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
KAIST, Korea
sangklic@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 $P$: What does it do? Executing the first test input, we learn how $P$ 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 $P$’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 will run on more than 25,000 machines fuzzing hundreds of security-critical software systems simultaneously and continuously.

KEYWORDS
software testing, fuzzing, efficiency, information theory, entropy

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.2 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 [26]. 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 [19] 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.

After a successful independent evaluation of Entropic by the company that develops LibFuzzer, our tool Entropic is currently the subject of public code review to be integrated into main-line LibFuzzer. Once integrated, Entropic will run on more than 25,000 machines fuzzing hundreds of security-critical software systems simultaneously and continuously. Entropic is available as open-source and will be linked here upon acceptance.

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

1https://github.com/google/clusterfuzz#trophies

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

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YT/MM.. $15.00
https://doi.org/10.1145/nnnnnn.nnnnnn

2As in, learning about the colors in an urn full of colored balls by sampling from it.
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 inputs they can generate. 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-deterministic 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://drive.google.com/drive/folders/1mnZXP-BTL_ZlkWamhhTJ2j1P3dkQG5OV

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.

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. We adopt the recently proposed STADS probabilistic model for non-deterministic blackbox fuzzing [3]: Each generated input can belong to one or more species. Beyond the probabilistic model, STADS provides bio-statistical 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 \(P\) be the program that we wish to fuzz. We call as \(P\)’s input space \(D\) the set of all inputs that \(P\) can take in. The fuzzing of \(P\) is a stochastic process

\[
F = \{X_n \mid X_n \in D\}_{n=1}^N
\]

of sampling \(N\) inputs with replacement from the program’s input space. We call \(F\) as fuzzing campaign and a tool that performs \(F\) as a non-deterministic blackbox fuzzer.

Suppose, we can subdivide the search space \(D\) into \(S\) individual subdomains \(\{D_i\}_{i=1}^S\) called species [3]. An input \(X_n \in F\) is said to discover species \(D_i\) if \(X_n \in D_i\) and there does not exist a previously sampled input \(X_m \in F\) such that \(m < n\) and \(X_m \in D_j\) (i.e., \(D_j\) 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 \(X_n \in D\) 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 \(D_i\),

\[
p_i = P[X_n \in D_i]
\]

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

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

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

\[\text{Figure 1: The expected number } S(n) \text{ of program behaviors that a test generator discovers as the number of generated test inputs } n \text{ increases. The rate } \Delta(n) = S(n+1) - S(n) \text{ at which behaviors are discovered decreases over time. Function } \Delta(n) \text{ gives the current discovery rate when the } n \text{-th test is generated. In the limit, the expected number of behaviors discovered approaches the asymptote } S.\]
limit, i.e., the asymptotic total number of species is given as

$$S = \lim_{n \to \infty} S(n).$$  \hspace{1cm} (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 [1, 27]. 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 $q_i$ be the probability that the fuzzer chooses the seed $t \in C$. For each seed $t$, let $D^t$ be the set of all inputs that can be generated by applying the available mutation operators to $t$. The mutation fuzzing of $t$ is a stochastic process

$$\mathcal{F}^t = \{X^t_n \mid X^t_n \in D^t\}_{n=1}^{N^t}$$  \hspace{1cm} (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 $X^t_n$ which is generated by mutating the seed $t \in C$ belongs to species $D_i$,

$$p_i^t = P[X^t_n \in D_i]$$  \hspace{1cm} (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 $t$. Note that global and local distributions, by the law of total expectation, are related as

$$p_i = \sum_{t \in C} q_t \cdot p_i^t.$$  \hspace{1cm} (7)

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

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

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

### 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.,

$$p_i = P[X_n \in D_i] = P[X_{n+1} \in D_i]$$  \hspace{1cm} and  \hspace{1cm} (9)

$$p_i^t = P[X^t_n \in D_i] = P[X^t_{n+1} \in D_i]$$  \hspace{1cm} (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 $X^t_n$ and $X^t_{n+1}$ 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 $d \in D$ 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 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 $d_1, d_2 \in D$, 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 $D_i = \emptyset$ 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 $\{p_i\}_{i=1}^{S}$, 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 know test cases for each species beforehand. 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 $H$ [26] 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:

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

This probability is also called the seed’s energy, weight, or perf_score (AFL).
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

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

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.

### 3.4 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
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.

**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 $l \in L^t$. Then,

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

$$P[X_n^t = t'] = P[X_{n+1}^t = t'] \quad \text{and} \quad (17)$$

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 $(p_t^i)_{i=1}^9$ is invariant throughout the campaign. The probability $p_t^i$ to generate an input that belongs to species $D_i$ 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 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 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.

---

*Our experimental setup for Figure 3 and Figure 4 is discussed in Section 5.2.4.

*LibFuzzer without the ability to add generated inputs to the corpus.*
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 [25], $t'$ 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 $D_i$ 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 [9, 28], 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 [7, 9]. 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 $H_{\text{MLE}}$ in Equation 15 cannot be used. A new seed $t$ that has never been fuzzed will always be assigned zero energy $H^t_{\text{MLE}} = 0$. 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., $p_i^t = \ldots = p_{|D|}^t$, where $p_i^t$ is the probability that fuzzing $t$ generates an input that belongs to species $D_i$. 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,

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

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., $\hat{H}_{\text{LAP}}^t \in H^t \pm 1\%$. 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 $Y_i^t$ of globally rare species $D_i$ 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 ENTRIC and ask the following research questions.

---

In contrast to global entropy $H$, local entropy $H^t$ is not subject to any adaptive bias; see Section 3.6.
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 (executing 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 investigate the resulting performance overhead 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 using two benchmarks containing 250+ open-source programs used in a vast array of different domains including browsers and online services. 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 [24] (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 [21] (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 corpora. 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.
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 fuzz-execution-suite (FES) started with a single seed input. For Figure 3, we conducted 10 runs of 20 hours using LibFuzzer both as blackbox and as greybox fuzzers. 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 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 [2] effect size \( A_{12} \). 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 (\(^* A_{12} \)) indicate statistical significance (Wilcoxon signed-rank 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 \( A_{12} \) 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 shows the mean coverage increase of Entropic over LibFuzzer on a logarithmic scale over all 263 subjects. The dashed 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
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.

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 report is made public. Three bugs are discovered only by ENTROPIC.

RQ3. Sensitivity Analysis
★ Overall, ENTROPIC performs best for an abundance threshold of \( \hat{\theta} = 4.096 \) (i.e., \( 0\times1000 \)) 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.

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 (\( \text{A}_{12} \)). Bold values indicate statistical significance.

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: ENTROPIC instrumentation overhead.

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. The measurement 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 derived 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 lines of code.

For a sound statistical analysis, we followed the recommendations of Arcuri et al. [2] and Klees et al. [18] 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, 6, 10, 12, 16, 17, 22, 23, 33]. We refer to a recent survey [20] for a comprehensive overview. Woo et al. [29] model blackbox fuzzing as a comprehensive overview. Woo et al. [29] model blackbox fuzzing as a greybox fuzzer that assigns more energy to seeds that have previously been observed to crash. AFLFast [6] is a greybox fuzzer that assigns more energy to seeds that exercise low-probability paths; the hope is to discover other low-probability paths. In contrast to AFLFast, 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 fundamentally 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 ∈ T fails, one may seek to minimize the number of test cases t ∈ T to execute while maximizing the probability executing t would reveal about the program’s correctness. Yang et al. [30, 31] give several strategies to select a test case t′ ∈ T (or a size-limited set of test cases T′ ⊆ 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. [13] propose an information-theoretic approach to measure the distance between test cases, based on Kolmogorov complexity, and uses it to maximize diversity of sets of test cases for test selection. Although their idea is complementary to ours, it is computationally too expensive to be directly applied to test generation.

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 [32] 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. [8] 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).

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. Our implementation of Entropic is scheduled to be incorporated into one of the most popular industrial fuzzers, and is going to run on more than 25,000 machines for fuzzing various applications. This result highlights the practical impact of our approach. We make our scripts and experimental data public at https://drive.google.com/drive/folders/1mnZXP-BT1_ZtkWamhhTj2jIf3oKQgSOV (anonymized). Entropic will be released as open source.

APPENDIX: PROOF OF THEOREM 1

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

\[ H = \log(c) + \sum_{n=1}^{\infty} \frac{\Delta(n)}{cn} \quad \text{where} \quad c = \sum_{j=1}^{S} p_j. \]  

(20)

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

\[ H = \log(c) - \sum_{i=1}^{S} p_i \log(p_i) c \quad \text{[by Eqn. (12)]} \]  

(21)

\[ = \log(c) - \sum_{i=1}^{S} p_i \log(1 + (1 - p_i)) c \]  

(22)

\[ = \log(c) - \sum_{i=1}^{S} p_i \left( - \frac{1}{n} \sum_{n=1}^{\infty} \frac{(1 - p_i)^n}{n} \right) \quad \text{[by Taylor expr.]} \]  

(23)

\[ = \log(c) + \sum_{n=1}^{\infty} \frac{1}{n} \sum_{j=1}^{S} p_j (1 - p_j)^n c \quad \text{[by Fubini]} \]  

(24)

\[ = \log \left( \sum_{j=1}^{S} p_j \right) + \sum_{n=1}^{\infty} \frac{\Delta(n)}{n \sum_{j=1}^{S} p_j} \quad \text{[by Ref. [4]]} \]  

(25)

\[ \square \]

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was fully or partially funded by the Australian Research Council (ARC) through a Discovery Early Career Researcher Award (DE190100046). 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).
REFERENCES


