



# Boosting Fuzzer Efficiency: An Information Theoretic Perspective

Marcel Böhme
Monash University, Australia
marcel.boehme@acm.org

Valentin J.M. Manès
CSRC, KAIST, Korea
valentinmanes@outlook.fr

Sang Kil Cha
CSRC, KAIST, Korea
sangkilc@kaist.ac.kr

## ABSTRACT

In this paper, we take the fundamental perspective of fuzzing as a learning process. Suppose before fuzzing, we know nothing about the behaviors of a program $\mathcal{P}$: What does it do? Executing the first test input, we learn how $\mathcal{P}$ behaves for this input. Executing the next input, we either observe the same or discover a new behavior. As such, each execution reveals "some amount" of information about $\mathcal{P}$'s behaviors. A classic measure of information is Shannon's entropy. Measuring entropy allows us to quantify how much is learned from each generated test input about the behaviors of the program. Within a probabilistic model of fuzzing, we show how entropy also measures fuzzer efficiency. Specifically, it measures the general *rate* at which the fuzzer discovers new behaviors. Intuitively, *efficient fuzzers maximize information.*

From this information theoretic perspective, we develop Entropic, an entropy-based power schedule for greybox fuzzing which assigns more energy to seeds that maximize information. We implemented Entropic into the popular greybox fuzzer LibFuzzer. Our experiments with more than 250 open-source programs (60 *million* LoC) demonstrate a substantially improved efficiency and *confirm* our hypothesis that an efficient fuzzer maximizes information. Entropic has been independently evaluated and invited for integration into main-line LibFuzzer. Entropic now runs on more than 25,000 machines fuzzing hundreds of security-critical software systems simultaneously and continuously.

## CCS CONCEPTS

• **Software and its engineering → Software testing**.

## KEYWORDS

software testing, fuzzing, efficiency, information theory, entropy

**ACM Reference Format:**
Marcel Böhme, Valentin J.M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409748


*ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA*

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409748

## 1 INTRODUCTION

Due to its efficiency, fuzzing has become one of the most successful vulnerability discovery techniques. For instance, in the three years since its launch, the ClusterFuzz project alone has found about 16,000 bugs in the Chrome browser and about 11,000 bugs in over 160 open source projects—only by fuzzing [1]. A fuzzer typically generates random inputs for the program and reports those inputs that crash the program. But, what *is* fuzzer efficiency?

In this paper, we take an information-theoretic perspective and understand fuzzing as a learning process.[1] We argue that a fuzzer's efficiency is determined by the average *information* that each generated input reveals about the program's behaviors. A classic measure of information is Shannon's entropy [39]. If the fuzzer exercises mostly the same few program behaviors, then Shannon's entropy is small, the information content for each input is low, and the fuzzer is not efficient at discovering new behaviors. If however, most fuzzer-generated inputs exercise previously unseen program behaviors, then Shannon's entropy is high and the fuzzer performs much better at discovering new behaviors.

We leverage this insight to develop the first entropy-based power schedule for greybox fuzzing. Entropic assigns more energy to seeds revealing more information about the program behaviors. The schedule's objective is to maximize the efficiency of the fuzzer by maximizing entropy. A *greybox fuzzer* generates new inputs by slightly mutating so-called seed inputs. It adds those generated inputs to the corpus of seed inputs which increase code coverage. The *energy* of a seed determines the probability with which the seed is chosen. A seed with more energy is fuzzed more often. A *power schedule* implements a policy to assign energy to the seeds in the seed corpus. Ideally, we want to assign most energy to those seeds that promise to increase coverage at a maximal rate.

We implemented our entropy-based power schedule into the popular greybox fuzzer LibFuzzer [27] and call our extension Entropic. LibFuzzer is a widely-used greybox fuzzer that is responsible for the discovery of several thousand security-critical vulnerabilities in open-source programs. Our experiments with more than 250 open-source programs (60 *million* LoC) demonstrate a substantially improved efficiency and *confirm* our hypothesis that an efficient fuzzer maximizes information.

Entropic has been evaluated independently by the developers of LibFuzzer, and with further improvements is now enabled by default. On Fuzzbench [30], Entropic turns LibFuzzer from the least well performing to the best performing fuzzer. LibFuzzer powers Google's OSSFuzz [31] and ClusterFuzz [1], as well as Microsoft's OneFuzz [11] fuzzing platforms. OSSFuzz alone fuzzes more than 350 often critical open source projects on more than 25,000 machines simultaneously and continuously.

[1] As in, learning about the colors in an urn full of colored balls by sampling from it.

In order to stand our information theoretic perspective on solid foundations, we explore a probabilistic model of the fuzzing process. We demonstrate how Shannon's entropy quantifies the rate at which a non-deterministic blackbox fuzzer discovers new behaviors in an (hypothetically) infinitely long fuzzing campaign. Simply speaking, if we are interested in achieving code coverage, Shannon's entropy quantifies the gradient of the coverage increase. We also show how non-deterministic greybox fuzzing can be reduced to a series of non-deterministic blackbox fuzzing campaigns, which allows us to derive the local entropy for each seed and the *current* global entropy for the fuzzer. To efficiently approximate entropy, we introduce several statistical estimators.

Our information-theoretic model explains the performance gains of our entropy-based power schedule for greybox fuzzing. Our schedule assigns more energy to seeds that have a greater local entropy, i.e., that promise to reveal more information. However, we also note that this information-theoretic model does not immediately apply to deterministic fuzzers which systematically enumerate all inputs they can generate, or whitebox fuzzers that systematically enumerate all (interesting) paths they can explore. For our probabilistic model to apply, the fuzzer should generate inputs in a random manner (with replacement). If a deterministic phase is followed by a non-deterministic phase (e.g., in the default configuration of the AFL greybox fuzzer), we can compute Shannon's entropy only for the non-determinstic phase.

*In summary*, our work makes the following contributions:

- We develop an information-theoretic foundation for non-deterministic fuzzing which studies the average information each test reveals about a program's behaviors.
- We formally link Shannon's entropy to a fuzzer's behavior discovery rate, i.e., we establish efficiency as an information-theoretic quantity.
- We introduce several practical estimators of information that are useful in the context of fuzzing.
- We present the first entropy-based power schedule to boost the efficiency of greybox fuzzers.
- We provide an open-source implementation, called Entropic, which will be linked here upon acceptance.
- We present a substantial empirical evaluation on over 250 widely-used, open-source C/C++ programs producing over 2 CPU years worth of data. Our data and R scripts are available here: https://doi.org/10.6084/m9.figshare.12415622.v2.

## 2 A PROBABILISTIC FRAMEWORK FOR BLACKBOX FUZZING

Fuzzing is an automatic software testing technique where the test inputs are generated in a random manner. Based on the granularity of the runtime information that is available to the fuzzer, we can distinguish three fuzzing approaches. A *blackbox fuzzer* does not observe or react to any runtime information. A *greybox fuzzer* leverages coverage or other feedback from the program's execution to dynamically steer the fuzzer. A *whitebox fuzzer* has a perfect view of the execution of an input. For instance, symbolic execution enumerates interesting program paths.

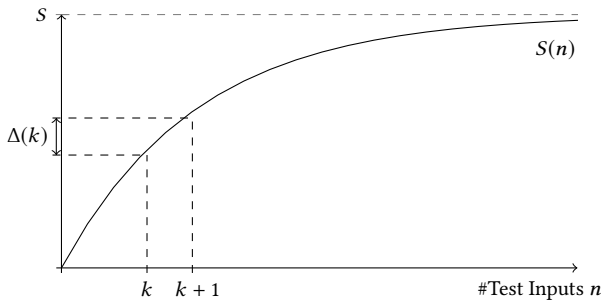



**Figure 1: The expected number $S(n)$ of program behaviors that a test generator discovers as the number of generated test inputs $n$ increases. The rate $\Delta(n) = S(n+1) - S(n)$ at which behaviors are discovered decreases over time. Function $\Delta(n)$ gives the *current* discovery rate when the $n$-th test is generated. In the limit, the expected number of behaviors discovered approaches the asymptote $S$.**

**Non-deterministic blackbox fuzzing** lends itself to probabilistic modeling because of the small number of assumptions about the fuzzing process. Unlike greybox fuzzing, blackbox fuzzing is not subject to adaptive bias.[2] We adopt the recently proposed STADS probabilistic model for non-deterministic blackbox fuzzing [6]: Each generated input can belong to one or more species. Beyond the probabilistic model, STADS provides biostatistical estimators, e.g., to estimate, after several hours of fuzzing, the probability of discovering a new species or the total number of species.

### 2.1 Software Testing as Discovery of Species

Let $\mathcal{P}$ be the program that we wish to fuzz. We call as $\mathcal{P}$'s *input space* $\boldsymbol{\mathcal{D}}$ the set of all inputs that $\mathcal{P}$ can take in. The fuzzing of $\mathcal{P}$ is a stochastic process

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

of sampling $N$ inputs *with replacement* from the program's input space. We call $\mathcal{F}$ as *fuzzing campaign* and a tool that performs $\mathcal{F}$ as a *non-deterministic blackbox fuzzer*.

Suppose, we can subdivide the search space $\boldsymbol{\mathcal{D}}$ into $S$ individual subdomains $\{\mathcal{D}_i\}_{i=1}^{S}$ called *species* [6]. An input $X_n \in \mathcal{F}$ is said to *discover* species $\mathcal{D}_i$ 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 sampled for the first time). An *input's species* is defined based on the dynamic program properties of the input's execution. For instance, each branch that is exercised by input $X_n \in \boldsymbol{\mathcal{D}}$ can be identified as a species. The discovery of the new species then corresponds to an increase in branch coverage.

**Global species discovery**. We let $p_i$ be the probability that the $n$-th generated input $X_n$ belongs to species $\mathcal{D}_i$,

$$p_i = P[X_n \in \mathcal{D}_i] \tag{2}$$

[2]Unlike for a blackbox fuzzer, for a greybox fuzzer the probability to observe certain program behaviors during fuzzing changes as new seeds are added to the corpus.

for $i : 1 \leq i \leq S$ and $n : 1 \leq n \leq N$. We call $\{p_i\}_{i=1}^{S}$ the fuzzer's *global species distribution*. The expected number of discovered species $S(n)$ can be derived as

$$S(n) = \sum_{i=1}^{S} \left[1 - (1 - p_i)^n\right] = S - \sum_{i=1}^{S} (1 - p_i)^n. \tag{3}$$

Figure 1 shows an example of a *species discovery curve* $S(n)$, i.e., the number of species covered with $n$ generated test cases. We can show that the number of species $S$ that the fuzzer discovers in the limit, i.e., the *asymptotic total number of species* is given as

$$S = \lim_{n \to \infty} S(n). \tag{4}$$

The *discovery rate* $\Delta(n)$, i.e., the expected number of species discovered with the $(n+1)$-th generated test input is defined as $\Delta(n) = S(n+1) - S(n)$. Discovery rate $\Delta(n)$ provides an excellent stopping rule [3, 40]. One could abort fuzzing when an insufficient progress is made for a long period of time, i.e., when $\Delta(n) < \theta$ for some $\theta$.

## 2.2 Mutation-Based Blackbox Fuzzing

We extend the STADS framework with a model for mutation-based fuzzing. Let $C$ be a set of seed inputs, called the seed corpus and $q_t$ be the probability that the fuzzer chooses the seed $t \in C$.[3] For each seed $t$, let $\boldsymbol{\mathcal{D}}^t$ be the set of all inputs that can be generated by applying the available mutation operators to $t$. The mutational fuzzing of $t$ is a stochastic process

$$\mathcal{F}^t = \left\{X_n^t \mid X_n^t \in \boldsymbol{\mathcal{D}}^t\right\}_{n=1}^{N^t} \tag{5}$$

of sampling $N^t$ inputs *with replacement* by random mutation of the seed $t$. We call all species that can be found by fuzzing a seed $t$ as the species in $t$'s *neighborhood*.

**Local species discovery**. We let $p_i^t$ be the probability that the $n$-th input $X_n^t$ which is generated by mutating the seed $t \in C$ belongs to species $\mathcal{D}_i$,

$$p_i^t = P[X_n^t \in \mathcal{D}_i] \tag{6}$$

for $i : 1 \leq i \leq S$ and $n : 1 \leq n \leq N$. We call $\{p_i^t\}_{i=1}^{S}$ the *local species distribution* in the neighborhood of the seed $t$. For a locally *unreachable* species $\mathcal{D}_j$, we have $p_j^t = 0$. Note that global and local distributions, by the law of total expectation, are related as

$$p_i = \sum_{t \in C} q_t \cdot p_i^t. \tag{7}$$

for $i : 1 \leq i \leq S$ such that by Equation (3),

$$S(n) = \sum_{i=1}^{S} \left[1 - \left(1 - \sum_{t \in C} q_t \cdot p_i^t\right)^n\right]. \tag{8}$$

is the species discovery curve for a mutation-based blackbox fuzzer.

[3]This probability is also called the seed's *energy*, *weight*, or *perf_score* (AFL).

## 2.3 Assumptions

For our probabilistic model of non-deterministic (mutational) blackbox fuzzing, we require that global and local species distributions are *invariant* throughout the fuzzing campaign, i.e.,

$$p_i = P[X_n \in \mathcal{D}_i] = P[X_{n+1} \in \mathcal{D}_i] \quad \text{and} \tag{9}$$

$$p_i^t = P[X_n^t \in \mathcal{D}_i] = P[X_{n+1}^t \in \mathcal{D}_i] \tag{10}$$

for $i : 1 \leq i \leq S$ and $n : 1 \leq n < N$, where $q_t$ is the probability that the fuzzer chooses the seed $t \in C$, where $X_n$ and $X_{n+1}$ are the $n$-th and $(n+1)$-th test inputs that the fuzzer generates, respectively, and where $X_n^t$ and $X_{n+1}^t$ are the $n$-th and $(n+1)$-th test inputs generated by fuzzing the seed $t$, respectively.

This is satisfied if, for any program input $d \in \mathcal{D}$ in the program's input space, we have that $P[X_n = d] = P[X_{n+1} = d]$, i.e., the probability to generate some input $d$ is invariant throughout the campaign. Our model accommodates that a blackbox fuzzer may generate inputs from a non-uniform distribution, i.e., for any two inputs $d_1, d_2 \in \mathcal{D}$, it is entirely possible that the probability that $n$-th test input is $d_1$ or $d_2$ differs, i.e., $P[X_n = d_1] \neq P[X_n = d_2]$.

For random testing tools and for generation- or mutation-based blackbox fuzzers that generate inputs by some random process it is *realistic* to assume that the probability to sample from a subdomain $\mathcal{D}_i \subseteq \boldsymbol{\mathcal{D}}$ does *not* change during the fuzzing campaign. Without any dynamic program feedback, a non-deterministic blackbox fuzzer has no reason to vary its fuzzing heuristics *during* the campaign. A mutation-based blackbox fuzzer usually has a fixed-size seed corpus $C$ and fixed-size set of mutation operators.

Otherwise, we make *no assumptions* about the number $S$, relative abundance $\{p_i\}_{i=0}^{S}$, or distribution of species in the fuzzer's search space. Specifically, there is no assumption that species are distributed equally. Some rare species (i.e., $p_i$ is very small) may well be clustered within a small region of the input space.

# 3 AN INFORMATION-THEORETIC MEASURE OF FUZZER EFFICIENCY

We provide an information-theoretic foundation for non-deterministic blackbox fuzzing. In the context of fuzzing, Shannon's entropy $H$ has several interpretations. It quantifies the average information a generated test input *reveals* about the behaviors (i.e., species) of the program. Alternatively we say, a generated test input reduces our uncertainty by $H$ information units (e.g., nats or bits), on average. Entropy also gives the minimum number of information units needed to reliably store the entire set of behaviors the fuzzer is capable of testing. Moreover, entropy is a measure of diversity. A low entropy means that the program does either not exhibit many behaviors or most generated inputs test the same behaviors (i.e., belong to an abundant species).

In this work, we show how entropy quantifies the efficiency of a blackbox fuzzer in terms of the species discovery rate. However, there are *several challenges* that we need to address. First, Shannon's entropy is defined in the case where each input belongs to exactly one species. How can we define an entropy when an input can belong to multiple species? Second, we cannot determine all the species of $\mathcal{P}$ and their probabilities with regards to $\mathcal{F}$ unless we

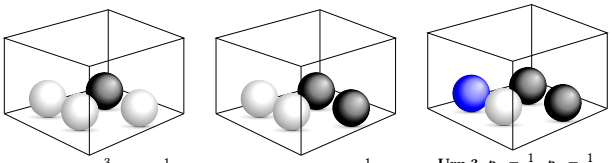



**Figure 2: Learning colors by sampling with replacement.**

know test cases for each species beforehand.[4] How can we quickly compute an approximate entropy? Third, we develop information-theoretic boosting for greybox fuzzing. Yet, for greybox fuzzers the probabilities $\{p_i\}_{i=1}^{S}$ can vary throughout the fuzzing campaign, which violates our assumptions (Sec.2.3). Is there a practical way to enforce our assumptions for a greybox fuzzer?

## 3.1 Information Theory in a Nutshell

Shannon's entropy $H$ [39] measures the average amount of information in each sample $X_n \in \mathcal{F}$ about the species that can be observed by executing the program $\mathcal{P}$. When there are $S$ distinct species, the entropy $H$ is:

$$H = -\sum_{i=1}^{S} p_i \log(p_i). \tag{11}$$

Figure 2 illustrates the concept informally. Each color represents a different species. We learn about the colors in each urn by sampling. Just how much we learn from each sampling differs from urn to urn. For instance, in Urn 1 it is three times more likely to draw a white ball than a black. It takes more attempts to learn about black balls in Urn 1 compared to Urn 2. Hence, we expect less information about the urn's colors in a draw from Urn 1. In fact, given the same number of colors $S$, the entropy is maximal when all colors are equiprobable $p_1 = \ldots = p_S$. Among the three urns, we expect to get the maximal amount of information about the urn's colors by drawing from Urn 3. Even though there is still a dominating color (black), there is now an additional color (blue) which can be discovered.

## 3.2 If Each Input Belongs to Multiple Species

Shannon's entropy is defined for the multinomial distribution where each input belongs to exactly one species (e.g., exercise exactly one path). However, an input can belong to several species so that $\sum_{i=1}^{S} p_i \geq 1$. For instance, considering a branch in $\mathcal{P}$ as a species, each input exercises multiple branches. The top-level branch is exercised with probability one. When it is possible that $\sum_{i=1}^{S} p_i \geq 1$, Chao et al. [15] and Yoo et al. [45] suggest to *normalize* the probabilities and compute $H = -\sum_{i=1}^{S} p'_i \log(p'_i)$, such that $p'_i = p_i / \sum_{j=1}^{S} p_j$. This normalization maintains the fundamental properties of information based on which Shannon developed his formula, i.e., that information due to independent events is additive, that information is a non-negative quantity, etc. The *normalized entropy* $H$ is computed as

$$H = -\sum_{i=1}^{S} p'_i \log(p'_i) = \log\left(\sum_{j=1}^{S} p_j\right) - \frac{\sum_{i=1}^{S} p_i \log(p_i)}{\sum_{j=1}^{S} p_j} \tag{12}$$

We refer to the Appendix for the derivation of this formula. We note that Equation (12) reduces to Equation (11) for the special case where $\sum_{j=1}^{S} p_j = 1$. We also note that the resulting quantity is technically *not* the average information per input.

## 3.3 The Local Entropy of a Seed

Recall from Section 2.2, that we call the probabilities $\{p_i^t\}_{i=1}^{S}$ that fuzzing a seed $t \in C$ generates an input that belongs to species $\mathcal{D}_i$ as the *local species distribution* of $t$. Moreover, we call the set of species $\{\mathcal{D}_i \mid p_i^t > 0 \wedge 1 \leq i \leq S\}$ as the *neighborhood* of the seed $t$. From the local species distribution of $t$, we can compute the *local entropy* $H^t$ of $t$ as a straight-forward application of Equation 12,

$$H^t = \log\left(\sum_{j=1}^{S} p_j^t\right) - \frac{\sum_{i=1}^{S} p_i^t \log(p_i^t)}{\sum_{j=1}^{S} p_j^t}. \tag{13}$$

The local entropy $H^t$ of $t$ quantifies the information that fuzzing $t$ reveals about the species in $t$'s neighborhood.

## 3.4 Information-Theoretic Efficiency Measure

Intuitively, the rate at which we learn about program behaviors, i.e., the species in the program, also quantifies a blackbox fuzzer's efficiency. We formally demonstrate how Shannon's entropy $H$ characterizes the general discovery rate $\Delta(n)$ as follows.

Theorem 1. *Let Shannon's entropy be defined as in Equation (12). Let $\Delta(n)$ be the expected number of new species the fuzzer discovers with the $(n+1)$-th generated test input, then*

$$H = \log(c) + \sum_{n=1}^{\infty} \frac{\Delta(n)}{cn} \tag{14}$$

*characterizes the rate at which species are discovered in an infinitely long-running campaign, where $c = \sum_{j=1}^{S} p_j$ is a normalizing constant.*

Proof. We refer to the Appendix for the proof. ■

According to Theorem 1, entropy measures the *species discovery rate* $\Delta(n)$ *over an infinitely long-running fuzzing campaign* where discovery is gradually discounted as the number of executed tests $n$ goes to infinity. In Figure 1, notice that $\Delta(n) \geq 0$ for *all* $n \geq 0$. If we simply took the sum of $\Delta(n)$ over all $n$, we would compute the total number of species $S = \sum_{n=1}^{\infty} \Delta(n)$. However, $S$ provides no insight on the *efficiency* of the discovery process. Instead, the diminishing factor $1/n$ in Equation (14) reduces the contribution of species discovery as testing effort $n$ increases. The number of species discovered at the *beginning* of the campaign has a higher contribution to $H$ then the number of species discovered later. In other words, a shorter "ramp up" time yields a higher entropy.

## 3.5 Maximum Likelihood Estimator

We estimate Shannon's entropy $H$ based on how often we have seen each observed species. The *incidence frequency* $Y_i$ for species $\mathcal{D}_i$ is the number of generated test inputs that belong to $\mathcal{D}_i$. Undetected

[4] Determining Shannon's entropy $H$ may require probabilistic symbolic execution which involves constraint solving and model counting [18, 20].

species yield $Y_i = 0$. An unbiased estimator of the local discovery probability $p_i$ is $\hat{p}_i = Y_i/n$, where $n$ is the total number of generated test inputs. By plugging $\hat{p}_i$ into Equation (12), we can estimate the entropy $H$ using *maximum likelihood estimation*. In our model, the estimated entropy $\hat{H}_{\text{MLE}}$ of $H$ is

$$\hat{H}_{\text{MLE}} = \log\left(\sum_{j=1}^{S} Y_j\right) - \frac{\sum_{i=1}^{S} Y_i \log(Y_i)}{\sum_{j=1}^{S} Y_j}, \tag{15}$$

where we assume that $\sum_{j=1}^{S} Y_j > 0$.

## 3.6 Tackling Adaptive Bias when Estimating the Global Entropy of a Greybox Fuzzer

We introduced our probabilistic model for non-deterministic blackbox fuzzing which satisfies the assumption that the global species distribution $\{p_i\}_{i=1}^{S}$ is invariant during the campaign (Section 2.3). However, this assumption does *not* hold for greybox fuzzing which leverages program feedback. Generated inputs that have discovered new species (i.e., increased coverage) are added to the corpus. The availability of added seeds changes the global species distribution (but not the local distributions for each seed) and thus the global entropy for a greybox fuzzer. In other words, a greybox fuzzer becomes more efficient over time.

**Global entropy**. Clearly, our probablistic model does not directly apply to model the efficiency of a non-deterministic greybox fuzzer. However, it is still worthwhile to derive a reasonabe heuristic to estimate the *current* global entropy for a greybox fuzzer. We propose to consider greybox fuzzing as a series of mutation-based blackbox fuzzing campaigns $\langle F_1, F_2, \ldots, F_m \rangle$, each starting with a fixed-size corpus right after a new seed has been added to the corpus. Hence, for greybox fuzzers we suggest to estimate the global entropy only from incidence frequencies that have been collected since the last seed was added. We call this a *de-biased estimator*.

Figure 3.top[5] shows the maximum likelihood estimate (MLE) $\hat{H}_{\text{MLE}}$ over time for a blackbox fuzzer.[6] We can see that the estimate approaches the true value reasonably fast: from under one minute onwards, the entropy appears as a straight line. Hence, it is fair to say that a monotonic increase in entropy would demonstrate a dynamic increase in fuzzer efficiency.

Figure 3.bottom shows two entropy estimates over time for a greybox fuzzer: the unmodified, adaptively biased MLE and the de-biased MLE. We expect that the efficiency of a greybox fuzzer and thus the entropy increases over time. As more seeds are added, previously rare species become "less rare". Indeed, we observe a monotonic increase. The *maximum likelihood estimator* (MLE, biased) is computed from unmodified incidence frequencies $Y_i$ while the *de-biased estimator* (MLE, de-biased) is computed from incidence frequencies that are reset to zero whenever a new seed is added. Considering a greybox fuzzing campaign as a series of blackbox campaigns, resetting will prevent any impact of adaptive bias on the current entropy estimate.

In Figure 3.bottom we observe that the de-biased MLE approaches the final entropy estimate much faster than the unmodified MLE. It takes the unmodified, adaptively biased MLE more than 10 hours to generate the same entropy estimate that the de-biased MLE generates in an hour. The drawback of resetting the incidence frequencies to zero is compensated by a relative increase in the quality of the collected data.

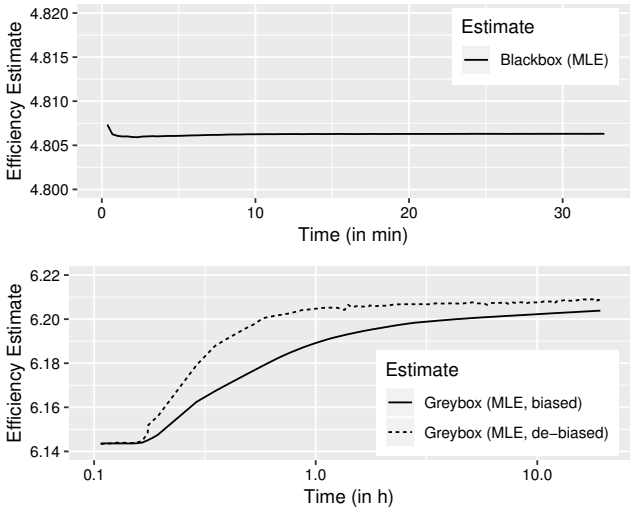



**Figure 3: Entropy estimates of a blackbox fuzzer's entropy (top) and a greybox fuzzer's entropy (bottom) over time. For each species $\mathcal{D}_i$, we count the number of generated inputs $Y_i$ belonging to $\mathcal{D}_i$ and in regular intervals compute $\hat{H}_{\text{MLE}}$ per Eqn. (15). For the de-biased estimator (bottom), we reset the incidence frequencies $Y_i = 0$ whenever a new seed is added.**

**Local entropy**. Unlike a fuzzer's global entropy, a seed's local entropy is *not* subject to adaptive bias. Given seed $t$, let $M$ be the greybox fuzzer's mutation operators, $L^t$ be the set of locations in $t$ where a mutation operator can be applied. Without loss of generality, suppose the fuzzer generates an input $t'$ only when applying the operator $m \in M$ to location $l \in L^t$.[7] Then,

$$P[X_n^t = t'] = P[A_n^t = m] \cdot P[B_n^t = l] \quad \text{and} \tag{16}$$

$$P[X_n^t = t'] = P[X_{n+1}^t = t'] \tag{17}$$

where $X_n^t$ is the $n$-th input that is generated from $t$ by fuzzing $t$, $X_{n+1}^t$ is the $(n+1)$-th input generated from $t$ by fuzzing $t$, $A_n^t$ and $B_n^t$ are the mutation operator and mutation location in $t$, respectively, and both are chosen at random when generating input $X_n^t$.

Hence, *a seed's local entropy within a greybox fuzzer is unbiased*. The local distribution $\{p_i^t\}_{i=1}^{S}$ is invariant throughout the campaign. The probability $p_i^t$ to generate an input that belongs to species $\mathcal{D}_i$ by fuzzing seed $t$ is the same every time $t$ is chosen for fuzzing.

# 4 INFORMATION-THEORETIC BOOSTING

We present an entropy-based boosting strategy for greybox fuzzing that maximizes the information each generated input reveals about the species (i.e., behaviors) in a program. Our technique EntropIC

[5] Our experimental setup for Figure 3 and Figure 4 is discussed in Section 5.2.4.
[6] LibFuzzer *without* the ability to add generated inputs to the corpus.
[7] The application of a mutation operator to a mutation location is merely an abstraction. Concretely, it means to randomly choose from a large but fixed set of possible modifications to $t$. For instance, the application of multiple concrete mutation operators can still be considered as the application of one abstract operator.

**Algorithm 1** EntropicAlgorithm.

**Input:** Program $\mathcal{P}$, Initial Seed Corpus $C$
1: **while** ¬Timeout() **do**
2: **for all** $t \in C$. assignEnergy($t$) *// power schedule*
3: $total = \sum_{t \in C} t.energy$ *// normalizing constant*
4: **for all** $t \in C$. $t.energy = \frac{t.energy}{total}$ *// normalized energy*
5: $t$ = sample $t$ from $C$ with probability $t.energy$
6: $t' = \text{Mutate}(t)$ *// fuzzing*
7: **if** $\mathcal{P}(t')$ crashes **then return** crashing seed $t'$
8: **else if** $\mathcal{P}(t')$ increases coverage **then** add $t'$ to $C$
9: **for all** covered elements $i \in \mathcal{P}$ exercised by $t'$ **do**
10: $Y_i^t = Y_i^t + 1$ *// local incidence freq.*
11: **end for**
12: **end while**
**return** Augmented Seed Corpus $C$

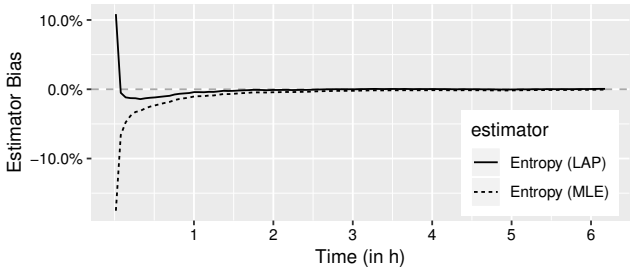



**Figure 4: Mean estimator bias over time. We monitored estimates for the same seed $t$ over 6h across 20 runs. Estimator bias is the difference between the mean estimate and the true value $H^t$ divided by the true value $H^t$, where $H^t$ is the average of both mean estimates at 6 hours into the campaign.**

is implemented into the popular greybox fuzzer LibFuzzer which is responsible for at least 12,000 bugs reported in security-critical open-source projects and over 16,000 bugs reported in a widely-used browser. After a successful independent evaluation of Entropic by the company that develops LibFuzzer, our tool Entropic is currently the subject of public code review[8] to be integrated into main-line LibFuzzer.

## 4.1 Overview of Entropic

A *greybox fuzzer* starts with a corpus of seed inputs and continuously fuzzes these by applying random mutations. Generated inputs that increase coverage are added to the corpus. The probability (i.e., frequency) with which a seed is chosen for fuzzing is called the seed's *energy*. The procedure that assigns energy to a seed is called the fuzzer's *power schedule*. For instance, LibFuzzer's standard schedule assigns more energy to seeds that were found later in the fuzzing campaign. It is this power schedule that we modify.

Algorithm 1 shows how greybox fuzzing is implemented in LibFuzzer; our changes for Entropic are shown as green boxes. In a continuous loop, the fuzzer samples a seed $t \in C$ from a distribution that is given by the seeds' normalized energy. This energy is computed using assignEnergy which implements one of our information-theoretic power schedules. The seed $t$ is then mutated using random bit flips and other mutation operators to generate an input $t'$. If the execution crashes or violates the fuzzer security policy, enacted by limits on execution time, memory usage, or sanitizers [38], $t'$ is returned as crashing input and LibFuzzer stops. If the execution increases coverage, $t'$ is added to the corpus. We call the number of inputs generated by fuzzing a seed $t \in C$ and that belong to species $\mathcal{D}_i$ as *local incidence frequency* $Y_i^t$.

## 4.2 Entropy-Based Power Schedule

Our entropy-based schedule assigns more energy to seeds that elicit more information about the program's species. In other words, the fuzzer spends more time fuzzing seeds that lead to more efficient discovery of new behaviors. The amount of information about the species in the neighborhood of a seed $t$ that we expect for each generated test input is measured using the seed's local entropy $H^t$.

The entropy-based power schedule is inspired by Active SLAM [13, 41], a problem in robot mapping: An autonomous robot is placed in an unknown terrain; the objective is to learn the map of the terrain as quickly as possible. General approaches approximate Shannon's entropy of the map under hypothetical actions [10, 13]. The next move is chosen such that the reduction in uncertainty is maximized. Similarly, our schedule chooses the next seed such that the information about the program's species is maximized.

*4.2.1 Improved Estimator.* During our experiments, we quickly noticed that the maximum likelihood estimator $\hat{H}_{\text{MLE}}$ in Equation 15 cannot be used. A new seed $t$ that has never been fuzzed will always be assigned zero energy $\hat{H}^t_{\text{MLE}} = 0$. Hence, it would never be chosen for fuzzing and forever remain with zero energy. We experimented with a screening phase to compute a rough estimate. Each new seed was first fuzzed for a fixed number of times. However, we found that too much energy was wasted gaining statistical power that could have otherwise been spent discovering more species.

To overcome this challenge we took a Bayesian approach. We know that entropy is maximal when all probabilities are equal. For a new seed $t$, we assume an uninformative prior for the probabilities $p_i^t$, i.e., $p_1^t = \ldots = p_S^t$, where $p_i^t$ is the probability that fuzzing $t$ generates an input that belongs to species $\mathcal{D}_i$. With each input that is generated by fuzzing $t$, the probabilities are incrementally updated. The posterior is a Beta distribution over $p_i^t$. The estimate $\hat{p}_i^t$ of $p_i^t$ is thus the mean of this beta distribution which is also known as the *Laplace estimator* or add-one smoothing,

$$\hat{p}_i^t = \frac{Y_i^t + 1}{S_g + \sum_{j=1}^{S} Y_j^t} \tag{18}$$

where $S_g = S(n)$ is the number of globally discovered species.

Thus, we define the improved entropy estimator $\hat{H}^t_{\text{LAP}}$ (LAP) as

$$\hat{H}^t_{\text{LAP}} = \log\left(S_g + \sum_{j=1}^{S} Y_j\right) - \frac{\sum_{i=1}^{S}(Y_i + 1)\log(Y_i + 1)}{S_g + \sum_{j=1}^{S} Y_j} \tag{19}$$

[8] Recall that the ESEC/FSE chairs advised us to blind the name of our tool, the name of the baseline tool, and the name of the company that is developing LibFuzzer. To understand our choice of baselines in the experiments, we should also mention that Entropic is *not* developed or built on top of the popular AFL greybox fuzzer.

Figure 4 illustrates the main idea. Both estimators are nearly unbiased from two hours onwards. In other words, they are within 1% from the true value, i.e., $\hat{H}^t_X \in H^t \pm 1\%$.[9] In the beginning, the MLE is negatively biased and approaches the true value from below while the LAP is positively biased and approaches the true value from above. Both estimators robustly estimate the same quantity, but only LAP assigns high energy when seed $t$ has not been fuzzed enough for an accurate estimate of the seeds information $H^t$.

*4.2.2 Measuring Information Only About Rare Species.* During our initial experiments, we also noticed that the entropy estimates for different seeds were almost the same. We found that the reason is a small number of very abundant species which have a huge impact on the entropy estimate. There are some abundant species to which each and every generated input belongs. Hence, we defined a global *abundance threshold* $\theta$ and only maintain local incidence frequencies $Y^t_i$ of *globally rare species* $\mathcal{D}_i$ that have a global incidence frequency $Y_i \leq \theta$. In Section 5, we report on the sensitivity of the boosting technique on the abundance threshold $\theta$.

# 5 EXPERIMENTAL EVALUATION

## 5.1 Research Questions

Our main hypothesis is that increasing information per generated input increases fuzzer efficiency. To evaluate our hypothesis, we implemented Entropic and ask the following research questions.

**RQ.1** *What is the empirical coverage improvement over the baseline?*

**RQ.2** *How much faster are bugs detected compared to the baseline?*

**RQ.3** *How does the choice of abundance threshold $\theta$ influence the performance of our technique?*

**RQ.4** *What is the cost of maintaining incidence frequencies?*

## 5.2 Setup and Infrastructure

*5.2.1 Implementation and Baseline.* We implemented our entropy-based power schedule into *LibFuzzer* (363 lines of change) and call our extension as *Entropic*. LibFuzzer is a state-of-the-art vulnerability discovery tool developed at [blinded] which has found almost 30k bugs in hundreds of closed- and open-source projects.

As a coverage-based greybox fuzzer (see Alg. 1), LibFuzzer seeks to maximize code coverage. Hence, our *species* is a coverage element, called feature. A *feature* is a combination of branch covered and hit count. For instance, two inputs (exercising the same branches) have a different feature set if one exercises a branch more often. Hence, *feature coverage subsumes branch coverage*. In contrast to LibFuzzer, Entropic also maintains the local and global incidence frequencies for each feature. We study the performance hit in RQ4.

LibFuzzer's original power schedule assigns more energy to seeds that have been added later in the fuzzing campaign. Entropic implements our entropy-based power schedule (Sec. 4.2) which is parameterized by the abundance threshold $\theta$. We investigate the impact of the choice of $\theta$ in RQ3.

★ *Our extension Entropic has been independently evaluated by the company that is developing LibFuzzer and was found to improve on LibFuzzer with statistical significance. Entropic was invited for integration into the main-line LibFuzzer and is currently subject to public code review. Once integrated, Entropic is poised to run on more than 25,000 machines fuzzing hundreds of security-critical software systems simultaneously and continuously.*

*5.2.2 Benchmark Subjects.* We compare Entropic with LibFuzzer on 2 benchmarks containing 250+ open-source programs used in many different domains, including browsers. We conducted almost 1,000 one-hour fuzzing campaigns and 2,000 six-hour campaigns to generate almost two CPU years worth of data.

**FTS** [37] (*12 programs, 1.2M LoC, 1 hour, 40 repetitions*) is a standard set of real-world programs to evaluate fuzzer performance. The subjects are widely-used implementations of file parsers, protocols, and data bases (e.g., libpng, openssl, and sqlite), amongst others. Each subject contains at least one known vulnerability (CVE), some of which require weeks to be found. The Fuzzer Test Suite (FTS) allows to compare the coverage achieved as well as the time to find the first crash on the provided subjects. There are originally 25 subjects, but we removed those programs where more than 15% of runs crash (leaving 12 programs with 1.2M LoC). As LibFuzzer aborts when the first crash is found, the coverage results for those subjects would be unreliable. We set a 8GB memory limit and ran LibFuzzer for *1 hour*. To gain statistical power, we repeated each experiment *40 times*. This required 40 CPU days.

**OSS-Fuzz** [31] (*263 programs, 58.3M LoC, 6 hours, 4 repetitions*) is an open-source fuzzing platform developed by Google for the large-scale continuous fuzzing of security-critical software. At the time of writing OSS-Fuzz featured 1,326 executable programs in 176 open-source projects. We selected 263 programs totaling 58.3 *million* lines of code by choosing subjects that did not crash or reach the saturation point in the first few minutes and that generated more than 1,000 executions per second. Even for the chosen subjects, we noticed that the initial seed corpora provided by the project are often for saturation: Feature discovery has effectively stopped shortly after the beginning of the campaign. It does not give much room for further discovery. Hence, we removed all initial seed corporas. We ran LibFuzzer for all programs for 6 hours and, given the large number of subjects, repeated each experiment 4 times. This required a total of 526 CPU days.

*5.2.3 Computational Resources.* All experiments for *FTS* were conducted on a machine with Intel(R) Xeon(R) Platinum 8170 2.10GHz CPUs with 104 cores and 126GB of main memory. All experiments for OSS-Fuzz were conducted on a machine with Intel(R) Xeon(R) CPU E5-2699 v4 2.20GHz with a total of 88 cores and 504GB of main memory. To ensure a fair comparison, we always ran all schedules simultaneously (same workload), each schedule was bound to one (hyperthread) core, and 20% of cores were left unused to avoid interference. In total our experiments took more than *2 CPU years* which amounts to more than 2 weeks of wall clock time.

*5.2.4 Setup for Figures 3 and 4.* Throughout the paper, we reported on the results of small experiments. In all cases when not otherwise specified, we used LibFuzzer with the original power schedule to fuzz the LibPNG project from the fuzzer-test-suite (FTS) started with a single seed input. For Figure 3, we conducted 10 runs of 20 hours using LibFuzzer both as blackbox and as greybox fuzzer. In blackbox mode, no seeds are added to the corpus. For Figure 4, we conducted 20 runs of 6 hours and monitored the single seed input

[9] In contrast to global entropy $H$, local entropy $H^t$ is *not* subject to any adaptive bias; see Section 3.6.

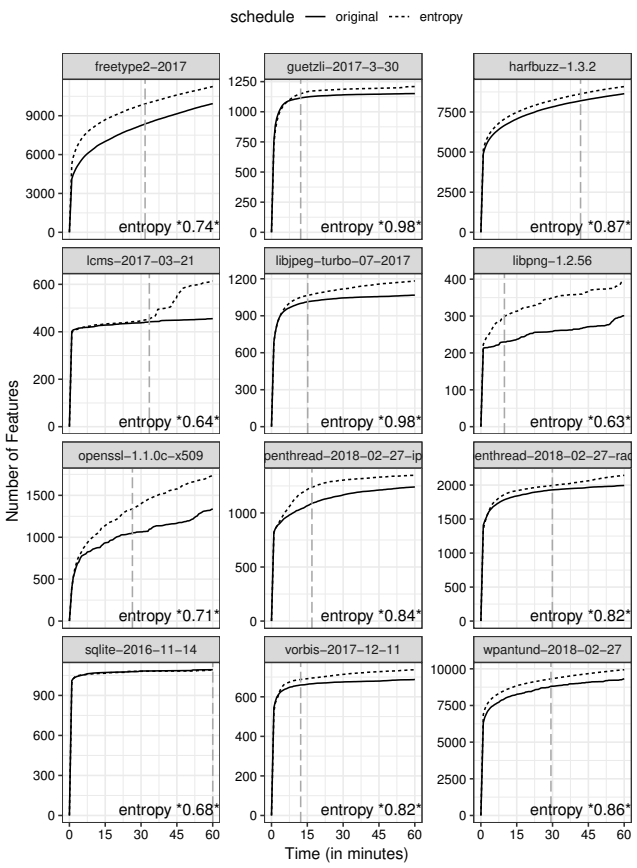
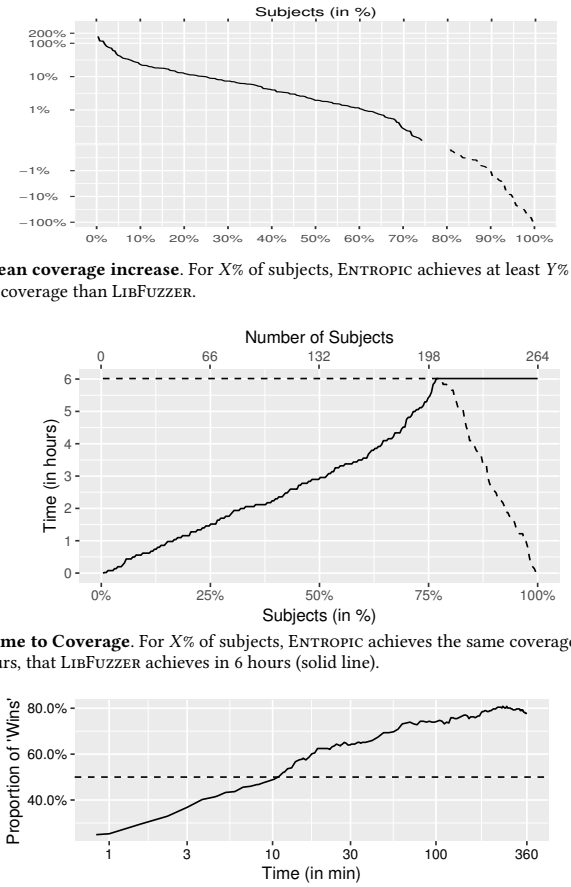



**Figure 5: Mean coverage in a 60 minute fuzzing campaign (12 subjects × 2 schedules × 40 runs × 1 hour ≈ 40 CPU days). The dashed, vertical lines show when Entropic achieves the same coverage as LibFuzzer in 1 hour. The values at the bottom right give the Vargha-Delaney effect size $\hat{A}_{12}$.**




(a) **Mean coverage increase**. For $X$% of subjects, Entropic achieves at least $Y$% more coverage than LibFuzzer.

(b) **Time to Coverage**. For $X$% of subjects, Entropic achieves the same coverage in $Y$ hours, that LibFuzzer achieves in 6 hours (solid line).

(c) **Entropic gets better at achieving coverage**. After $X$ seconds of fuzzing, Entropic achieves more coverage than LibFuzzer for $Y$% of the 263 subjects.

**Figure 6: OSS-Fuzz coverage results (263 subjects × 2 schedules × 4 runs × 6 hours ≈ 1.5 CPU years).**

that we started LibFuzzer with. We printed all four estimates in regular intervals.

## RQ1.1 Code Coverage on FTS

★ *Empirical results* confirm *our hypothesis that increasing the average information each generated input reveals about the program's species increases the rate at which new species are discovered. By choosing the seed that reveals more information, efficiency is improved.*

Figure 5 shows the mean coverage over time and the Vargha-Delaney [4] effect size $\hat{A}_{12}$. Values above 0.56, 0.63, 0.71 indicate a small, medium, and large effect size, respectively. More intuitively, the values indicate how much more likely it is for an Entropic run to cover more features than a LibFuzzer run (or less likely if under 0.5). Values in stars ($^*\hat{A}_{12}^*$) indicate statistical significance (Wilcoxon rank-sum test; $p < 0.05$). For instance, for `vorbis-2017-12-11` Entropic (with our entropy-based schedule) covers about 700 features in under 15 minutes while LibFuzzer (with the original schedule) takes one hour.

Entropic substantially outperforms LibFuzzer within the one-hour time budget for 9 of 12 subjects. For two out of three cases where the $\hat{A}_{12}$ effect size is considered medium, the mean difference in feature coverage is substantial (30% and 80% *increase* for libpng and openssl-1.1.0c, resp.). In almost all cases, Entropic is more than twice as fast (2x) as LibFuzzer. The same coverage that LibFuzzer achieves in one hour, Entropic can achieve in less than 30 minutes. All differences are statistically significant. The coverage trajectories seem to indicate that the benefit of our entropy schedule becomes even more pronounced for longer campaigns. We increase campaign length to $6h$ for our experiments with OSS-Fuzz (RQ2).

## RQ1.2 Large Scale Validation on OSS-Fuzz

★ *Results for 263 open-source C/C++ projects validate our empirical findings. Entropic generally achieves more coverage than LibFuzzer. The coverage increase is larger than 10% for a quarter of programs. Entropic is more than twice as fast as LibFuzzer for half the programs. The efficiency boost increases with the length of the campaign.*

Figure 6.(a) shows the mean coverage increase of Entropic over LibFuzzer on a logarithmic scale over *all* 263 subjects. The dashed

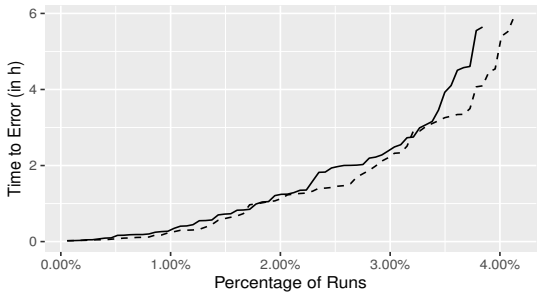



**Figure 7: OSS-Fuzz crash Time-To-Error results (≈ 1.5 CPU years). *X*% of runs crashed in *Y* hours or less. Entropic (dashed) and LibFuzzer (solid). Lower is better.**

| Max. abundance | Original | Entropy Schedule | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0x100 | | 0x1000 | | 0x10000 | |
| | $\mu$ | $\mu$ | $\hat{A}_{12}$ | $\mu$ | $\hat{A}_{12}$ | $\mu$ | $\hat{A}_{12}$ |
| freetype2-2017 | 9,932.8 | **10,818.2** | **0.67** | **11,219.1** | **0.73** | **11,018.1** | **0.71** |
| guetzli-2017-3-30 | 1,151.0 | **1,245.5** | **0.97** | **1,189.5** | **0.98** | **1,183.7** | **0.96** |
| harfbuzz-1.3.2 | 8,639.9 | **9,095.6** | **0.83** | **9,036.2** | **0.82** | **8,972.5** | **0.82** |
| lcms-2017-03-21 | 455.2 | 461.6 | 0.52 | **563.8** | **0.66** | 518.7 | 0.59 |
| libjpeg-turbo-07-2017 | 1,067.2 | **1,117.0** | **0.72** | **1,174.8** | **0.97** | **1,157.2** | **0.96** |
| libpng-1.2.56 | 300.8 | 328.1 | 0.51 | 289.4 | 0.41 | 358.9 | 0.56 |
| openssl-1.1.0c-x509 | 1,338.0 | 1,249.1 | 0.42 | **1,724.8** | **0.65** | **1,744.0** | **0.68** |
| ot-2018-02-27-ip6 | 1,240.9 | **1,085.3** | **0.32** | **1,306.3** | **0.79** | 1,243.3 | 0.45 |
| ot-2018-02-27-radio | 1,994.8 | 1,988.0 | 0.50 | **2,100.2** | **0.76** | **2,015.7** | **0.70** |
| sqlite-2016-11-14 | 1,092.7 | **1,073.3** | **0.16** | **1,084.2** | **0.69** | 1,073.0 | 0.56 |
| vorbis-2017-12-11 | 687.9 | **740.0** | **0.89** | **749.5** | **0.90** | **721.9** | **0.80** |
| wpantund-2018-02-27 | 9,323.9 | **9,927.1** | **0.86** | **9,905.0** | **0.85** | **9,936.8** | **0.86** |

**Figure 8: Sensitivity to abundance threshold which constitutes "rare" feature (12 subjects x 3 abundance thresholds x 40 runs x 1h each = 60 CPU days). Showing mean coverage ($\mu$) and Vargha-Delaney effect size ($\hat{A}_{12}$). Bold values indicate statistical significance.**

line represents the coverage increase of LibFuzzer over Entropic for the cases when LibFuzzer achieves more coverage. We can see that Entropic achieves more coverage than LibFuzzer after six hours of fuzzing for about 77% of subjects. Entropic covers at least 10% more features than LibFuzzer for about 25% of subjects. We investigated more closely the 23% of the subjects where Entropic achieves less coverage. First, for half of them, the coverage difference was marginal (less than 2%). Second, these subjects were much larger (twice the number of branches on average). As we will see in RQ4 that the performance overhead incurred by Entropic grows linearly with the number of branches.

Figure 6.(b) shows how much faster Entropic is in achieving the coverage that LibFuzzer achieves in six hours. Again, the dashed line shows the inverse when LibFuzzer achieves more coverage at the six hour mark. We can see that Entropic achieves the same coverage twice as fast for about 50% of subjects and four times as fast for 25% of subjects. More specifically, Entropic achieves the same coverage in 1.5h as LibFuzzer achieves in 6h for 66 subjects.

Figure 6.(c) shows the proportion of subjects where Entropic achieves more coverage than LibFuzzer (i.e., wins) over time. Both fuzzers break even at about 10 minutes. After 30 minutes, Entropic already wins for 64% of subjects, until at 6 hours, Entropic wins for about 77% of subjects. We interpret this result as Entropic becoming more effective at boosting LibFuzzer as saturation is being approached. In the beginning of the campaign, almost every new input leads to species discovery. Later in the fuzzing campaign, it becomes more important to choose high-entropic seeds. Moreover, estimator bias is reduced when more inputs have been generated.

## RQ2. Crash Detection

★ *In our experiments with OSS-Fuzz, Entropic found most crashes faster than LibFuzzer. Some of these crashes were found only by Entropic. These crashes are potential zero-day vulnerabilities in widely-used security-critical open-source libraries.*

Figure 7 shows the time it takes to find each crash as an aggregate statistic in ascending order over all crashes that have been discovered in any of the four runs of all subjects for both fuzzers. For instance, for 2.5% of 263 · 4 runs, Entropic finds a crash in 1.5 hours or less while LibFuzzer takes 2 hours or less. The crashes are real and potentially exploitable. All subjects are security-critical and widely used. Google maintains a responsible disclosure policy for bugs found by OSS-Fuzz. This gives maintainers some time to patch the crash before the bug report is made public. Three bugs are discovered only by Entropic.

## RQ3. Sensitivity Analysis

★ *Overall, Entropic performs best for an abundance threshold of* $\theta = 4{,}096$ *(i.e.,* `0x1000`*) for the FTS subjects.*

We analyze the sensitivity of fuzzer performance on the abundance threshold $\theta$. The abundance threshold specifies an upper limit on the global incidence frequencies $Y_i$ below which a species is considered as "rare". Only incidence frequencies of rare species are used when computing a seed's energy (see Section 4). This overcomes the challenge of overly abundant species dominating the energy values, a challenge we observed in our initial experiments. In order to study the behavior of Entropic schedules, we vary the abundance threshold $\theta$ on a logarithmic scale. To gain statistical power, all experiments of one (1) hour are repeated 40 times.

Figure 8 shows the mean coverage for LibFuzzer (original) and Entropic as well as the Vargha-Delaney effect size. The values in bold indicate statistical significance of the observed difference. A closer look at the table reveals that often the best performing threshold value appears to be subject-specific. This opens the opportunity for research on hyper-parameter tuning.

## RQ4. Performance Overhead

★ *There is a 2% median overhead across the 12 FTS subjects for maintaining incidence frequencies when compared to the entire fuzzing process. There is a 12% median overhead when compared to the time spent only in the fuzzer (and not in the subject).*

Figure 9 shows the proportion of the time that Entropic spends in the different phases of the fuzzing process. In all cases, the most time is spent executing the subject (bright gray). Entropic executes the subject between 10,000 and 100,000 times per second. The remainder of the time is spent in the fuzzer, where the darker gray bars represent functions that LibFuzzer normally performs while the black bars represent the overhead brought by Entropic.

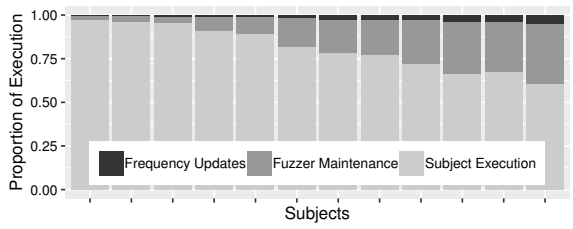



**Figure 9: Entropic instrumentation overhead.**

The maintenance of incidence frequencies takes more time away from the fuzzing process than we expected, given the substantial performance gains discussed in RQ1 and RQ2. Note that Entropic outperforms LibFuzzer *despite* this additional overhead. Entropic is a prototype. We are confident that there are plenty of opportunities to reduce this overhead to further boost Entropic's efficiency.

## 6 THREATS TO VALIDITY

Like for any empirical study, there are threats to the validity of our results. We adopted several strategies to increase *internal validity*. In order to put no fuzzer at a disadvantage, we used default configurations, provided the exact same starting condition, and executed each technique several times and under the same workload. The time when the fuzzer crashes identifies unambiguously when a bug is found. To define species in our experiments, we use the natural measure of progress for LibFuzzer and its extension Entropic. To mitigate threats to *construct validity* such as bugs in Entropic or observed performance differences that are *not* due to our discussed improvements, we extended the baseline LibFuzzer using a readily comprehensible 363 lines of code. We adopted several strategies to increase *external validity*. We repeated all experiments from which we derive empirical statements (RQ1, RQ2, RQ4) at least 40 times. To increase the generality of our results, we conducted experiments on OSS-Fuzz totaling 263 C/C++ programs and 58.3 *million* LoC.

For a sound statistical analysis, we followed the recommendations of Arcuri et al. [4] and Klees et al. [24] to the extent to which our computational resources permitted.

## 7 RELATED WORK

We begin with an overview of existing approaches to increase the efficiency of greybox fuzzing, and continue with earlier applications of information theory to software testing and debugging.

Fuzzing has recently gained substantial academic interest [5, 14, 16, 19, 21, 22, 32–36, 46]. We refer to a recent survey [28] for a comprehensive overview. Woo et al. [42] model blackbox fuzzing as a multi-armed bandit problem, where the fuzzer assigns more energy to seeds that have previously been observed to crash. PerfFuzz [25] assigns more energy to seeds that maximize execution counts to discover algorithmic complexity vulnerabilities. FairFuzz [26] and AFLFast [9] are greybox fuzzers that assign more energy to seeds that exercise low-probability paths or branches to discover more low-probability paths or branches. In contrast, Entropic computes the energy of a seed $t$ based on the discovery probabilities of species in the *neighborhood* of $t$ rather than of $t$'s species itself. Moreover, Entropic is motivated by our key insight that maximizing *information* each input reveals about a program's behaviors increases the fuzzer's efficiency.

Information theory has previously found application in *software test selection*. Given a test suite $T$ and the probability $p(t)$ that test case $t \in T$ fails, one may seek to minimize the number of test cases $t \in T$ to execute while maximizing the information that executing $t$ would reveal about the program's correctness. Yang et al. [43, 44] give several strategies to select a test case $t' \in T$ (or a size-limited set of test cases $T' \subseteq T$) such that—if we were to execute $t'$ (or $T'$)—the uncertainty about test case failure (i.e., entropy) is *minimized*. Unlike our model, the model of Yang et al. requires to specify for each input the expected output as well as the probability of failure (i.e., the observed not matching the expected output). Hence, Yang's model is practical only in the context of test selection, but not in the context of automated test generation. Similarly, Feldt et al. [17] propose an information-theoretic approach to measure the distance between test cases, based on Kolmogorov complexity, and uses it to maximize diversity of selected tests. Although their idea is complementary to ours, it is computationally too expensive to be directly applied to test generation. Finally, by considering fuzzing as a random process in a multi-dimensional space, Ankou [29] enables the detection of different *combination* in fuzzers' fitness function.

Information theory has also found application in *software fault localization*. Given a *failing* test suite $T$, suppose we want to localize the faulty statement as quickly as possible. Yoo, Harmann, and Clark [45] discuss an approach to execute test cases in the order of how much information they reveal about the fault location. Specifically, test cases—which most reduce the uncertainty that a statement *is* the fault location—will be executed first. Campos et al. [12] propose a search-based test generation technique with a fitness function that maximizes the information about the fault location. In contrast, our objective is to quantify and maximize the *efficiency* of the test generation process in learning about the program's behaviors (including whether or not it contains faults).

Bug finding efficiency and scalability are important properties of a fuzzing campaign. Böhme and Paul [8] conducted a probabilistic analysis of the efficiency of blackbox versus whitebox fuzzing, and provide concrete bounds on the time a whitebox fuzzer can take per test input in order to remain more efficient than a blackbox fuzzer. Böhme and Falk [7] empirically investigate the scalability of non-deterministic black- and greybox fuzzing and postulate an exponential cost of vulnerability discovery. Specifically, they make the following counter-intuitive observation: Finding the same bugs linearly faster requires linearly more machines. Yet, finding linearly more bugs in the same time requires exponentially more machines.

Alshahwan and Harman [2] introduced the concept of "output uniqueness" as (blackbox) coverage criterion, where one test suite is considered as more effective than another if it elicits a larger number of unique program outputs. This blackbox criterion turns out to be similarly effective as whitebox criteria (such as code coverage) in assessing test suite effectiveness. In our conceptual framework, a unique output might be considered as a species. Entropy could be used as blackbox measure of the efficiency of discovering different unqiue outputs during testing.

## 8 CONCLUSION

In this paper, we presented Entropic, the first greybox fuzzer that leverages Shannon's entropy for scheduling seeds. The key intuition behind our approach is to prefer seeds that reveal more information about the program under test. Our extensive empirical study confirms that our information-theoretic approach indeed helps in boosting fuzzing performance in terms of both code coverage and bug finding ability.

**Information theory**. We formally link entropy (as measure of information) to fuzzer efficiency, develop estimators and boosting techniques for greybox fuzzing that maximize information, and empirically investigate the resulting improvement of fuzzer efficiency. We extend the STADS statistical framework [6] to incorporate mutation-based blackbox fuzzing where a new input is generated by modifying a seed input. We hope that our information-theoretic perspective provides a general framework to think about efficiency in software testing irrespective of the chosen measure of effectiveness (i.e., independent of the coverage criterion).

**Practical Impact**. Our implementation of Entropic has been incorporated into LibFuzzer, one of the most popular industrial fuzzers. At the time of writing, Entropic was enabled for 50% of fuzzing campaigns that are run on more than 25,000 machines for finding bugs and security vulnerabilities in over 350 open-source projects, including Google Chrome. After several additional improvements, rapidly facilitated by the Fuzzbench team, Entropic now outperforms all other fuzzers available on FuzzBench [30], Google's fuzzer benchmarking platform. This result highlights the practical impact of our approach.

**Open science and reproducibility**. The practical impact of Entropic is a testament to the effectiveness of open science, open source, and open discourse. There is a growing number of authors that publicly release their tools and artifacts. Conferences are adopting artifact evaluation committees to support reproducibility [23], but, as always, more can be done to accommodate reproducibility as first-class citizens into our peer-reviewing process. We strongly believe that *openness* is a reasonable pathway to foster rapid and sound advances in the field and to enable a meaningful engagement between industry and academia.

- We make our scripts and experimental data publicly available at https://doi.org/10.6084/m9.figshare.12415622.v2
- We provide detailed instructions to reproduce our results at https://github.com/researchart/fse20
- Our results for Entropic have been independently reproduced at https://www.fuzzbench.com/reports/2020-03-04.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partly funded by the Australian Research Council (ARC) through a Discovery Early Career Researcher Award (DE190100046). This research was supported by use of the Nectar Research Cloud, a collaborative Australian research platform supported by the NCRIS-funded Australian Research Data Commons (ARDC). This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-01697, Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security).

## APPENDIX: PROOF OF THEOREM 1

In our model where each input can belong to one or more species, let $\Delta(n)$ be the expected number of new species the fuzzer discovers with the $(n+1)$-th generated test input. We prove

$$H = \log(c) + \sum_{n=1}^{\infty} \frac{\Delta(n)}{cn} \qquad \text{where } c = \sum_{j=1}^{S} p_j. \tag{20}$$

Proof. Let $c = \sum_{j=1}^{S} p_j$. We prove by Taylor expansion and Fubini's theorem that

$$H = \log(c) - \frac{\sum_{i=1}^{S} p_i \log(p_i)}{c} \qquad \text{[by Eqn. (12)]} \tag{21}$$

$$= \log(c) - \frac{\sum_{i=1}^{S} p_i \log(1-(1-p_i))}{c} \tag{22}$$

$$= \log(c) - \frac{\sum_{i=1}^{S} p_i \left[-\sum_{n=1}^{\infty} \frac{(1-p_i)^n}{n}\right]}{c} \qquad \text{[by Taylor exp.]} \tag{23}$$

$$= \log(c) + \frac{\sum_{n=1}^{\infty} \frac{1}{n} \sum_{i=1}^{S} p_i (1-p_i)^n}{c} \qquad \text{[by Fubini]} \tag{24}$$

$$= \log\left(\sum_{j=1}^{S} p_j\right) + \sum_{n=1}^{\infty} \frac{\Delta(n)}{n \sum_{j=1}^{S} p_j} \qquad \text{[by Ref. [8]]} \tag{25}$$

■

## APPENDIX: DERIVATION OF EQUATION (12)

When it is possible that $\sum_{i=1}^{S} p_i \geq 1$, we *normalize* the probabilities and compute $H = -\sum_{i=1}^{S} p'_i \log(p'_i)$ such that $p'_i = p_i / \sum_{j=1}^{S} p_j$.

$$H = -\sum_{i=1}^{S} p'_i \log(p'_i) = -\sum_{i=1}^{S} \frac{p_i}{\sum_{j=1}^{S} p_j} \log\left(\frac{p_i}{\sum_{j=1}^{S} p_j}\right) \tag{26}$$

$$= -\left[\sum_{i=1}^{S} \frac{p_i}{\sum_{j=1}^{S} p_j} \left(\log(p_i) - \log(\sum_{j=1}^{S} p_j)\right)\right] \tag{27}$$

$$= -\frac{\sum_{i=1}^{S} p_i \log(p_i)}{\sum_{j=1}^{S} p_j} + \left[\frac{\sum_{i=1}^{S} p_i}{\sum_{j=1}^{S} p_j} \log(\sum_{j=1}^{S} p_j)\right] \tag{28}$$

since $\sum_{j=1}^{S} p_j$ is constant.

$$= \log\left(\sum_{j=1}^{S} p_j\right) - \frac{\sum_{i=1}^{S} p_i \log(p_i)}{\sum_{j=1}^{S} p_j} \tag{29}$$

## REFERENCES

[1] Abhishek Aarya, Oliver Chang, Max Moroz, Martin Barbella, and Jonathan Metzman. 2019. Open sourcing ClusterFuzz. https://security.googleblog.com/2019/02/open-sourcing-clusterfuzz.html. (2019). Accessed: 2020-09-30.

[2] Nadia Alshahwan and Mark Harman. 2014. Coverage and Fault Detection of the Output-Uniqueness Test Selection Criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. 181âĂŞ192. https://doi.org/10.1145/2610384.2610413

[3] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '15)*. 33–43.

[4] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 1–10.

[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*.

[6] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Transactions on Software Engineering and Methodology* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[7] Marcel Böhme and Brandon Falk. 2020. 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 (ESEC/FSE)*. 1–12. https://doi.org/10.1145/3368089.3409729

[8] Marcel Böhme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering* 42, 4 (April 2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* (2017), 1–18.

[10] Mitch Bryson and Salah Sukkarieh. 2008. Observability Analysis and Active Control for Airborne SLAM. *IEEE Trans. Aerospace Electron. Systems* 44, 1 (January 2008), 261–280.

[11] Justin Campbell and Mike Walker. 2020. Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/. (2020). Accessed: 2020-09-30.

[12] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. 2013. Entropy-based Test Generation for Improved Fault Localization. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. 257–267.

[13] Henry Carrillo, Ian Reid, and José A. Castellanos. 2012. On the Comparison of Uncertainty Criteria for Active SLAM. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '12)*. 2080–2087.

[14] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '15)*. 725–741.

[15] Anne Chao, Y. T. Wang, and Lou Jost. 2013. Entropy and the species accumulation curve: a novel entropy estimator via discovery rates of new species. *Methods in Ecology and Evolution* 4, 11 (2013), 1091–1100.

[16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2095–2108. https://doi.org/10.1145/3243734.3243849

[17] R. Feldt, S. Poulding, D. Clark, and S. Yoo. 2016. Test Set Diameter: Quantifying the Diversity of Sets of Test Cases. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*. 223–233.

[18] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic Pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 622–631.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT '20)*. 1–12.

[20] 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)*. 166–176.

[21] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred Model-based Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '17)*. 2345–2358.

[22] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '19)*.

[23] Ben Herrmann, Stefan Winter, and Janet Siegmund. 2020. Community Expectations for Research Artifacts and Evaluation Processes. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. (ESEC/FSE 2020)*. 1–12. https://doi.org/10.1145/3368089.3409767

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2123–2138.

[25] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. 254âĂŞ265. https://doi.org/10.1145/3213846.3213874

[26] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. New York, NY, USA, 475âĂŞ485. https://doi.org/10.1145/3238147.3238176

[27] LibFuzzer. 2019. LibFuzzer: A library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html. (2019). Accessed: 2019-02-20.

[28] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2946563

[29] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference. In *Proceedings of the International Conference on Software Engineering*. 1024–1036.

[30] Jonathan Metzmann, Abhishek Arya, and Lászl'o Szekeres. 2020. FuzzBench: Fuzzer Benchmarking as a Service. https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html. (2020). Accessed: 2020-09-17.

[31] OSS-Fuzz. 2019. Continuous Fuzzing Platform. https://github.com/google/oss-fuzz/tree/master/infra. (2019). Accessed: 2019-02-20.

[32] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. New York, NY, USA, 329âĂŞ340. https://doi.org/10.1145/3293882.3330576

[33] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation (ICST 2020)*. 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[34] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019), 1–17.

[35] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the USENIX Security Symposium (SEC '14)*. 861–875.

[36] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*.

[37] Konstantin Serebryany. 2017. https://github.com/google/fuzzer-test-suite/blob/master/engine-comparison/tutorial/abTestingTutorial.md. (2017). Accessed: 2019-02-20.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12)*. 28–28.

[39] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27 (1948).

[40] Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. 2009. Saturation-based Testing of Concurrent Programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. 53–62.

[41] Sebastian Thrun. 2003. Exploring Artificial Intelligence in the New Millennium. Chapter Robotic Mapping: A Survey, 1–35.

[42] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '13)*. 511–522.

[43] Linmin Yang. 2011. *Entropy and Software Systems: Towards an Information-theoretic Foundation of Software Testing*. Ph.D. Dissertation. Advisor(s) Dang, Zhe and Fischer, Thomas R.

[44] Linmin Yang, Zhe Dang, and Thomas R. Fischer. 2011. Information gain of black-box testing. *Formal Aspects of Computing* 23, 4 (01 Jul 2011), 513–539.

[45] Shin Yoo, Mark Harman, and David Clark. 2013. Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches. *ACM Transactions on Software Engineering and Methodology* 22, 3, Article 19 (July 2013), 29 pages.

[46] Michal Zalewski. 2019. AFL: American Fuzzy Lop Fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt. (2019). Accessed: 2019-02-20.