" species categories.
ICE-1 [37]	*Modified incidence-based coverage estimator (2013)* $\hat{S}_{ICE-1} = S(n)_{freq} + \frac{S(n)_{infr}}{\hat{C}(n)_{ICE}}\hat{\tilde{\gamma}}^2_{infr}$ where $\hat{\tilde{\gamma}}^2_{infr} = \max\left(\hat{\gamma}^2_{infr}\left(\frac{1+n(1-\hat{C}(n)_{ICE})\sum_{i=1}^{k} i(i-1)f_i(n)}{n(n-1)\hat{C}(n)_{ICE}}\right), 0\right)$
JK1 [15]	*First-order Jackknife estimator (1978)* $\hat{S}_{jk1} = S(n) + \frac{n-1}{n}f_1(n) \approx S(n) + f_1(n)$
JK2 [15]	*Second-order Jackknife estimator (1978)* $\hat{S}_{jk2} = S(n) + \frac{2n-3}{n}f1(n) - \frac{(n-2)^2}{n(n-1)}f_2(n) \approx S(n) + 2f_1(n) - f_2(n)$

Fig. 4: State-of-the-art species richness estimators under the Bernoulli-product model [17], [24].

Intuition. From Figure 4, we can easily see that all estimators are functions of the number of rare species (e.g., singletons f_1 and doubletons f_2). The underpinning insight is that the number of rare *and discovered* species (f_1 through f_k) are excellent predictors of even rarer *and undiscovered* species f_0. Rare species simply carry most information about unseen species [24].

Incidence-based coverage estimators (ICE) extend this idea of the improved Chao estimator *iChao* to consider more than just singletons and doubletons and integrate the recent understanding of sample coverage $C(n)$. When the individuals are independently sampled from an invariant species distribution, the complement of the missing mass probability given by Good-Turing estimator [34] (i.e. $\hat{C} = 1 - f_1/n$) provides a highly accurate estimate of the current sample coverage. By incorporating sample coverage, the *incidence-based coverage estimator* (ICE) was derived [43]. It assumes that any species distribution $\{p\}_{i=1}^{S}$ can be fully characterized by the coefficient of variation γ^2 and the mean incidence probability $1/S$ [20]. Under ICE, species are distinguished, using an arbitrary threshold k, as "frequent" and "rare" (i.e., infrequent) species. Using this distinction, we can obtain an accurate estimate of γ^2 using only the rare species category. However, the ICE estimator is known to underestimate for highly heterogeneous distribution with very high species richness; like the ones we encounter in fuzzing. A modified version, ICE-1 [37] is derived through standard statistical approximations for such *mega-diverse* communities; but with the same set of factors as ICE. For the modified ICE (*ICE-1*), the coefficient of variation $\hat{\tilde{\gamma}}^2$ is estimated as a factor of the coefficient of variation $\hat{\gamma}^2$ for the original ICE.

Jackknife (JK) estimators for species richness are instances of the general Jackknife method, which uses resampling to estimate certain statistical parameters and to reduce bias. The j^{th}-order Jackknife estimate (JK-j) is obtained by excluding j sample points from the original sample of size n. For instance, the first-order Jackknife $JK1$ of species richness [15] uses the Good-Turing estimator $1 - \widehat{C}(n) = f_1(n)/n$ of the probability of discovering (single) new species. *Assuming* this probability be equivalent for the reduced sample with $n-1$ sampling units, the expected number of undiscovered species f_0 is approximate $(n-1)f_1/n$.

IV. Experimental Setup

The *main objective* of this work is to evaluate the statistical method for approximating reachable coverage. Specifically, we evaluate how well the seven state-of-the-art species richness estimators in Figure 4) perform as approximators of the total number of coverage elements reachable during fuzzing. To evaluate estimator performance under different assumptions for the ground truth, we design three research questions.

A. Research Questions

RQ.1 (Ground truth). *For programs where we know the reachable coverage, how do existing species richness estimators (Figure 4) perform as approximators of reachable coverage?*

RQ.2 (Real world). *For real-world programs, how do existing estimators perform as approximators of reachable coverage?*

RQ.3 (Bootstrapped ground truth). *For real-world programs, where we "force" a known asymptote by bootstrapping, how do existing estimators perform?*

B. Variables and Measures

Our estimation target (i.e., *estimand*) is the total number of reachable basic blocks, i.e., the maximum achievable basic block coverage. For every estimator in Figure 4, we measure the mean bias and imprecision w.r.t. the estimand as a function of the number of test inputs n that have been generated throughout a fuzzing campaign. The *mean bias*(n) of an estimator $\hat{S}$ is the average degree to which $\hat{S}$ systematically over- or underestimates the estimand S:

$$bias(n) = \sum_{i=1}^{N} \frac{\hat{S}_i(n) - S}{NS}$$

Subject	Project	Version	LoC	# BBs
`readelf`	Binutils	2.29	22,347	18,578
`ffmpeg`	FFmpeg	n3.3.2	522,813	432,244
`ftfuzzer`	FreeType2	2.7	44,686	27,521
`gif2png`	Gift2png	2.5.3	988	700
`jasper`	JasPer	1.900.0	17,385	14,417
`jsoncpp_fuzz`	JsonCpp	1.8.4	7,251	5,938
Total			615,470	499,398

(a) Real-world. Open-source programs with unknown reachable coverage from the OSS-Fuzz [3] continuous fuzzing platform.

Subject	Version	LoC	# BBs
`tcas`	2.0	173	63
`totinfo`	2.0	565	132
`replace`	2.1	564	228
`schedule2`	2.0	374	138
`printtokens2`	2.0	570	198
Total		2246	1261

(b) **Ground-truth**. Programs with known reachable coverage from the Software-artifact Infrastructure Repository (SIR; by Siemens Corporate Research).

Fig. 5: Fuzztastic subject programs.

where N is the total number of repetitions. The *imprecision*(n) of an estimator $\hat{S}$ is the variance of the (individual) bias:

$$\textit{imprecision}(n) = \sqrt{\frac{\sum_{i=1}^{N}\left(\frac{\hat{S}_i(n)-S}{S} - \frac{\left[\sum_{i=1}^{N}\hat{S}_i(n)-S\right]}{NS}\right)^2}{N-1}}$$

Intuitively, an estimator with a high, negative mean bias and a low imprecision produces similar underestimates while an estimator with a low positive mean bias and a high imprecision generates many different estimates that, *on the average*, slightly over-estimate.

C. Benchmarks: Ground-Truth and Real-World

For our evaluation, we use two benchmarks integrated into Fuzztastic [44] fuzzer evaluation platform. Fuzztastic allows us to track hit counts and calculate the required quantities (f_k) used for estimation.[6] Fuzztastic records hit counts for every basic block, i.e., the number of fuzzer-generated inputs that exercise that basic block. From this hit count information, we can empirically measure coverage over time $S(n)$ as well as the number of rarely covered basic blocks (e.g., $f_1(n)$, $f_2(n)$).

Infrastructure. Each fuzzing campaign was run on one of four (4) virtual machines with 32x 2GHz x86_64 CPU cores, 32 GB of RAM, and 200 GB of disk space each. All VMs were running in the Nectar Research Cloud. To address randomness in the outcomes, we repeated each campaign at least 30 times.

Real-world. We selected six (6) open-source C programs that have already been integrated into Fuzztastic [44] and are widely used in practice (cf. Figure 5.a). These are programs or libraries that process binary, movie, font, image, and JSON files (in that order). However, as discussed in Section II-A, there is no indication of the maximum achievable coverage in log-scale time even for extremely long fuzzing campaigns. For these programs, we do not have the ground-truth value of the estimand S in order to evaluate the performance of the estimators $\hat{S}$. For this reason, we evaluate estimator performance on smaller-scale programs where the ground-truth estimand can be discovered.

Ground-truth. We selected the "Siemens programs"; five (5) C programs from the Software-artifact Infrastructure Repository (SIR) [1] that are widely used for fault localization research (cf. Figure 5.b). The Siemens programs are small enough that we expect to see the reachable basic block coverage achieved within a practical time limit. This allows us to establish the ground-truth estimand S and evaluate estimator performance with respect to the ground-truth S.

Fuzzer. We selected AFL++ [31] (version: 2.64c; `-m none`), a community-driven extension of the AFL fuzzer [55] which is also used in the OSS-Fuzz platform [3]. For the listed programs, Fuzztastic uses as *fuzz harnesses* the provided command line options for the subject programs and uses the *initial seeds* provided by AFL's GitHub repository.[7]

V. Empirical Evaluation of the Statistical Method to Approximate Reachable Coverage

RQ1. Ground Truth-Based Estimator Performance Evaluation

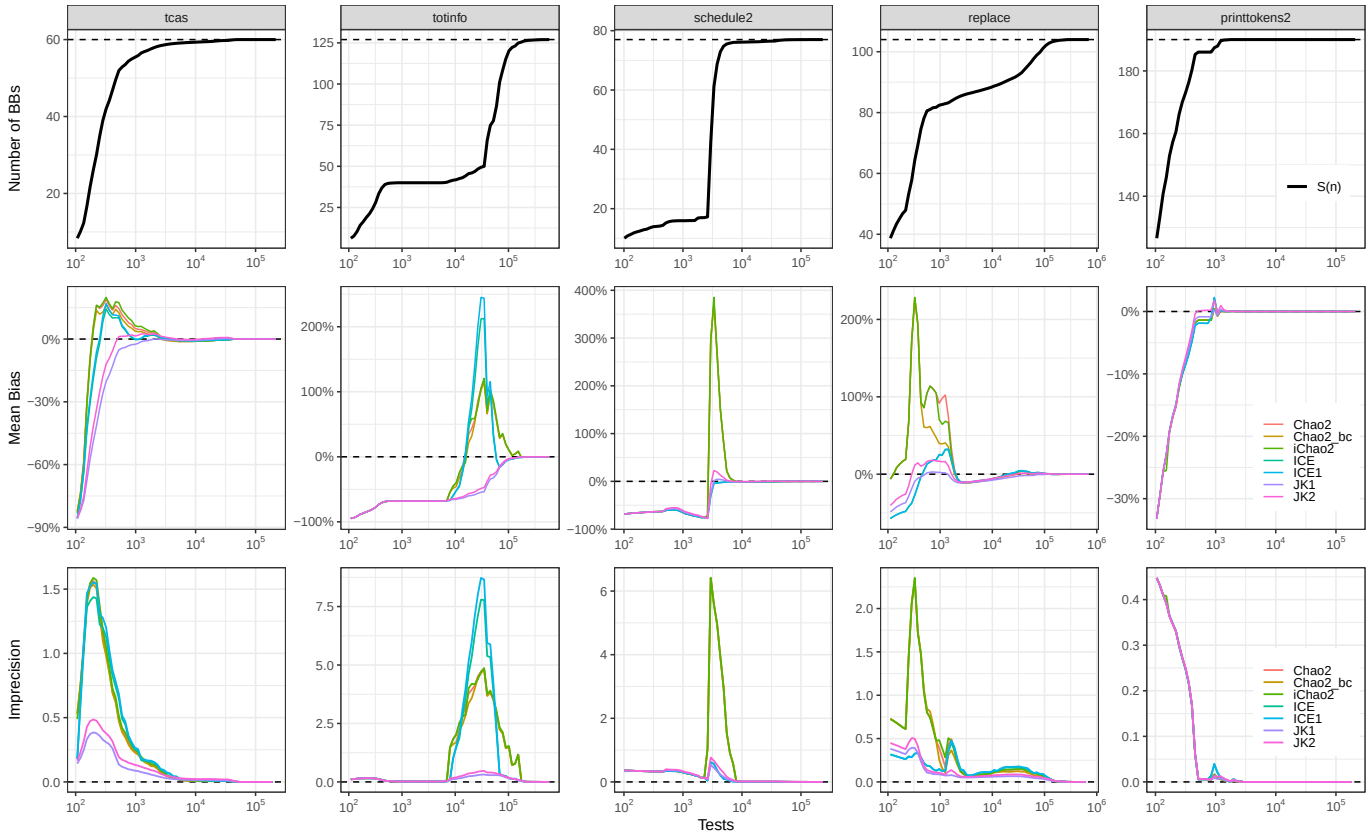
Ground-truth estimand S. Figure 6.a-top shows the basic block coverage as campaign length n increases on a log-scale. In all cases, we see basic block coverage saturate to the extent that increasing the number of generated test inputs *by one order of magnitude* does not change the current basic block coverage $S(n)$ anymore. For all programs, 100% reachable coverage is achieved before $n = 10^6$ test inputs are generated. Hence, it is reasonable to take $S(10^6)$ as the maximum achievable coverage S for our five Siemens programs and benchmark the estimators against this value.

Estimating S. Figure 6.a-middle shows the *estimator mean bias* for the Siemens programs. At the beginning of the campaign, all estimators exhibit a *negative* mean bias. For tcas, replace, and printtokens2, the negative bias is particularly obvious for the period where $S(n)$ is still increasing at a high rate. For totinfo and schedule2, the fuzzer seems to be stuck trying to resolve a coverage roadblock for one or two orders of magnitude until the coverage achieved $S(n)$ suddenly increases again. During this period the estimators maintain a fairly constant negative bias. One or two orders of magnitude before $S(n)$ reaches S all estimators start exhibiting a positive bias. They over-estimate S. However, over time the magnitude of the bias decreases and the estimate becomes highly accurate. The estimators are *asymptotically consistent*. We make a similar observation for *estimator variance* (Figure 6.a-bottom). In that region where the estimators tend to over-estimate, the variance (thus imprecision) is maximized, as well.

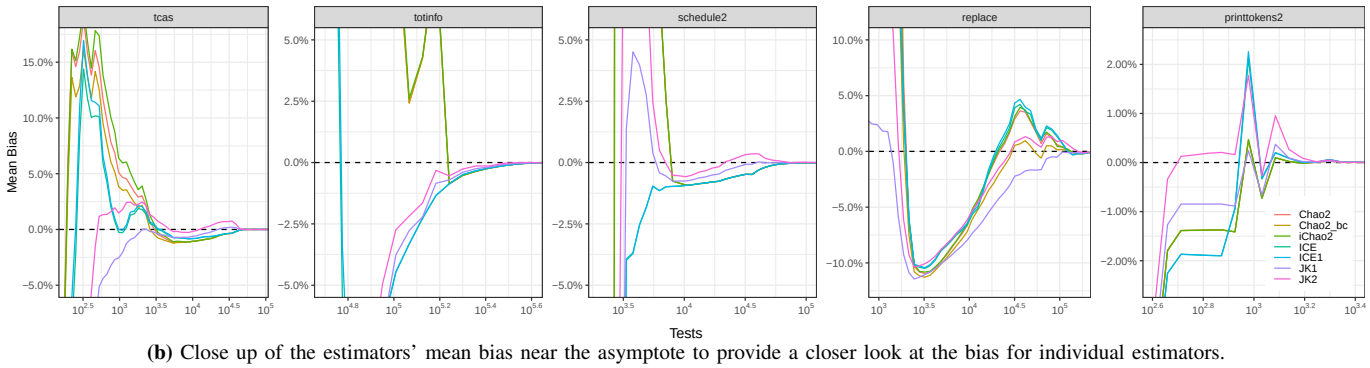
Overall, the *Jackknife estimators* (JK1, JK2) exhibit the lowest degree of bias and variance. Figure 6.b gives a close-up. We can see for the Siemens programs, the Jackknife estimates $\hat{S}$ are closer to the actual reachable coverage S than the

[6] Fuzzbench [45] does not provide hit count information.

[7] https://github.com/google/AFL/tree/master/testcases



(a) Basic block coverage $S(n)$, *mean bias*(n), and *imprecision*(n) for the estimators of S as the number n of generated test inputs increases (log-scale).



(b) Close up of the estimators' mean bias near the asymptote to provide a closer look at the bias for individual estimators.

Fig. 6: **Ground-truth**. Estimator performance in Siemens programs from the SIR (more than 990 repetitions for each program).

other estimators, both individually (low variance) and on the average (low mean bias). The incidence-based estimators (ICE, ICE1) appear on second place and perform almost identically. They outperform the Jackknife estimators (only) on schedule2. The improved Chao estimator (iChao2) performs worst and exhibits the largest bias and variance.

After an initial burn-in period, apart from iChao2, all estimators over-estimate the reachable coverage to within 15%, on average. All estimators are asymptotically consistent*, i.e., the magnitude of the bias reduces as n increases. For these programs that are small enough that the asymptote can be known, the Jackknife estimators perform best.*

Statically approximating S. How does a static approximation of the number S of reachable basic blocks stack up? We used the Static Value-Flow Analysis Framework (SVF) [51], a state-of-the-art static analysis framework for C that enables interprocedural dependence analysis and is able to perform pointer alias analysis, memory SSA form construction, value-flow tracking, and context-free-reachability analysis. Figure 7 shows the number of basic blocks (BBs) that SVF considers reachable, and the number of BBs that the fuzzer achieved upon saturation (see ground-truth estimand).

The result confirms our previous results in Section II-B: Static analysis over-counts reachable coverage. For instance, there is a difference of 3 BBs between the statically ap-

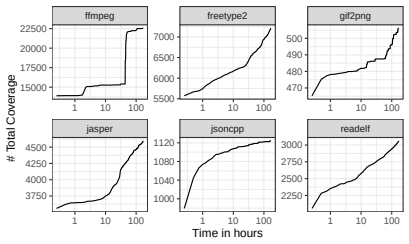
Subject	# Covered BBs by AFL++	# Reachable BBs Using Static PA	Total # BBs
tcas	60	63	63
totinfo	127	132	132
replace	104	228	228
schedule2	77	138	138
printtokens2	190	198	198

Fig. 7: Static approximation of reachable coverage.

proximated and the actual reachable coverage in `tcas`. We confirmed two as unreachable error conditions due to reading the input file. The third is manually *marked as unreachable* [2].

Even an advanced static analysis tool designed for inter-procedural reachability analysis that can handle pointer aliasing cannot *produce a precise approximation for our small benchmark programs. Because the static analysis tool cannot solve all possible path conditions, it can only conservatively over-approximate reachable coverage. In contrast to estimation, there are no systematic means to dynamically improve the accuracy of the approximation.*

RQ2. Real-World Estimator Performance Evaluation



Subject	# Reachable BBs Using Static PA
ffmpeg	8,850
freetype2	1,512
gif2png	700
jasper	13,712
jsoncpp	2,454
readelf	11,215

Fig. 8: Basic block (BB) coverage over time (3.5 CPU years) and SVF's static approximation of reachable BB coverage.

To investigate estimator performance in a more realistic setup where the ground-truth value of the reachable coverage S is *unknown*, we conducted 30 fuzzing campaigns for each of six (6) programs (cf. Figure 5.a) that lasted for seven (7) days. Figure 8.left shows the average basic block coverage achieved over the 1-week period and confirms our previous observation that the ground-truth reachable coverage can indeed *not* be identified (cf. Section II-A). Figure 8.right shows the static approximation of reachable coverage as produced by the SVF tool [51] and confirms our previous observation that static analysis over- and undercounts (cf. Section II-B).

Estimator performance. Figure 9 shows the coverage achieved $S(n)$ and our non-parametric estimates $\hat{S}$ of the reachable coverage for these fuzzing campaigns. Our main observation is that all seven state-of-the-art estimators predict "false peaks":[8] In most cases, more coverage can be achieved upon reaching the predicted maximum achievable (i.e., reachable) coverage. For instance, the current coverage $S(n)$ for the [1-week]-long campaign is often *higher* than

[8]In mountaineering, a *false peak* is a location where the climbers predict to be the summit; but upon reaching, it turns out the summit is higher.

Subject	Campaign Length	Current Coverage	Estimates of the Asymptote: *Chao2*	*Chao2 (bc)*	*iChao2*	*ICE*	*ICE-1*	*JK1*	*JK2*
ffmpeg	15 min	13900	13975	13968	13975	13992	13992	13943	13973
	3 hrs	15087	16011	15589	16012	15248	15249	15133	15176
	1 day	15296	17237	16932	17049	16241	16258	15362	15428
	1 week	22555	31047	28991	75424	24300	24304	22774	22685
freetype2	15 min	5573	6181	6137	6258	5734	5734	5715	5832
	3 hrs	5984	7456	7323	7541	6476	6476	6181	6352
	1 day	6294	6836	6806	7068	6629	6629	6439	6551
	1 week	7204	7498	7481	7410	7296	7297	7235	7257
gif2png	15 min	465	466	466	466	465	465	466	466
	3 hrs	479	480	480	480	479	479	480	480
	1 day	486	486	486	486	486	486	486	486
	1 week	506	506	506	506	506	506	506	506
jasper	15 min	3560	11213	9542	12201	4023	4023	3831	4092
	3 hrs	3671	4929	4429	4732	4344	4390	3697	3722
	1 day	3959	4517	4402	4190	4254	4256	4004	4046
	1 week	4584	4585	4585	4585	4586	4586	4585	4584
jsoncpp	15 min	979	1009	1003	1001	999	999	987	989
	3 hrs	1095	1095	1095	1095	1095	1095	1095	1095
	1 day	1113	1113	1113	1113	1113	1113	1113	1113
	1 week	1124	1124	1124	1124	1124	1124	1124	1124
readelf	15 min	2057	2527	2516	2719	2746	2746	2278	2446
	3 hrs	2431	2560	2548	2818	2510	2510	2492	2464
	1 day	2716	3105	3064	2782	3105	3105	2871	2963
	1 week	3056	3458	3451	3593	3629	3629	3311	3461

Fig. 9: The average basic block (BB) coverage for AFL++ campaigns of different length and the corresponding estimates of reachable BB coverage.

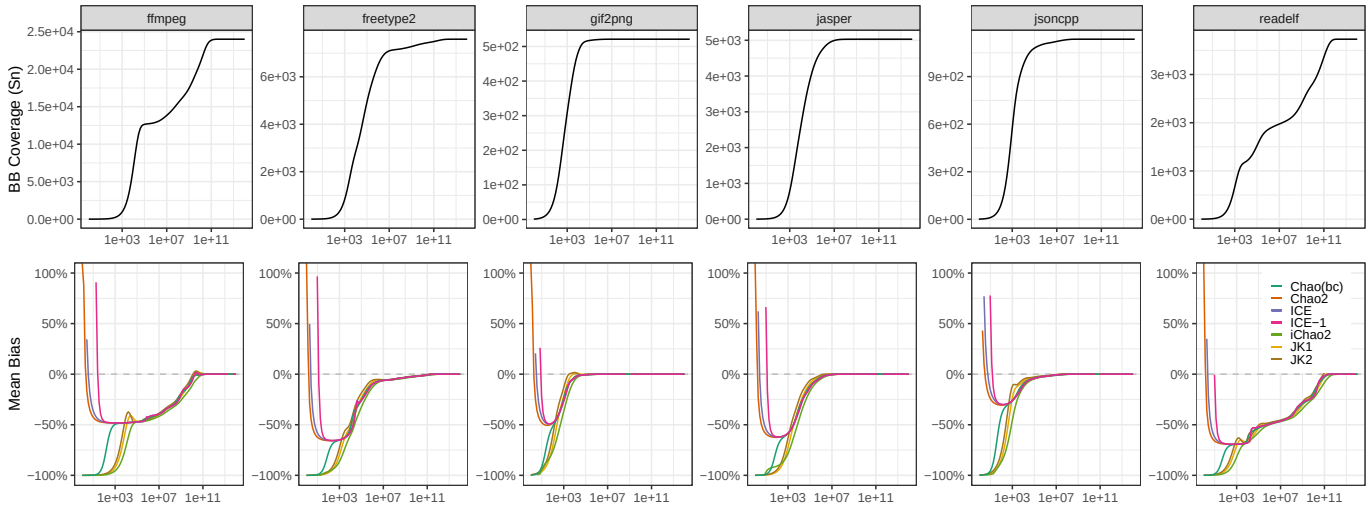
the reachable coverage $\hat{S}$ that was predicted by any of the estimators. Particularly for the Chao-type estimators (i.e., all except JK1 and JK2) this is reasonable since they were derived as lower bounds on the total number of species. However, this also means that none of these state-of-the-art estimators can be used as estimators of fuzzing effectiveness.

One possible explanation is that applied statisticians would consider our domain as *mega-diverse* [23]: There is a very large number of classes or species that we wish to tally, and most of these species are very rare. For such mega-diverse communities, the *ICE-1 estimator* was specifically developed as an estimator of species richness. So, we were particularly interested in its performance. However, compared to the baseline incidence- and coverage-based species richness estimator (ICE), the improved estimator (ICE-1) often only predicts a few more species with a similar magnitude in bias.

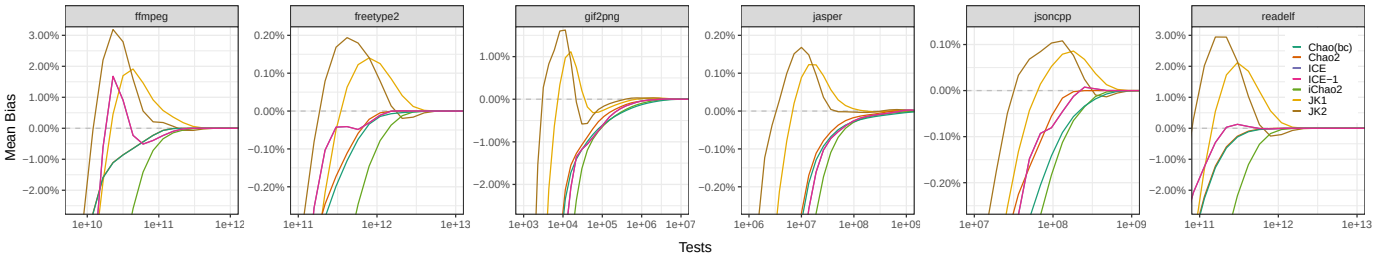
All evaluated estimators predict false peaks*: In most cases, more coverage can be achieved upon reaching the predicted reachable coverage. All estimators under-estimate S.*

RQ3. Real-World: Boostrapping Reachable Coverage

How can we evaluate estimator performance in the absence of a ground-truth? *Bootstrapping* is a powerful methodology in statistics to conduct statistical inference from an approximate (observed sample or empirical) distribution [46]. Bootstrapping is a resampling process with replacement to compute various population statistics, such as the confidence interval around a point estimate. The key intuition is that the relationship between the approximate (empirical) distribution and the theoretical population from which it is derived is



(a) Basic block coverage and average estimator bias as a function of the number of generated test inputs (log-scale).



(b) Close up of the estimators' mean bias near the asymptote of the bootstrapped BB coverage curve.

Fig. 10: Basic block coverage and estimator performance when bootstrapping $S(n)$ from the empirical distribution.

asymptotically equivalent to the relationship between the bootstrap samples and the empirical distribution. The Jackknife presented in Figure 4 is also based on the bootstrapping method (by excluding sampling points). Another advantage of bootstrapping is, due to resampling with replacement, we can control for adaptive bias prevalent in greybox fuzzing. In other words, adaptive bias cannot be an explanation if the estimators do not perform well. Lastly, we can directly work with expected values rather than random variables. For instance, we can directly compute $S(n) = S - \sum_{i=1}^{S}(1-p_i)^n$ instead of resampling n inputs from the multinomial.

Ground-truth. For every real-world program in Figure 5.a, we simulate the reachable coverage by bootstrapping from the empirical distribution. Concretely, we i) compute the probability p_i that an input exercises a basic block BB_i by dividing the hit count for BB_i by the tallied hit count across all BBs, and ii) resample from the resulting distribution. This "forces" a natural asymptote at the empirically observed number of BBs of a week-long fuzzing campaign and allows us to evaluate the estimators against this bootstrapped ground-truth asymptote. Indeed, in Figure 10, we observe the *bootstrapped* reachable coverage is achieved within a practical period.

Results. Figure 10 shows the performance for the estimators introduced in Figure 4 and evaluated for smaller programs with a *known* reachable coverage in RQ1. Our first observation is that the magnitude of the bias is substantial at the beginning of the campaign. Until about one order of magnitude *before* the asymptote is actually reached, there does not seem to be any information about the asymptote within the estimates. This confirms the *false-peaks* observation of RQ2 for the period before the asymptote of $S(n)$ is discernible. It is notable that the classic Chao estimator (Chao2) and the incidence-based estimators (ICE, ICE-1) substantially over-estimate very early in the fuzzing campaign when a few thousand test inputs have been generated.

> *At the beginning of the campaign when no asymptote is discernible in log-scale time, all estimators predict false peaks (as in RQ.2). However, the systematic negative bias reduces as the campaign length increases, demonstrating the asymptotic consistency of all estimators. After the "initial" burn-in period, all estimators start accurately predicting the reachable coverage to within $\pm 3\%$. Among evaluated estimators, Jackknifes perform best (as in RQ.1).*

Overall, the *Jackknife* estimators (JK1, JK2) exhibit the lowest degree of bias. Figure 10.b gives a close-up. We can see that second-order Jackknife (JK2) is the first to estimate reachable coverage within $\pm 3\%$ for the largest, smallest, and a medium size program, and to within $\pm 0.2\%$ for the remaining three programs. In the second place, we find the coverage-based estimators (ICE, ICE-1), which almost never over-

estimate, unlike the Jackknifes. On our fuzzing campaigns bootstrapped from the empirical distribution in real-world programs, the improved Chao estimator (iChao2) performs worst. In summary, this ranking of estimators *agrees with the ranking we have got in RQ1* for the Siemens programs where the reachable coverage is known (rather than bootstrapped).

VI. Threats to Validity

As for any empirical study, there are various threats to the validity of our results and conclusions.

One concern is *external* validity, i.e., the degree to which our study can be generalized to and across other fuzzers, static analysis tools, programs, and estimators. We evaluate coverage saturation in terms of branch coverage by running 11 popular fuzzers for 7 days on 32 real-world C programs started from the available initial corpus. The fuzzers represent a range of the most popular fuzzers. The programs cover a wide variety of domains and are part of a fuzzer benchmarking suite developed by Google [45]. The initial corpus was provided by the FuzzBench benchmarking platform. However, we note if the initial corpus were collected over many years of greybox fuzzing, an asymptote may be observed in practical time even for these large programs. Furthermore, we evaluate the performance of seven state-of-the-art species richness estimators for reachable coverage on five smaller toy programs and six real-world programs using the AFL++ fuzzer started from a single seed. As our statistical framework requires the fuzzer to be a stochastic process where inputs are sampled with replacement, we suggest that a similar study be conducted to assess estimator performance for fuzzers where this assumption does not hold, specifically symbolic execution-based whitebox fuzzers. However, the most widely-used fuzzers are greybox or blackbox. Since all fuzzers 11 fuzzers currently in FuzzBench are subject to the saturation illusion (Fig. 2), we do not expect major differences in estimator performance for similar fuzzers. We also checked and found that the distribution of hit counts (as input to the evaluated estimators) is similar across fuzzers.

Another concern is *internal validity*, i.e., the degree to which our study minimizes systematic error. Previous studies of species richness estimators evaluated estimator performance against the total number of species observed after a very large sampling effort. This assumes that saturation is possible with practical sampling effort. However, as we show in this paper, this assumption is invalid in the context of fuzzing. To mitigate this threat to internal validity, we investigate estimator performance (i) for small programs where the ground truth reachable coverage can be known, and (ii) for real-world programs where we develop and employ a new method to simulate reachable coverage by bootstrapping from the empirical distribution (cf. RQ3).

VII. Related Work

Effectiveness. Given a program and an oracle that distinguishes passing from failing executions, the *problem of automated software testing*, or fuzzing for short, is to generate inputs that witness failing executions. However, as Dijkstra notes, "program testing can be used to show the presence of bugs, but never to show their absence".[9] Hence, in order to *measure testing effectiveness* in the absence of bugs, it was proposed to measure other properties of a program, such as the number of structural code elements that are covered or the number of artificially injected faults that could be discovered (as in fault- or mutation-based testing [47], [41]). Indeed, recent empirical studies have shown that the code coverage achieved is a strong predictor of the bug finding ability of a fuzzer (i.e., there is a strong correlation) [11], [33], [36], [26].

Counting. Common to all effectiveness measures is *to count*. The more of something is covered, found, or exercised, the better the fuzzer. This general insight has previously been studied both theoretically and empirically under *partition testing*, where the input space is partitioned into (sometimes overlapping) subdomains [40], [30], [38], [25], [54]. We can think of the number of partitions, that the fuzzer can generate inputs for, as the *fuzzer's search space* (as opposed to the program's input space). For instance, the more program statements a fuzzer can exercise, the greater its search space. While previous work on partition testing was concerned with the optimal partitioning strategy, our work is concerned with the total number S of partitions in the fuzzer's search space, given a concrete partitioning strategy (e.g., a coverage criterion). If we can measure the fuzzer's search space, i.e., its reachable coverage, we can also measure the fuzzing campaign's progress towards completion (e.g., "99% of *reachable* statements are covered").

Finding unreachable code. We wish to count the number of reachable code elements. As we have established in Section I, precisely determining statically which code elements can be reached is at least as hard as the verification problem. Statically reachable coverage can only be approximated resulting in over- and under-counting problems (cf. Sec. II-B & V). Nevertheless, removing *un*reachable code to reduce the size of the compiled binary is a common compiler optimization, called *dead code elemination* [28], [52]. If the problem of identifying unreachable code complements the problem of identifying reachable code, why is dead code elimination so effective? Well, dead code elimination is under-approximated and does not remove all unreachable code. For instance, only by a short glance at the control-flow graph, we could find (unreachable) code that follows a `return` statement. Similar analysis rules exist for other common cases [51].

Finding reachable code. Given a program statement s, *static program slicing* allows us to compute all statements whose value *may* affect the execution or the value of s [53]. Starting from the slicing criterion, static program slicing computes the transitive closure of data- and control-flow dependencies in the program dependence graph. To establish the reachability of s by the fuzzer, we could check whether the statement holding the fuzzer's input is in the static backward slice w.r.t. s. However, static slicing is *over-approximate*.

[9] EWD249, §3: https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

Statements that are not members of the slice have *definitely* no impact on the reachability or value of the slicing criterion. Statements in the slice *may or may not* have an impact on the reachability or value of the slicing criterion. Despite a statement being a member of the slice, no execution may exist that contains both, the reported statement and the slicing criterion s. The resulting slices can be impractically large, sometimes containing the entire program [6]. In this work, we do not require program dependence graphs or static analysis to establish the reachable code. Instead, we evaluate the possibility of estimating the total number of coverage elements in the fuzzer's search space.

Estimation in fuzzing. While the development of more effective and efficient software testing techniques has a long history in software engineering research, the statistically sound estimation of pertinent properties has found significant interest only more recently. For instance, Böhme and Falk [9] empirically investigated and probabilisticly explained the *scalability* of fuzzing, i.e., the number of new species discovered within a given time budget as the number of available machines increases, where vulnerability discovery was found to be subject to an exponential cost. Böhme, Liyanage, and Wüstholz [10] investigated means to estimate the residual risk of a fuzzing campaign that has found no errors. Lam and Grunske [49] developed general estimators of the diversity of program behaviors excercised by a fuzzer. Previous work explored the probabilistic and information-theoretic foundations of *fuzzer efficiency* [12], [11], i.e., the rate at which a fuzzer discovers new species. In this paper, we have proposed the estimation of *fuzzer effectiveness*, i.e., the total number of coverage elements ("species") in the fuzzer's search space.

The STADS statistical framework [8] casts fuzzing as a stochastic process ammenable to estimation and discusses various estimators from biostatistics. The author already discusses some species richness estimators in the STADS paper, but only provides a preliminary evaluation of one estimator (Chao2) while we evaluate all seven (7) state-of-the-art estimators. As is customary for the evaluation of species richness estimators [24], [48], [20], the preliminary evaluation in the STADS paper [8] also *assumes* that the campaign is saturated after 24 hours and evaluates the performance of the estimator against this assumption. However, as we demonstrate by fuzzing 32 programs for 116 CPU years, *saturation is an illusion* in log-scale time (Sec.II-A). Hence, we develop a simulation methodology based on bootstrapping from the empirical distribution that allows us to "force" saturation for a sound evaluation of estimator performance (RQ.3) and investigate estimator performance for small programs where the asymptote can actually be (and is) known (RQ.1). In contrast to previous work, we also evaluate static analysis as a means to compute reachable coverage and study the problem of over- and under-counting (cf. Sec. II-B and RQ.1).

VIII. Discussion

The *reachable coverage* S for a fuzzer can be used to measure (a) the amount of untested code, (b) the effectiveness of a fuzzer, and (c) the completeness of an ongoing fuzzing campaign. We empirically studied two approaches to approximate S. *Static analysis* [16], [51] analyzes the program's source code *before* the fuzzer is run. *Estimation* analyzes a sample of program executions *while* the fuzzer is run.

Experiment ID	Commit ID	# of subjects	# of fuzzers	# of runs
2021-04-11-7d-paper	69b67c0	21	10	10
2021-04-23-7d-paper	1471300	11	11	10
2021-07-10-redo-7d	00dbe2a	4	11	10
2021-08-19-crash-s-7d	db192b6	10	11	10
Experiment infrastructure	https://gitlab.lrz.de/fuzztastic/			
Experiment data	https://www.fuzzbench.com/reports/[ExperimentID]			

(a) Fuzzbench reproducibility information

Repository	Commit ID
Repo fuzztastic-evaluations	dd51a2b8
Repo fuzztastic-llvm-pass	3b68483c
Experiment infrastructure	https://gitlab.lrz.de/fuzztastic/
Experiment data	https://anonymous.4open.science/r/reachable_coverage-experiments/data/experimental_data.zip

(b) Fuzztastic reproducibility information

Fig. 11: Reproducibility information.

For *static analysis*, we find that even state-of-the-art tools developed in industry and academia drastically over- or under-count reachable coverage. Fuzz-Introspector [16] is developed by Google to measure fuzzer effectiveness. SVF [51] is developed by Sui's research group to maximize precision during inter-procedural reachability analysis. Yet, their approximation cannot be reliably interpreted. In fact, the concrete degree of inaccuracy is hard to determine and *cannot* be dynamically improved simply by running the analysis for longer.

For estimation, we find that the Jackknife estimators [15] exhibit the least bias or variance compared to the other state-of-the-art estimators. The 1st-order Jackknife (JK1) is computed as the number of covered code elements added to the number of singletons $\hat{S} = S(n) + f_1(n)$. The mean difference between the estimate $\hat{S}$ and the actual reachable coverage S is within $\pm 3\%$ of S between one or two orders of magnitude *before* S is actually achieved. To exemplify this statement, if S was achieved in 30 days, then the Jacknifes would estimate $S \pm 3\%$ within about three (3) days. However, without knowing the actual value of S, we found that all estimators predict *false peaks* at the beginning of the fuzzing campaign: Upon reaching the predicted reachable coverage $\hat{S}$, there was always more coverage to be achieved, $S > \hat{S}$.

Assuming completeness, static analysis can only ever provide an *upper bound* on S (unless the verification problem is practical). In contrast, the evaluated state-of-the-art estimators can only provide a *lower bound*. As we find in this paper, in both cases, it is difficult to determine precisely the *degree* to which both methods over- or under-approximate S. However, unlike a static analysis, the estimation always allows us to improve the lower bound by running the campaign for longer. In practice, we recommend employing both methods.

Open Science. We make the experiment infrastructure and all data publicly available at https://anonymous.4open.science/r/reachable_coverage-experiments. Figure 11 shows more. We will submit the artifact to Zenodo for long-term archival.

References

[1] "Software-artifact infrastructure repository (sir)," https://sir.csc.ncsu.edu/php/previewfiles.php, accessed: 2022-08-20.

[2] "Tcas. unreachable coverage," https://github.com/lichcat/DyVerifyUsingCrest/blob/master/pgcrest/benchmarks/tcas/source/tcas.c#L129, accessed: 2022-08-20.

[3] *OSS-Fuzz: Google's Continuous Fuzzing Service for Open-Source Software*. Vancouver, BC: USENIX Association, aug 2017.

[4] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 263–272.

[5] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, p. 219–250, may 2014. [Online]. Available: https://doi.org/10.1002/stvr.1486

[6] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering Methodology*, vol. 16, no. 2, p. 8–es, apr 2007. [Online]. Available: https://doi.org/10.1145/1217295.1217297

[7] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and Reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9166552/

[8] M. Böhme, "STADS: Software testing as species discovery," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 2, pp. 7:1–7:52, Jun. 2018.

[9] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2020, pp. 747–758.

[10] M. Böhme, D. Liyanage, and V. Wüstholz, "Estimating residual risk in greybox fuzzing," ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 230–241. [Online]. Available: https://doi.org/10.1145/3468264.3468570

[11] M. Böhme, V. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2020, pp. 970–981.

[12] M. Böhme and S. Paul, "A probabilistic analysis of the efficiency of automated software testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 345–360, April 2016.

[13] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1–13.

[14] M. Branco, F. G. Figueiras, and P. Cermeño, "Assessing the efficiency of non-parametric estimators of species richness for marine microplankton," *Journal of Plankton Research*, vol. 40, no. 3, pp. 230–243, 03 2018.

[15] K. P. Burnham and W. S. Overton, "Estimation of the size of a closed population when capture probabilities vary among animals," *Biometrika*, vol. 65, no. 3, pp. 625–633, 1978.

[16] O. Chang, N. Emamdoost, A. Korczynski, and D. Korczynski, "Introducing fuzz introspector, an openssf tool to improve fuzzing coverage," https://openssf.org/blog/2022/06/09/introducing-fuzz-introspector-an-openssf-tool-to-improve-fuzzing-coverage/, accessed: 2022-08-15.

[17] A. Chao, K. H. Ma, T. C. Hsieh, and C. H. Chiu, "Online program spader (species-richness prediction and diversity estimation in r)," Program and User's Guide published at http://chao.stat.nthu.edu.tw/wordpress/software_download, 2015.

[18] A. Chao, "Nonparametric estimation of the number of classes in a population," *Scandinavian Journal of Statistics*, vol. 11, no. 4, pp. 265–270, 1984.

[19] ——, "Estimating the population size for capture-recapture data with unequal catchability," *Biometrics*, vol. 43, no. 4, pp. 783–791, 1987.

[20] ——, "Species estimation and applications," *Wiley StatsRef: Statistics Reference Online*, 2014.

[21] A. Chao, R. L. Chazdon, R. K. Colwell, and T.-J. Shen, "A new statistical approach for assessing similarity of species composition with incidence and abundance data," *Ecology letters*, vol. 8, no. 2, pp. 148–159, 2005.

[22] A. Chao, C.-H. Chiu, R. K. Colwell, L. F. S. Magnago, R. L. Chazdon, and N. J. Gotelli, "Deciphering the enigma of undetected species, phylogenetic, and functional diversity based on good-turing theory," *Ecology*, vol. 98, no. 11, pp. 2914–2929, 2017.

[23] A. Chao, C.-H. Chiu, and L. Jost, "Statistical challenges of evaluating diversity patterns across environmental gradients in mega-diverse communities," *Journal of Vegetation Science*, vol. 27, no. 3, pp. 437–438, 2016.

[24] A. Chao and R. K. Colwell, "Thirty years of progeny from chao's inequality: Estimating and comparing richness with incidence data and incomplete sampling," *Statistics and Operations Research Transactions*, vol. 41, no. 1, pp. 3–54, 2017.

[25] T. Y. Chen and Y.-T. Yu, "On the expected number of failures detected by subdomain testing and random testing." *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, 1996.

[26] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size," in *Proceedings of the ACM International Conference on Automated Software Engineering*, 2020, pp. 237–249. [Online]. Available: https://doi.org/10.1145/3324884.3416667

[27] C.-H. Chiu, Y.-T. Wang, B. A. Walther, and A. Chao, "An improved nonparametric lower bound of species richness via a modified good–turing frequency formula," *Biometrics*, vol. 70, no. 3, pp. 671–682, 2014.

[28] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[29] R. M. Dorazio and J. Andrew Royle, "Mixture models for estimating the size of a closed population when capture rates vary among individuals," *Biometrics*, vol. 59, no. 2, pp. 351–364, 2003.

[30] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 438–444, Jul. 1984.

[31] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *Proceedings of the USENIX Workshop on Offensive Technologies*, 2020.

[32] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[33] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing Non-adequate Test Suites Using Coverage Criteria," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2013, pp. 302–313. [Online]. Available: https://doi.org/10.1145/2483760.2483769

[34] I. J. Good, "The population frequencies of species and the estimation of population parameters," *Biometrika*, vol. 40, no. 3-4, pp. 237–264, 1953.

[35] Google, "Oss-fuzz: Continuous fuzzing for open source software," https://github.com/google/oss-fuzz#trophies, accessed: 2022-08-12.

[36] R. Gopinath, C. Jensen, and A. Groce, "Code Coverage for Suite Evaluation by Developers," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 72–82. [Online]. Available: https://doi.org/10.1145/2568225.2568278

[37] N. J. Gotelli and A. Chao, "Measuring and estimating species richness, species diversity, and biotic similarity from sampling data," 2013.

[38] W. J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, Sep. 1999.

[39] D. C. Gwinn, M. S. Allen, K. I. Bonvechio, M. V. Hoyer, and L. S. Beesley, "Evaluating estimators of species richness: the importance of considering statistical error rates," *Methods in Ecology and Evolution*, vol. 7, no. 3, pp. 294–302, 2016.

[40] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence [program testing]," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402–1411, Dec 1990.

[41] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[42] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the Conference on Computer and Communications Security*. New York, NY, USA: ACM, oct 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[43] S.-M. Lee and A. Chao, "Estimating population size via sample coverage for closed capture-recapture models," *Biometrics*, pp. 88–97, 1994.

[44] S. Lipp, D. Elsner, T. Hutzelmann, S. Banescu, A. Pretschner, and M. Böhme, "FuzzTastic: A fine-grained, fuzzer-agnostic coverage ana-

lyzer," in *Proceedings of the 44th International Conference on Software Engineering Companion*, ser. ICSE'22 Companion, 2022, pp. 1–5.
[45] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932
[46] C. Z. Mooney, C. F. Mooney, C. L. Mooney, R. D. Duval, and R. Duvall, *Bootstrapping: A nonparametric approach to statistical inference*. sage, 1993, no. 95.
[47] L. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 08, pp. 844–857, aug 1990.
[48] W. E. Nagy and R. C. Anderson, "How many words are there in printed school english?" *Reading research quarterly*, pp. 304–330, 1984.
[49] H. L. Nguyen and L. Grunske, "Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1–13.
[50] G. C. Reese, K. R. Wilson, and C. H. Flather, "Performance of species richness estimators across assemblage types and survey parameters," *Global Ecology and Biogeography*, vol. 23, no. 5, pp. 585–594, 2014.
[51] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.
[52] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
[53] M. Weiser, "Program slicing," ser. ICSE '81. IEEE Press, 1981, p. 439–449.
[54] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703–711, Jul 1991.
[55] M. Zalewski, "American fuzzy lop (afl)," https://lcamtuf.coredump.cx/afl/, accessed: 2021-03-12.