Scheduling Black-box Mutational Fuzzing

Maverick Woo Sang Kil Cha Samantha Gottlieb David Brumley
Carnegie Mellon University
{pooh,sangkilc,sgottlie,dbrumley}@cmu.edu

ABSTRACT

Black-box mutational fuzzing is a simple yet effective technique to find bugs in software. Given a set of program-seed pairs, we ask how to schedule the fuzzings of these pairs in order to maximize the number of unique bugs found at any point in time. We develop an analytic framework using a mathematical model of black-box mutational fuzzing and use it to evaluate 26 existing and new randomized online scheduling algorithms. Our experiments show that one of our new scheduling algorithms outperforms the multi-armed bandit algorithm in the current version of the CERT Basic Fuzzing Framework (BFF) by finding 1.5× more unique bugs in the same amount of time.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Security

Keywords

Software Security; Fuzz Configuration Scheduling

1 Introduction

A General (or professor) walks into a cramped cubicle, telling the lone security analyst (or graduate student) that she has one week to find a zero-day exploit against a certain popular OS distribution, all the while making it sound as if this task is as easy as catching the next bus. Although our analyst has access to several program analysis tools for finding bugs [8, 10, 11, 21] and generating exploits [4, 9], she still faces a harsh reality: the target OS distribution contains thousands of programs, each with potentially tens or even hundreds of yet undiscovered bugs. What tools should she use for this mission? Which programs should she analyze, and in what order? How much time should she dedicate to a given program? Above all, how can she maximize her likelihood of success within the given time budget?

In this paper, we focus on the setting where our analyst has chosen to find bugs via *black-box mutational fuzzing*. At a high level, this technique takes as input a program p and a seed s that is usually assumed to be a well-formed input for p. Then, a program known as a black-box mutational *fuzzer* is used to *fuzz* the program p with the seed s, i.e., execute p on a potentially malformed input x obtained by randomly mutating s in a precise manner to be described in §2. Through repeated fuzzings, we may discover a number of inputs that crash p. These crashing inputs are then passed to downstream analyses to triage each crash into a corresponding bug, test each newly-discovered bug for exploitability, and generate exploits when possible.

Intuitively, our analyst may try to improve her chances by finding the greatest number of *unique* bugs among the programs to be analyzed within the given time budget. To model this, let us introduce the notion of a *fuzz campaign*. We assume our analyst has already obtained a list of program-seed pairs (p_i, s_i) to be fuzzed through prior manual and/or automatic analysis. A fuzz campaign takes this list as input and reports each new (previously unseen) bug when it is discovered. As a simplification, we also assume that the fuzz campaign is orchestrated in *epochs*. At the beginning of each epoch we select one program-seed pair based only on information obtained during the campaign, and we fuzz that pair for the entire epoch. This latter assumption has two subtle but important implications. First, though it does not limit us to fuzzing with only one computer, it does require that every computer in the campaign fuzz the same program-seed pair during an epoch. Second, while our definition of a *fuzz configuration* in §2 is more general than a program-seed pair, we also explain our decision to equate these two concepts in our present work. As such, what we need to select for each epoch is really a fuzz configuration, which gives rise to our naming of the *Fuzz Configuration Scheduling* (FCS) problem.

To find the greatest number of unique bugs given the above problem setting, our analyst must allocate her time wisely. Since initially she has no information on which configuration will yield more new bugs, she should explore the configurations and reduce her risk by fuzzing each configuration for an adequate amount of time. As she starts to identify some configurations that she believes may yield more new bugs in the future, she should also exploit this information by increasing the time allocated to fuzz these configurations. Of course, any increase in exploitation reduces exploration,

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.
CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516736.

which may cause our analyst to under-explore and miss configurations that are capable of yielding more new bugs. This is the classic "exploration vs. exploitation" trade-off, which signifies that we are dealing with a Multi-Armed Bandit (MAB) problem [5].

Unfortunately, merely recognizing the MAB nature of our problem is not sufficient to give us an easy solution. As we explain in §3, even though there are many existing MAB algorithms and some even come with excellent theoretical guarantees, we are not aware of any MAB algorithm that is designed to cater to the specifics of finding *unique* bugs using black-box mutational fuzzing. For example, suppose we have just found a crash by fuzzing a program-seed pair and the crash gets triaged to a new bug. Should an MAB algorithm consider this as a high reward, thus steering itself to fuzz this pair more frequently in the future? Exactly what does this information tell us about the probability of finding another new bug from this pair in future fuzzes? What if the bug was instead a duplicate, i.e., one that has already been discovered in a previous fuzz run? Does that mean we should assign a zero reward since this bug does not contribute to the number of unique bugs found?

As a first step to answer these questions and design more suitable MAB algorithms for our problem, we discover that the memoryless property of black-box mutational fuzzing allows us to formally model the repeated fuzzings of a configuration as a bug arrival process. Our insight is that this process is a *weighted* variant of the Coupon Collector's Problem (CCP) where each coupon type has its own fixed but initially *unknown* arrival probability. We explain in §4.1 how to view each fuzz run as the arrival of a coupon and each unique bug as a coupon type. Using this analogy, it is easy to understand the need to use the weighted variant of the CCP (WCCP) and the challenge in estimating the arrival probabilities.

The WCCP connection has proven to be more powerful than simply affording us clean and formal notation—not only does it explain why our problem is impossible to optimize in its most general setting due to the No Free Lunch Theorem, but it also pinpoints how we can circumvent this impossibility result if we are willing to make certain assumptions about the arrival probabilities in the WCCP (§4.2). Of course, we also understand that our analyst may not be comfortable in making any such assumptions. This is why we have also investigated how she can use the statistical concept of *confidence intervals* to estimate an upperbound on the sum of the arrival probabilities of the unique bugs that remain to be discovered in a fuzz configuration. We argue in §4.3 why this upperbound offers a pragmatic way to cope with the above impossibility result.

Having developed these analytical tools, we explore the design space of online algorithms for our problem in §4.4. We investigate two epoch types, five belief functions that estimate future bug arrival using past observations, two MAB algorithms that use such belief functions and three that do not. By combining these dimensions, we obtain 26 online algorithms for our problem. While some of these algorithms have appeared in prior work, the majority of them are new. In addition, we also present offline algorithms for our problem in §4.5. In the case where the sets of unique bugs from each configuration are disjoint, we obtain an efficient algorithm that computes the offline *optimal*, i.e., the maximum number of unique bugs that can be found by any algorithm in any given time budget. In the other case where these sets may overlap, we also propose an efficient heuristic that lowerbounds the offline optimal.

To evaluate our online algorithms, we built FuzzSim, a novel replay-based fuzz simulation system that we present in §5. FuzzSim is capable of simulating any online algorithm using pre-recorded fuzzing data. We used it to implement numerous algorithms, including the 26 presented in this paper. We also collected two extensive sets of fuzzing data based on the most recent stable release of the Debian Linux distribution up to the time of our data collection. To this end, we first assembled 100 program-seed pairs comprising FFMpeg with 100 different seeds and another 100 pairs comprising 100 different Linux file conversion utilities, each with an input seed that has been manually verified to be valid. Then, we fuzzed *each* of these 200 program-seed pairs for 10 days, which amounts to $48,000$ CPU hours of fuzzing in total. The performance of our online algorithms on these two datasets is presented in §6. In addition, we are also releasing FuzzSim as well as our datasets in support of open science. Besides replicating our experiments, this will also enable fellow researchers to evaluate other algorithms. For details, please visit `http://security.ece.cmu.edu/fuzzsim/`.

2 Problem Setting and Notation

Let us start by setting out the definitions and assumptions needed to mathematically model black-box mutational fuzzing. Our model is motivated by and consistent with real-world fuzzers such as **zzuf** [16]. We then present our problem statement and discuss several algorithmic considerations. For the rest of this paper, the terms "fuzzer" and "fuzzing" refer to the black-box mutational variant unless otherwise stated.

2.1 Black-box Mutational Fuzzing

Black-box mutational fuzzing is a dynamic bug-finding technique. It endeavors to find bugs in a given program p by running it on a sequence of inputs generated by randomly mutating a given seed input s. The program that generates these inputs and executes p on them is known as a black-box mutational fuzzer. In principle, there is no restriction on s other than it being a string with a finite length; however, in practice, s is often chosen to be a well-formed input for p in the interest of finding bugs in p more effectively. With each execution, p either crashes or properly terminates. Multiple crashes, however, may be due to the same underlying bug. Thus there needs to be a bug-triage process to map each crash into its corresponding bug. Understanding the effects of these multiplicities is key to analyzing black-box mutational fuzzing.

To formally define black-box mutational fuzzing, we need a notion of "random mutations" for bit strings. In what follows, let $|s|$ denote the bit-length of s.

Definition 2.1. *A* random mutation of a bit *b is the exclusive-or*[1] *of the bit b and a uniformly-chosen bit. With respect to a given* mutation ratio *$r \in [0,1]$, a* random mutation of a string *s is generated by first selecting $d = r \cdot |s|$ different bit-positions uniformly at random among the $\binom{|s|}{d}$ possible combinations and then randomly mutating those d bits in s.*

[1]Mutations in the form of unconditionally setting or unsetting the bit are possible, but they are both harder to analyze mathematically and less frequently used in practice. To justify the latter, we note that **zzuf** defaults to exclusive-or.

DEFINITION 2.2. *A* black-box mutational fuzzer *is a randomized algorithm that takes as input a* fuzz configuration*, which comprises (i) a program p, (ii) a seed input s, and (iii) a mutation ratio $r \in [0, 1]$. In a* fuzz run*, the fuzzer generates an input x by randomly mutating s with the mutation ratio r and then runs p on x. The* outcome *of this fuzz run is a crash or a proper termination of p.*

At this point, it is convenient to set up one additional notation to complement Definition 2.1. Let $H_d(s)$ denote the set of all strings obtained by randomly-mutating s with the mutation ratio $r = d/|s|$. This notation highlights the equivalence between the set of all obtainable inputs and the set of all $|s|$-bit strings within a Hamming distance of d from s. In this notation, the input string x in Definition 2.1 is simply a string chosen uniformly at random from $H_d(s)$. As we explain below, in this paper we use a globally-fixed mutation ratio and therefore d is fixed once s is given. This is why we simply write $H(s)$ instead of $H_d(s)$.

We now state and justify several assumptions of our mathematical model, all of which are satisfied by typical fuzzers in practice.

ASSUMPTION 1. *Each seed input has* finite *length.*

This assumption is always satisfied when fuzzing file inputs. In practice, some fuzzers can also perform stream fuzzing, which randomly mutates each bit in an input stream with a user-configurable probability. Notice that while the expected number of randomly-mutated bits is fixed, the actual number is *not*. We do not model stream fuzzing.

ASSUMPTION 2. *An execution of the program exhibits* exactly *one of the following two possible outcomes—it either crashes or properly terminates.*

In essence, this assumption means we focus exclusively on finding bugs that lead to crashes. Finding logical bugs that do not lead to crashes would typically require a correctness specification of the program under test. At present, such specifications are rare in practice and therefore this assumption does not impose a severe restriction.

ASSUMPTION 3. *The outcome of an execution of the program depends* solely *on the input x generated by the fuzzer.*

This assumption ensures we are not finding bugs caused by input channels not under the fuzzer's control. Since the generated input alone determines whether the program crashes or terminates properly, all bugs found during fuzzing are deterministically reproducible. In practice, inputs that do not cause a crash in downstream analyses are discarded.

Mutation Ratio. We include the mutation ratio as a third parameter in our definition of fuzz configurations given in Definition 2.2. Our choice reflects the importance of this parameter in practice since different seeds may need to be fuzzed at different mutation ratios to be effective in finding bugs. However, in order to evaluate a large number of scheduling algorithms, our work is based on a replay simulation as detailed in §5. Gathering the ground-truth fuzzing data for such simulations is resource-intensive, prohibitively so if we examine multiple mutation ratios. As such, our current project globally fixes the mutation ratio at 0.0004, the default value used in `zzuf`. Accordingly, we suppress the third parameter of a fuzz configuration in this paper, effectively equating program-seed pairs with fuzz configurations. For further discussion related to the mutation ratio, see §6.6.

2.2 Problem Statement

Given a list of K fuzz configurations $\{(p_1, s_1), \cdots, (p_K, s_K)\}$ and a time budget T, the *Fuzz Configuration Scheduling* problem seeks to maximize the number of *unique* bugs discovered in a fuzz campaign that runs for a duration of length T. A *fuzz campaign* is divided into *epochs*, starting with epoch 1. We consider two *epoch types*: fixed-run and fixed-time. In a fixed-run campaign, each epoch corresponds to a constant number of fuzz runs; since the time required for individual fuzz runs may vary, fixed-run epochs may take variable amounts of time. On the other hand, in a fixed-time campaign, each epoch corresponds to a constant amount of time. Thus, the number of fuzz runs completed may vary across fixed-time epochs.

An online algorithm $\mathcal{A}$ for the Fuzz Configuration Scheduling problem operates before each epoch starts. When the campaign starts, $\mathcal{A}$ receives the number K. Suppose the campaign has completed ℓ epochs so far. Before epoch $(\ell+1)$ begins, $\mathcal{A}$ should select a number $i \in [1, K]$ based on the information it has received from the campaign. Then the entire epoch $(\ell + 1)$ is devoted to fuzzing (p_i, s_i). When the epoch ends, $\mathcal{A}$ receives a sequence of IDs representing the outcomes of the fuzz runs completed during the epoch. If an outcome is a crash, then the returned ID is the bug ID computed by the bug triage process, which we assume is non-zero. Otherwise, the outcome is a proper termination, and the returned ID is 0. Also, any ID that has never been encountered by the campaign prior to epoch $(\ell + 1)$ is marked as new. Notice that a new ID can signify either the first proper termination in the campaign or a new bug discovered during epoch $(\ell + 1)$. Besides the list of IDs, $\mathcal{A}$ also receives statistical information about the epoch. In a fixed-run campaign, it receives the time spent in the epoch; in a fixed-time campaign, it receives the number of fuzz runs that ended inside the epoch.

Algorithmic Considerations. We now turn to a few technical issues that we withheld from the above problem statement. First, we allow $\mathcal{A}$ to be either deterministic or randomized. This admits the use of various existing MAB algorithms, many of which are indeed randomized.

Second, notice that $\mathcal{A}$ receives only the number of configurations K but not the actual configurations. This formulation is to prevent $\mathcal{A}$ from analyzing the content of any p_i's or s_i's. Similarly, we prevent $\mathcal{A}$ from analyzing bugs by sending it only the bug IDs but not any concrete representation.

Third, $\mathcal{A}$ also does not receive the time budget T. This forces $\mathcal{A}$ to make its decisions without knowing how much time is left. Therefore, $\mathcal{A}$ has to attempt to discover new bugs as early as possible. While this rules out any algorithm that adjusts its degree of exploration based on the time left, we argue that this not a severe restriction from the perspective of algorithm design. For example, one of the algorithms we use is the EXP3.S.1 algorithm [2]. It copes with the unknown time horizon by partitioning time into exponentially longer periods and picking new parameters at the beginning of each period, which has a known length.

Fourth, our analysis assumes that the K fuzz configurations are chosen such that they yield disjoint sets of bugs. This assumption is needed so that we can consider the bug arrival process of fuzzing each configuration independently. While this assumption may be valid when every configuration involves a different program, as in one of our two datasets,

satisfying it when one program can appear in multiple configurations is non-trivial. In practice, it is achieved by selecting seeds that exercise different code regions. For example, in our other data set, we use seeds of various file formats to fuzz the different file parsers in a media player.

Finally, at present we do not account for the time spent in bug triage, though this process requires considerable time. In practice, triaging a crash takes approximately the same amount of time as the fuzz run that initially found the crash. Therefore, bug triage can potentially account for over half of the time spent in an epoch if crashes are extremely frequent. We plan to incorporate this consideration into our project at a future time.

3 Multi-Armed Bandits

As explained in §1, the Fuzz Configuration Scheduling problem is an instance of the classic Multi-Armed Bandit (MAB) problem. This has already been observed by previous researchers. For example, the CERT Basic Fuzzing Framework (BFF) [14], which supports fuzzing a single program with a collection of seeds and a set of mutation ratios, uses an MAB algorithm to select among the seed-ratio pairs during a fuzz campaign. However, we must stress that recognizing the MAB nature of our problem is merely a first step. In particular, we should not expect an MAB algorithm with provably "good" performance, such as one from the UCB [3] or the EXP3 [2] families, to yield good results in our problem setting. There are at least two reasons for this.

First, although many of these algorithms are proven to have optimal *regret* in various forms, the most common form of regret does not actually give good guarantees in our problem setting. In particular, this form of regret measures the difference between the expected reward of an algorithm and the reward obtained by consistently fuzzing the single best configuration that yields the greatest number of unique bugs. However, we are interested in evaluating performance relative to the total number of unique bugs from *all* K configurations, which may be much greater than the number from one *fixed* configuration. Thus, the low-regret guarantee of many MAB algorithms is in fact measuring against a target that is likely to be much lower than what we desire. In other words, given our problem setting, these algorithms are not guaranteed to be competitive at all!

Second, while there exist algorithms with provably low regret in a form suited to our problem setting, the actual regret bounds of these algorithms often do not give meaningful values in practice. For example, one of the MAB algorithms we use is the EXP3.S.1 algorithm [2], proven to have an expected *worst-case regret* of $\frac{S+2e}{\sqrt{2}-1}\sqrt{2K\ell\ln(K\ell)}$, where S is a certain hardness measure of the problem instance as defined in [2, §8] and ℓ is the number of epochs in our problem setting. Even assuming the easiest case where S equals to 1 and picking K to be a modest value of 10, the value of this bound when $\ell = 4$ is already slightly above 266. However, as we see in §6, the number of bugs we found in our two datasets are 200 and 223 respectively. What this means is that this regret bound is very likely to dwarf the number of bugs that can be found in real-world software after a very small number of epochs. In other words, even though we have the right kind of guarantee from EXP3.S.1, the guarantee quickly becomes meaningless in practical terms.

Having said the above, we remark that this simply means such optimal regret guarantees may not be useful in *ensuring* good results. As we will see in §6, EXP3.S.1 can still obtain reasonably good results in the right setting.

4 Algorithms for the FCS Problem

Our goal in this section is to investigate how to design online algorithms for the Fuzz Configuration Scheduling problem. We largely focus on developing the design space (§4.4), making heavy use of the mathematical foundation we lay out in §4.1 and §4.3. Additionally, we present two impossibility results in §4.2, one of which requires a precise condition that greatly informs our algorithm design effort. We also present two offline algorithms for our problem. While such algorithms may not be applicable in practice, a unique aspect of our project allows us to use them as benchmarks which we measure our online algorithms against. We explain this along with the offline algorithms in §4.5.

4.1 Fuzzing as a Weighted CCP

Let us start by mathematically modeling the process of repeatedly fuzzing a configuration. As we explained in §2, the output of this process is a stream of crashes intermixed with proper terminations, which is then transformed into a stream of IDs by a bug triage process. Since we want to maximize the number of unique bugs found, we are naturally interested in when a new bug arrives in this process. This insight quickly leads us to the Coupon Collector's Problem (CCP), a classical arrival process in probability theory.

The CCP concerns a consumer who obtains one coupon with each purchase of a box of breakfast cereal. Suppose there are M different coupon types in circulation. One basic question about the CCP is: what is the expected number of purchases required before the consumer amasses k ($\leq M$) *unique* coupons? In its most elementary formulation, each coupon is chosen uniformly at random among the M coupon types. In this setting, many questions related to the CCP—including the one above—are relatively easy to answer.

Viewing Fuzzing as WCCP with Unknown Weights. Unfortunately, our problem setting actually demands a weighted variant of the CCP which we dub the WCCP. Intuitively, this is because the probabilities of the different outcomes from a fuzz run are not necessarily (and unlikely to be) uniform. This observation has also been made by Arcuri et al. [1].

Let $(M-1)$ be the actual number of unique bugs discoverable by fuzzing a certain configuration. Then including proper termination of a fuzz run as an outcome gives us exactly M distinct outcome types. We thus relate the process of repeatedly fuzzing a configuration to the WCCP by viewing fuzz run outcomes as coupons and their associated IDs as coupon types.

However, unlike usual formulations of the WCCP where the distribution of outcomes across type is given, in our problem setting this distribution is unknown *a priori*. In particular, there is no way to know the true value of M for a configuration without exhaustively fuzzing all possible mutations. As such, we utilize statistical estimations of these distributions rather than the ground-truth in our algorithm design. An important question to consider is whether accurate estimations are feasible.

We now explain why we prefer the sets of bugs from different configurations used in a campaign to be disjoint. Observe that our model of a campaign is a combination of multiple independent WCCP processes. If a bug that is new to one process has already been discovered in another, then this

bug cannot contribute to the total number of unique bugs. This means that overlap in the sets of bugs diminishes the fidelity of our model, so that any algorithm relying on its predictions may suffer in performance.

WCCP Notation. Before we go on, let us set up some additional notation related to the WCCP. In an effort to avoid excessive indices, our notation implicitly assumes a fixed configuration (p_i, s_i) that is made apparent by context. For example, M, the number of possible outcomes when fuzzing a given configuration as defined above, follows this convention.

(i) Consider the fixed sequence σ of outcomes we obtain in the course of fuzzing (p_i, s_i) during a campaign. We label an outcome as type k if it belongs to the kth distinct type of outcome in σ. Let $\mathcal{P}_k$ denote the probability of encountering a type-k outcome in σ, i.e.,

$$\mathcal{P}_k = \frac{|\{x \in H(s_i) : x \text{ triggers an outcome of type } k\}|}{|H(s_i)|}. \quad (1)$$

(ii) Although both the number and frequency of outcome types obtainable by fuzzing (p_i, s_i) are unknown a priori, during a campaign we do have empirical observations for these quantities up to any point in σ. Let $\hat{M}(\ell)$ be the number of distinct outcomes observed from epoch 1 through epoch ℓ. Let $n_k(\ell)$ be the number of inputs triggering outcomes of type k observed throughout these ℓ epochs. Notice that over the course of a campaign, the sequence σ is segmented into subsequences, each of which corresponds to an epoch in which (p_i, s_i) is chosen. Thus, the values of $\hat{M}(\cdot)$ and $n_k(\cdot)$ will not change if (p_i, s_i) is not chosen for the current epoch. With this notation, we can also express the empirical probability of detecting a type-k outcome following epoch ℓ as

$$\hat{\mathcal{P}_k}(\ell) = \frac{n_k(\ell)}{\sum_{k'=1}^{\hat{M}(\ell)} n_{k'}(\ell)}.$$

4.2 Impossibility Results

No Free Lunch. The absence of any assumption on the distribution of outcome types in the WCCP quickly leads us to our first impossibility result. In particular, *no* algorithm can consistently outperform other algorithms for the FCS problem. This follows from a well-known impossibility result in optimization theory, namely the "No Free Lunch" theorem by Wolpert and Macready [22]. Quoting Wolpert and Macready, their theorem implies that "any two optimization algorithms are equivalent when their performance is averaged across all possible problems." In our problem setting, maximizing the number of bugs found in epoch $(\ell + 1)$ amounts to, for each configuration, estimating its $\mathcal{P}_{\hat{M}(\ell)+1}$ in equation (1) using only past observations from that configuration. Intuitively, by averaging across all possible outcome type distributions, any estimation will be incorrect *sufficiently often* and thus lead to suboptimal behavior that cancels any advantage of one algorithm over another.

While we may consider this result to be easy to obtain once we have properly set up our problem using §2 and §4.1, we consider it to be an important intellectual contribution for the pragmatic practitioners who remain confident that they *can* design algorithms that outperform others. In particular, the statement of the No Free Lunch theorem itself reveals precisely how we can circumvent its conclusion—our estimation procedure must *assume* the outcome type distributions have particular characteristics. Our motto is thus "there is no free lunch—please bring your own prior!"

Tight K-Competitiveness. Our second impossibility result shows that there are problem instances in which the time spent by any deterministic online algorithm to find a given number of unique bugs in a fixed-time campaign is at least K times larger than the time spent by an optimal offline algorithm. Using the terminology of competitive analysis, this shows that the competitive ratio of any deterministic online algorithm for this problem is at least K.

To show this, we fix a deterministic algorithm A and construct a contrived problem instance in which there is only *one* bug among all the configurations in a campaign. Since A is deterministic, there exists a unique pair (p_i^*, s_i^*) that gets chosen *last*. In other words, the other $(K-1)$ pairs have all been fuzzed for at least one epoch when (p_i^*, s_i^*) is fuzzed for the first time. If the lone bug is only triggered by fuzzing (p_i^*, s_i^*), then A will have to fuzz for at least K epochs to find it.

For an optimal offline algorithm, handling this contrived scenario is trivial. Since it is offline, it has full knowledge of the outcome distributions, enabling it to hone in on the special pair (p_i^*, s_i^*) and find the bug in the first epoch. This establishes that K is a lowerbound for the competitive ratio of any deterministic algorithm.

Finally, we observe that Round-Robin is a deterministic online algorithm that achieves the competitive ratio K in *every* problem instance. It follows immediately that K is tight.

4.3 Upperbounding the Probability of Seeing a New Outcome During Fuzzing

Having seen such strong impossibility results, let us consider what a pragmatist might do before bringing in any prior on the outcome type distribution. In other words, if we do not want to make any assumptions on this distribution, is there a justifiable approach to designing online algorithms for the FCS problem?

We argue that the answer is yes. Consider two program-seed pairs (p_1, s_1) and (p_2, s_2) for which we have *upperbounds* on the probability of finding a new outcome if we fuzz them once more. Assume that the upperbound for (p_1, s_1) is the higher of the two.

We stress that what we know are merely upperbounds—it is still possible that the true probability of yielding a new outcome from fuzzing (p_1, s_1) is lower than that of (p_2, s_2). Nonetheless, with no information beyond the ordering of these upperbounds, fuzzing (p_1, s_1) first is arguably the more prudent choice. This is because to do otherwise would indicate a belief that the actual probability of finding a new outcome by fuzzing (p_1, s_1) in the next fuzz run is lower than the upperbound for (p_2, s_2).

Accepting this argument, how might we obtain such upperbounds? We introduce the Rule of Three for this purpose.

Rule of Three. Consider an experiment of independent Bernoulli trials with identical success and failure probabilities p and $q = (1 - p)$. Suppose we have carried out $N \geq 1$ trials so far and every trial has been a success. What can we say about q other than the fact that it must be (i) at least 0 to be a valid probability and (ii) strictly less than 1 since p is evidently positive? In particular, can we place a lower upperbound on q?

Unfortunately, the answer is a resounding *no*: even with q arbitrarily close to 1, we still have ($p^N > 0$). This means our observation really *could* have happened even if it is extremely unlikely.

Fortunately, if we are willing to rule out the possibility of encountering extremely unlikely events, then we may compute a lower upperbound for q by means of a confidence interval. For example, a 95% confidence interval on q outputs an interval that includes the true value of q of the underlying experiment with 95% certainty. In other words, if the outputted interval does not contain the true value of q for the experiment, then the observed event must have a likelihood of at most 5%.

For the above situation, there is particularly neat technique to compute a 95% confidence interval on q. Known as the "Rule of Three", this method simply outputs 0 and $3/N$ for the lowerbound and upperbound, respectively. The lowerbound is trivial, and the upperbound has been shown to be a good approximation for $N > 30$. See [15] for more information on this technique, including the relationship between 95% confidence and the constant 3.

How We Use Rule of Three. In order to apply the Rule of Three, we must adapt our fuzzing experiments with any $M > 1$ possible outcome types to fit the mold of Bernoulli trials.

We make use of a small trick. Suppose we have just finished epoch ℓ and consider a particular configuration (p_i, s_i). Using our notation, we have observed $\hat{M}(\ell)$ different outcomes so far and for $1 \leq k \leq \hat{M}(\ell)$, we have observed $n_k(\ell)$ counts of outcome of type k. Let $N(\ell) = \sum_{k=1}^{\hat{M}(\ell)} n_k(\ell)$ denote the total number of fuzz runs for this pair through epoch ℓ. The trick is to define a "success" to be finding an outcome of type 1 through type $\hat{M}(\ell)$. Then, in hindsight, it *is* the case that our experiment has only yielded success so far.

With this observation, we may now apply the Rule of Three to conclude that $[0, 3/N(\ell)]$ is a 95% confidence interval on the "failure" probability—the probability that fuzzing this configuration will result in an outcome type that we have *not* seen before, i.e., a new outcome. Then, as desired, we have an easy-to-compute upperbound on the probability of finding a new outcome for each configuration.

We introduce one more piece of notation before proceeding: define the Remaining Probability Mass (RPM) of (p_i, s_i) at the end of epoch ℓ, denoted $\text{RPM}(\ell)$, to be the probability of finding a new outcome if we fuzz (p_i, s_i) once more. Note that the pair in $\text{RPM}(\ell)$ is implicit, and that this value is upperbounded by $3/N(\ell)$ if we accept a 95% confidence interval.

4.4 Design Space

In this section, we explore the design space that a pragmatist may attempt when designing online algorithms for the Fuzz Configuration Scheduling problem. A depiction of the design space, along with our experimental results, is given in Table 2 in §6. Our focus here is to explain our motivation for choosing the three dimensions we explore and the particular choices we include in each dimension. By combining these dimensions, we obtain 26 online algorithms for our problem. We implemented these algorithms inside a simulator, FuzzSim, the detail of which is presented in §5.

Epoch Type. We consider two possible definitions of an epoch in a fuzz campaign. The first is the more traditional choice and is used in the current version of CERT BFF v2.6 [14]; the second is our proposal.

Fixed-Run. Each epoch executes a constant number of fuzz runs. In FuzzSim, a fixed-run epoch consists of 200 runs. Note that any differential in fuzzing speed across configurations translates into variation in the time spent in fixed-run epochs.

Fixed-Time. Each epoch is allocated a fixed amount of time. In FuzzSim, a fixed-time epoch lasts for 10 seconds. Our motivation to investigate this epoch type is to see how heavily epoch time variation affects the results obtained by systems with fixed-run epochs.

Belief Metrics. Two of the MAB algorithms we present below make use of a belief metric that is associated with each configuration and is updated after each epoch. Intuitively, the metrics are designed such that fuzzing a configuration with a higher metric *should* yield more bugs in expectation. The first two beliefs below use the concept of RPM to achieve this without invoking any prior; the remaining three embrace a "bug prior". For now, suppose epoch ℓ has just finished and we are in the process of updating the belief for the configuration (p_i, s_i).

RPM. We use the upperbound in the 95% confidence interval given by the Rule of Three to approximate $\text{RPM}(\ell)$. The belief is simply $3/N(\ell)$.

Expected Waiting Time Until Next New Outcome (EWT). Since RPM does not take into account of the speed of each fuzz run, we also investigate a speed-normalized variant of RPM. Let $\textsc{Time}(\ell)$ be the cumulative time spent fuzzing this configuration from epoch 1 to epoch ℓ. Let $\textsc{avgTime}(\ell)$ be the average time of a fuzz run, i.e., $\frac{\textsc{Time}(\ell)}{N(\ell)}$. Let W be a random variable denoting the waiting time until the next new outcome. Recall that $\text{RPM}(\ell)$ is the probability of finding a new outcome in the next fuzz run and assume it is independent of $\textsc{avgTime}(\ell)$. To compute $E[W]$, observe that either we find a new outcome in the next fuzz run, or we do not and we have to wait again. Therefore,

$$\begin{aligned} E[W] &= \text{RPM}(\ell) \times \textsc{avgTime}(\ell) \\ &\quad + (1 - \text{RPM}(\ell)) \times (\textsc{avgTime}(\ell) + E[W]). \end{aligned}$$

(Notice that RPM does not change even in the second case; what changes is our upperbound on RPM.) Solving for $E[W]$ yields $\frac{\textsc{avgTime}(\ell)}{\text{RPM}(\ell)}$, and we substitute in the upperbound of the 95% confidence interval for $\text{RPM}(\ell)$ to obtain $E[W] \geq \frac{\textsc{avgTime}(\ell)}{3/N(\ell)} = \frac{\textsc{Time}(\ell)}{3}$. Since a larger waiting time is less desirable, the belief used is its reciprocal, $3/\textsc{Time}(\ell)$.

Rich Gets Richer (RGR). This metric is grounded in what we call the "bug prior", which captures our empirical observation that code tends to be either robust or bug-ridden. Programs written by programmers of different skill levels or past testing of a program might explain this real-world phenomenon. Accordingly, demonstrated bugginess of a program serves as a strong indicator that more bugs will be found in that program and thus the belief is $\hat{M}(\ell)$.

Density. This is a runs-normalized variant of RGR and is also the belief used in CERT BFF v2.6 [14]. The belief function is $\hat{M}(\ell)/N(\ell)$. Observe that this is the belief function of RPM scaled by $\hat{M}(\ell)/3$. In other words, Density can be seen as RPM adapted with the bug prior.

Rate. This is a time-normalized variant of RGR. The belief function is $\hat{M}(\ell)/\,\text{TIME}(\ell)$. Similar to Density, Rate can be seen as EWT adapted with the bug prior.

Bandit Algorithms. Since the FCS problem is an instance of an MAB problem, naturally we explore a number of MAB algorithms.

Round-Robin. This simply loops through the configurations in a fixed order, dedicating one epoch to each configuration. Note that Round-Robin is a non-adaptive, deterministic algorithm.

Uniform-Random. This algorithm selects uniformly at random from the set of configurations for each epoch. Like Round-Robin, this algorithm is non-adaptive; however, it is randomized.

Weighted-Random. Configurations are selected at random in this algorithm, with the probability associated with each configuration is linked to the belief metric in use. The weight of a well-performing configuration is adjusted upward via the belief metric, thereby increasingly the likelihood of selecting that configuration in future epochs. This mechanism functions in reverse for configurations yielding few or no bugs.

ϵ-Greedy. The ϵ-Greedy algorithm takes an intuitive approach to the exploration vs. exploitation trade-off inherent to MAB problems. With probability ϵ, the algorithm selects a configuration uniformly at random for exploration.With probability $(1-\epsilon)$, it chooses the configuration with the highest current belief, allowing it to exploit its current knowledge for gains. The constant ϵ serves as a parameter balancing the two competing goals, with higher ϵ values corresponding to a greater emphasis on exploration.

EXP3.S.1. This is an advanced MAB algorithm by Auer et al. [2] for the non-stochastic MAB problem. We picked this algorithm for three reasons. First, it is from the venerable EXP3 family, and so likely to be picked up by practitioners. Second, this is one of the EXP3 algorithms that is not parameterized by any constants and thus no parameter tuning is needed. Third, this algorithm is designed to have an optimal worst-case regret, which is a form of regret that suits our problem setting. Note that at its core EXP3.S.1 is a weighted-random algorithm. However, since we do not have a belief metric that corresponds to the one used in EXP3.S.1, we did not put it inside the Weighted-Random group.

4.5 Offline Algorithms

Early on in our research design, we recognized the importance of evaluating a large number of algorithms. Out of budgetary constraints, we have taken a simulation approach so that we can replay the events from previous fuzzings to try out new algorithms. Since we have recorded all the events that may happen during any fuzz campaign of the same input configurations, we can even attempt to compute what an optimal offline algorithm would do and compare the results of our algorithms against it. In the case when the configurations do not yield duplicated bugs, such as in our Inter-Program dataset (§6), we devise a pseudo-polynomial time algorithm that computes the offline optimal. In the other case where duplicated bugs are possible, we propose a heuristic to post-process the solution from the above algorithm to obtain a lowerbound on the offline optimal.

No Duplicates. Assuming that the sets of unique bugs from different configurations are disjoint, our algorithm is a small variation on the dynamic programming solution to the Bounded Knapsack problem. Let K be the number of configurations and B be the total number of unique bugs from all K configurations. Let $t(i,b)$ be the minimum amount of time it takes for configuration i to produce b unique bugs. Note that $t(i,b)$ is assumed to be ∞ when configuration i never produces b unique bugs in our dataset. We claim that $t(i,b)$ can be pre-computed for all $i \in [1,K]$ and $b \in [0,B]$, where each entry takes amortized $O(1)$ time given how events are recorded in our system.

Let $m(i,b)$ be the minimum amount of time it takes for configurations 1 through i to produce b unique bugs. We want to compute $m(K,b)$ for $b \in [0,B]$. By definition, $m(1,b) = t(1,b)$ for $b \in [0,B]$. For $i > 1$, observe that $m(i,b) = \min_{c\in[0,B]}\{t(i,c)+m(i-1,b-c)\}$. This models partitioning the b unique bugs into c unique bugs from configuration i and $(b-c)$ unique bugs from configurations 1 through $(i-1)$. Computing each $m(i,b)$ entry takes $O(B)$ time. Since there are $O(K \times B)$ entries, the total running time is $O(K \times B^2)$.

Discounting Duplicates. The above algorithm is incorrect when the sets of unique bugs from different configurations are not disjoint. This is because the recurrence formula of $m(i,b)$ assumes that the c unique bugs from configuration i are different from the $(b-c)$ unique bugs from configurations 1 through $(i-1)$. In this case, we propose a heuristic to compute a lowerbound on the offline optimal.

After obtaining the $m(i,b)$ table from the above, we post-process bug counts by the following discount heuristic. First, we compute the maximum number of bugs that can be found at each time by the above algorithm by examining the K-th row of the table. Then, by scanning forward from time 0, whenever the bug count goes up by one due to a duplicated bug (which must have been found using another configuration), we discount the increment. Since the optimal offline algorithm can also pick up exactly the same bugs in the same order as the dynamic programming algorithm, our heuristic is a valid lowerbound on the maximum number of bugs that an optimal offline algorithm would find.

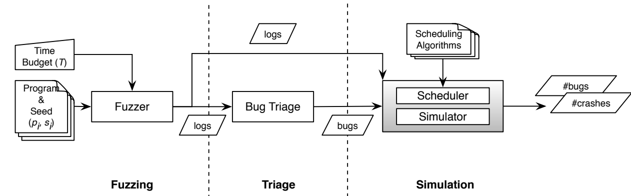


Figure 1: FuzzSim architecture.

5 Design & Implementation

This section presents FuzzSim, our replay-based fuzz simulation system built for this project. We describe the three steps in FuzzSim and explain the benefit of its design, which is then followed by its implementation detail. Of special note is that we are releasing our source code and our datasets in support of open science at the URL found in §5.2.

5.1 Overview

FuzzSim is a simulation system for black-box mutational fuzzing that is designed to run different configuration scheduling algorithms using logs from previous fuzzings. Figure 1 summarizes the design of FuzzSim, which employs a three-step approach: (1) fuzzing, (2) triage, and (3) simulation.

Fuzzing. The first step is fuzzing and collecting run logs from a fuzzer. FuzzSim takes in a list of program-seed pairs (p_i, s_i) and a time budget T. It runs a fuzzer on each configuration for the full length of the time budget T and writes to the log each time a crash occurs. Log entries are recorded as 5-tuples of the form (p_i, s_i, time stamp, #runs, mutation identifier).

In our implementation, we fuzz with **zzuf**, one of the most popular open-source fuzzers. **zzuf** generates a random input from a seed file as described in §2.1. The randomization in **zzuf** can be reproduced given the mutation identifier, thus enabling us to reproduce a crashing input from its seed file and the log entry associated with the crash. For example, an output tuple of (**FFMpeg**, a.avi, 100, 42, 1234) specifies that the program **FFMpeg** crashed at the 100-th second with an input file obtained from "a.avi" according to the mutation identifier 1234. Interested readers may refer to **zzuf** [16] for details on mutation identifiers and the actual implementation.

The deterministic nature of **zzuf** allows FuzzSim to triage bugs after completing all fuzz runs first. In other words, FuzzSim does not compute bug identifiers during fuzzing and instead re-derives them using the log. This does not affect any of our algorithms since none of them relies on the actual IDs. In our experiments, we have turned off address space layout randomization (ASLR) in both the fuzzing and the triage steps in order to reproduce the same crashes.

Triage. The second step of FuzzSim maps crashing inputs found during fuzzings into bugs. At a high level, the triage phase takes in the list of 5-tuples (p_i, s_i, time-stamp, #runs, mutation identifier) logged during the fuzzing step and outputs a new list of 5-tuples of the form (p_i, s_i, time-stamp, #runs, bug identifier). More specifically, FuzzSim replays each recorded crash under a debugger to collect stack traces. If FuzzSim does not detect a crash during a particular replay, then we classify that test case to be a non-deterministic bug and discard it.

We then use the collected stack traces to produce bug identifiers, essentially hashes of the stack traces. In particular, we use the fuzzy stack hash algorithm [19], which identifies bugs by hashing the normalized line numbers from a stack trace. With this algorithm, the number of stack frames to hash has a significant influence on the accuracy of bug triage. For example, taking the full stack trace often leads to misclassifying a single bug into multiple bugs, whereas taking only the top frame can easily lead to two different bugs being mis-classified as one. To match the state of the art, FuzzSim uses the top 3 frames as suggested in [19]. We stress that even though inaccurate bug triage may still occur with this choice of parameter, perfecting bug triage techniques is beyond the scope of this paper.

Simulation. The last step simulates a fuzz campaign on the collected ground-truth data from the previous steps using a user-specified scheduling algorithm. More formally, the simulation step takes in a scheduling algorithm and a list of 5-tuples of the form (p_i, s_i, timestamp, #runs, bug identifier) and outputs a list of 2-tuples (timestamp, #bugs) that represent the accumulated time before the corresponding number of unique bugs are observed under the given scheduling algorithm.

Since FuzzSim can simulate any scheduling algorithm in an offline fashion using the pre-recorded ground-truth data, it enables us to efficiently compare numerous scheduling algorithms without actually running a large number of fuzz campaigns. During replay, FuzzSim outputs a timestamp whenever it finds a new bug. Therefore, we can easily plot and compare different scheduling algorithms by comparing the number of bugs produced under the same time budget.

We summarize FuzzSim's three-step algorithm below.

```
Fuzzing: ({(p_i, s_i)}, T)
    → {p_i, s_i, timestamp, #runs, mutation id}
Triage: {(p_i, s_i, timestamp, #runs, mutation id)}
    → {(p_i, s_i, timestamp, #runs, bug id)}
Simulation: {(p_i, s_i, timestamp, #runs, bug id)}
    → {(timestamp, #bugs)}
```

Algorithm 1: FuzzSim algorithms.

5.2 Implementation & Open Science

We have implemented our data collection and bug triage modules in approximately 1,000 lines of OCaml. This includes the capability to run and collect crash logs from Amazon EC2. We used **zzuf** version 0.13. Our scheduling engine is also implemented in OCaml and spans about 1,600 lines. This covers the 26 online and the 2 offline algorithms presented in this paper.

We invite our fellow researchers to become involved in this line of research. In support of open science, we release both our datasets and the source code of our simulator at http://security.ece.cmu.edu/fuzzsim/.

6 Evaluation

To evaluate the performance of the 26 algorithms presented in §4, we focus on the following questions:

1. Which scheduling algorithm works best for our datasets?
2. Why does one algorithm outperform the others?
3. Which of the two epoch types—fixed-run or fixed-time—works better, and why?

6.1 Experimental Setup

Our experiments were performed on Amazon EC2 instances that have been configured with a single Intel 2GHz Xeon CPU core and 4GB RAM each. We used the most recent Debian Linux distribution at the time of our experiment (April 2013) and downloaded all programs from the then-latest Debian Squeeze repository. Specifically, the version of FFMpeg we used is SVN-r0.5.10-4:0.5.10-1, which is based on a June 2012 FFMpeg release with Debian-specific patches.

6.2 Fuzzing Data Collection

Our evaluation makes use of two datasets: (1) FFMpeg with 100 different input seeds, and (2) 100 different Linux applications, each with a corresponding input seed. We refer to these as the "intra-program" and the "inter-program" datasets respectively.

For the intra-program dataset, we downloaded 10,000 video/image sample files from the MPlayer website at http://samples.mplayerhq.hu/. From these samples, we selected 100 files uniformly at random and took them as our input

Dataset	#runs	#crashes	#bugs
Intra-program	636,998,978	906,577	200
Inter-program	4,868,416,447	415,699	223

Table 1: Statistics from fuzzing the two datasets.

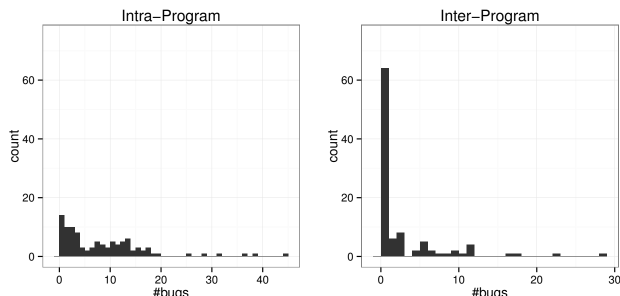


Figure 2: Distribution of the number of bugs per configuration in each dataset.

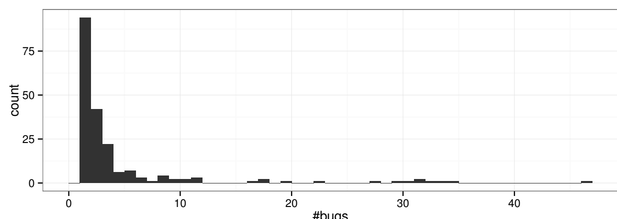
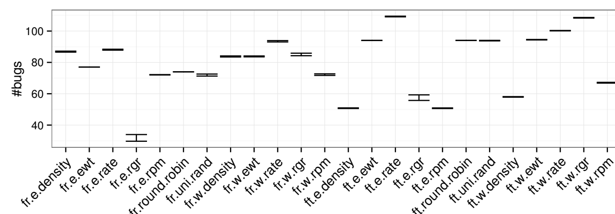
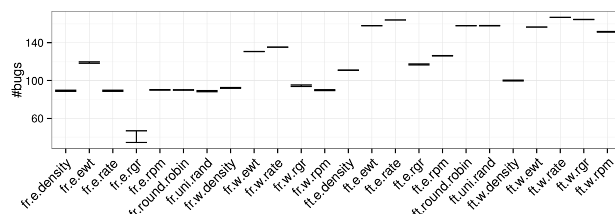


Figure 3: Distribution of bug overlaps across multiple seeds for the intra-program dataset.



(a) Intra-program.



(b) Inter-program.

Figure 4: The average number of bugs over 100 runs for each scheduling algorithm with error bars showing a 99% confidence interval. "ft" represents fixed-time epoch; "fr" represents fixed-run epoch; "e" represents ϵ-Greedy; "w" represents Weighted-Random.

seeds. The collected seeds include various audio and video formats such as ASF, QuickTime, MPEG, FLAC, etc. We then used `zzuf` to fuzz FFMpeg with each seed for 10 days.

For the inter-program dataset, we downloaded 100 different file conversion utilities in Debian. To select these 100 programs, we first enumerated all file conversion packages tagged as "use::converting" in the Debian package tags interface (debtags). From this list of packages, we manually identified 100 applications that take a file name as a command line argument. Then we manually constructed a valid seed for each program and the actual command line to run it with the seed. After choosing these 100 program-seed pairs, we fuzzed each for 10 days as well. In total, we have spent 48,000 CPU hours fuzzing these 200 configurations.

To perform bug triage, we identified and re-ran every crashing input from the log under a debugger to obtain stack traces for hashing. After triaging with the fuzzy stack hash algorithm described in §5.1, we found 200 bugs from the intra-program dataset and 223 bugs from the inter-program dataset. Table 1 summarizes the data collected from our experiments. The average fuzzing throughput was 8 runs per second for the intra-program dataset and 63 runs per second for the inter-program dataset. This difference is due to the higher complexity of FFMpeg when compared to the programs in the inter-program dataset.

6.3 Data Analysis

What does the collected fuzzing data look like? We studied our data from fuzzing and triage to answer two questions: (1) How many bugs does a configuration trigger? (2) How many bugs are triggered by multiple seeds in the intra-program dataset?

We first analyzed the distribution of the number of bugs in the two datasets. On average, the intra- and the inter-program datasets yielded 8.2 and 2.4 bugs per configuration respectively. Figure 2 shows two histograms, each depicting the number of occurrences of bug counts. There is a marked difference in the distributions from the two datasets: 64% of configurations in the inter-program dataset produce no bugs, whereas the corresponding number in the intra-program dataset is 15%. We study the bias of the bug count distribution in §6.4.

Second, we measured how many bugs are shared across seeds in the intra-program dataset. As an extreme case, we found a bug that was triggered by 46 seeds. The average number of seeds leading to a given bug is 4. Out of the 200 bugs, 97 were discovered from multiple seeds. Figure 3 illustrates the distribution of bug overlaps. Our results suggest that there is a small overlap in the code exercised by different seed files even though they have been chosen to be of different types. Although this shows that our bug disjointness assumption in the WCCP model does not always hold in practice, the low average number of seeds leading to a given bug in our dataset means that the performance of our algorithms should not have been severely affected.

6.4 Simulation

We now compare the 26 scheduling algorithms based on the 10-day fuzzing logs collected for the intra- and inter-program datasets. To compare the performance of scheduling algorithms, we use the total number of unique bugs reported by the bug triage process. Recall from §4.4 that these algorithms vary across three dimensions: (1) epoch types, (2) belief metrics, and (3) MAB algorithms. For each valid combination (see Table 2), we ran our simulator 100 times and averaged the results to study the effect of randomness on each scheduling algorithm. In our experiments, we allocated 10 seconds to each epoch for fixed-time campaigns and 200 runs for fixed-run campaigns. For the ϵ-Greedy algorithm, we chose ϵ to be 0.1.

Table 2 summarizes our results. Each entry in the table represents the average number of bugs found by 100 sim-

Dataset	Epoch	MAB algorithm	#bugs found for each belief				
			RPM	EWT	Density	Rate	RGR
Intra-Program	Fixed-Run	ϵ-Greedy	72	77	87	88	32
		Weighted-Random	72	84	84	**93**	85
		Uniform-Random			72		
		EXP3.S.1			58		
		Round-Robin			74		
	Fixed-Time	ϵ-Greedy	51	94	51	**109**	58
		Weighted-Random	67	94	58	100	108
		Uniform-Random			94		
		EXP3.S.1			95		
		Round-Robin			94		
Inter-Program	Fixed-Run	ϵ-Greedy	90	119	89	89	41
		Weighted-Random	90	131	92	**135**	94
		Uniform-Random			89		
		EXP3.S.1			72		
		Round-Robin			90		
	Fixed-Time	ϵ-Greedy	126	158	111	164	117
		Weighted-Random	152	157	100	**167**	165
		Uniform-Random			158		
		EXP3.S.1			161		
		Round-Robin			158		

Table 2: Comparison between scheduling algorithms.

ulations of a 10-day campaign. We present ϵ-Greedy and Weighted-Random at the top of each epoch-type row group, each showing five entries that correspond to the belief metric used. For the other three MAB algorithms, we only show a single entry in the center because these algorithms do not use our belief metrics. Figure 4 describes the variability of our data using error bars showing a 99% confidence interval. Notice that 94% of our scheduling algorithms have a confidence interval that is less than 2 (bugs). RGR gives the most volatile algorithms. This is not surprising because RGR tends to under-explore by focusing too much on bug-yielding configurations that it encounters early on in a campaign. In the remainder of this section, we highlight several important aspects of our results.

Fixed-time algorithms prevail over fixed-run algorithms. In the majority of Table 2, except for RPM and Density in the intra-program dataset, fixed-time algorithms always produced more bugs than their fixed-run counterparts. Intuitively, different inputs to a program may take different amounts of time to execute, leading to different fuzzing throughputs. A fixed-time algorithm can exploit this fact and pick configurations that give higher throughputs, ultimately testing a larger fraction of the input space and potentially finding more bugs. To investigate the above exceptions, we have also performed further analysis on the intra-program dataset. We found that the performance of the fixed-time variants of RPM and Density greatly improves in longer simulations. In particular, *all* fixed-time algorithms outperform their fixed-run counterparts after day 11.

Along the same line, we observe that fixed-time algorithms yield 1.6× more bugs on average when compared to their fixed-run counterparts in the inter-program dataset. In contrast, the improvement is only 1.1× in the intra-program dataset. As we have explained above, fixed-time algorithms tend to perform more fuzz runs and potentially finding more bugs by taking advantage of faster configurations. Thus, if the runtime distribution of fuzz runs is more biased, as in the case of the inter-program dataset, then fixed-time algorithms tend to gain over their fixed-run counterparts.

Time-normalization outperforms runs-normalization. In our results, EWT always outperforms RPM and Rate always outperforms Density. We believe that this is because EWT and Density do not spend more time on slower programs and slower programs are not necessarily buggier. The latter hypothesis seems highly plausible to us; if true, it would imply that time-normalized belief metrics are more desirable than runs-normalized metrics.

Fixed-time Rate works best. In both datasets, the best-performing algorithms use fixed-time epochs and Rate as belief (entries shown in boldface in Table 2). Since Rate can be seen as a time-normalized variant of RGR, this gives further evidence of the superiority of time normalization. In addition, it also supports the plausibility of the bug prior.

6.5 Speed of Bug Finding

Besides the number of bugs found at the end of a fuzz campaign, the speed at which bugs are discovered is also an important metric for evaluating scheduling algorithms. We address two questions in this section. First, is there a scheduling algorithm that prevails throughout an entire fuzz campaign? Second, how effective are the algorithms with respect to our offline algorithm in §4.5? To answer the questions, we first show the speed of each algorithm in Figure 5 and Figure 6 by computing the number of bugs found over time. For brevity and readability, we picked for each belief metric the algorithm that produced the greatest average number of unique bugs at the end of the 10-day simulations.

Speed. We observe that Rate and RGR are in the lead for the majority of the time during our 10-day simulations. In other words, not only do they find more unique bugs at the end of the simulations, but they also outperform other algorithms at almost any given time. This lends further credibility to the bug prior.

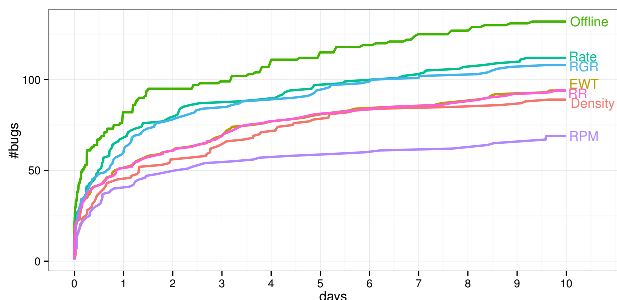


Figure 5: Bug finding speed of different belief-based algorithms for the intra-program dataset.

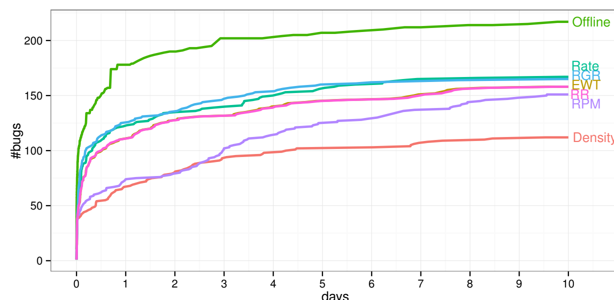


Figure 6: Bug finding speed of different belief-based algorithms for the inter-program dataset.

Effectiveness. We also compare the effectiveness of each algorithm by observing how it compares against our offline algorithm. We have implemented the offline algorithm discussed in §4.5 including the post-processing step that discounts duplicated bugs and computed the solution for each dataset. The numbers of bugs found by the offline algorithm for the intra- and the inter-program datasets are 132 and 217 respectively. (Notice that due to bug overlaps and the discount heuristic, these are lowerbounds on the offline optimal.) As a comparison, Rate found 83% and 77% of bugs in the intra- and inter-program datasets, respectively. Based on these numbers, we conclude that Rate-based algorithms are effective.

6.6 Comparison with CERT BFF

At present, the CERT Basic Fuzzing Framework (BFF) [14] is the closest system that makes use of scheduling algorithms for fuzz campaigns. In this section, we evaluate the effectiveness of BFF's scheduling algorithm using our simulator.

Based on our study of the source code of BFF v2.6 (the latest version as of this writing), it uses a fixed-run weighted-random algorithm with Density ($\frac{\#\text{bugs}}{\#\text{runs}}$) as its belief metric. However, a key feature of BFF prevented us from completely implementing its algorithm in our simulation framework. In particular, while BFF focuses on fuzzing a single program, it considers not only a collection of seeds but also a set of predetermined mutation ratios. In other words, instead of choosing program-seed pairs as in our experiments, BFF chooses seed-ratio pairs with respect to a single program. Since our simulator does not take mutation ratio into account, it can only emulate BFF's algorithm in configuration selection using a fixed mutation ratio. We note that adding the capability to vary the mutation ratio is prohibitively expensive for us: FuzzSim is an offline simulator, and therefore we need to collect ground-truth data for all possible configurations. Adding a new dimension into our current system would directly multiply our data collection cost.

Going back to our evaluation, let us focus on the Weighted-Random rows in Table 2. Density with fixed-run epochs (BFF) yields 84 and 92 bugs in the two datasets. The corresponding numbers for Rate with fixed-time epochs (our recommendation) are 100 and 167, with respective improvements of 1.19× and 1.82× (average 1.5×). Based on these numbers, we believe future versions of BFF may benefit from switching over to Rate with fixed-time epochs.

7 Related Work

Since its introduction in 1990 by Miller et al. [18], fuzzing in its various forms has become the most widely-deployed technique for finding bugs. There has been extensive work to improve upon their ground-breaking work. A major thrust of this research concerns the generation of test inputs for the target program and the two main paradigms in use are mutational and generational fuzzing [17].

More recently, sophisticated techniques for dynamic test generation have been applied in fuzzing [8, 11]. White-box fuzzing [7] is grounded in the idea of "data-driven improvement," which uses feedback from previous fuzz runs to "focus limited resources on further research and improve future runs." The feedback data used in determining inputs is obtained via symbolic execution and constraint solving; other work in feedback-driven input generation relies on taint analysis and control flow graphs [13, 20]. Our works bears some similarity to feedback-driven or evolutionary fuzzing in that we also use data from previous fuzz runs to improve fuzzing effectiveness. However, the black-box nature of our approach implies that feedback is limited to observing crashes. Likewise, our focus on mutating inputs means that we do not construct brand new inputs and instead rely on selecting among existing configurations. Thus, our work can be cast as dynamic scheduling of fuzz configurations.

Despite its prominence, we know of no previous work that has systematically investigated the effectiveness of different scheduling algorithms in fuzzing. Our approach focuses on allocating resources for black-box mutational fuzzing in order to maximize the number of unique bugs found in any period of time. The closest related work is the CERT Basic Fuzzing Framework (BFF) [14], which considers parameter selection for `zzuf`. Like BFF, we borrow techniques from Multi-Armed Bandits (MAB) algorithms. However, unlike BFF, which considers repeated fuzz runs as independent Bernoulli trials, we model this process as a Weighted Coupon Collector's Problem (WCCP) with unknown weights to capture the decrease in the probability of finding a new bug over the course a fuzz campaign.

In constructing our model, we draw heavily on research in software reliability as well as random testing. The key insight of viewing random testing as coupon collecting was recently made in [1]. A key difference between our work and [1] is that their focus is on the formalization of random testing, whereas our goal is to maximize the number of bugs found in a fuzz campaign. Software reliability refers to the probability of failure-free operation for a specified time period

and execution environment [6]. As a measure of software quality, software reliability is used within the software engineering community to "plan and control resources during the development process" [12], which is similar to the motivation behind our work.

8 Conclusion and Future Work

In this paper we studied how to find the greatest number of unique bugs in a fuzz campaign. We modeled black-box mutational fuzzing as a WCCP process with unknown weights and used the condition in the No Free Lunch theorem to guide us in designing better online algorithms for our problem. In our evaluation of the 26 algorithms presented in this paper, we found that the fixed-time weighted-random algorithm with the Rate belief metric shows an average of 1.5× improvement over its fixed-run Density-based counterpart, which is currently used by the CERT Basic Fuzzing Framework (BFF). Since our current project does not investigate the effect of varying the mutation ratio, a natural follow-up work would be to investigate how to add this capability to our system in an affordable manner.

Acknowledgment

The authors thank Will Dormann, Jonathan Foote, and Allen Householder of CERT for encouragement and fruitful discussions. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution.

References

[1] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal Analysis of the Effectiveness and Predictability of Random Testing. In *International Symposium on Software Testing and Analysis*, pages 219–229, 2010.

[2] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The Nonstochastic Multiarmed Bandit Problem. *Journal on Computing*, 32(1):48–77, 2002.

[3] P. Auer, N. Cesa-Bianchi, and F. Paul. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.

[4] T. Avgerinos, S. K. Cha, B. T. H. Lim, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2011.

[5] D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, 1985.

[6] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, pages 85–103, 2007.

[7] E. Bounimova, P. Godefroid, and D. Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the International Conference on Software Engineering*, pages 122–131, 2013.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 209–224, 2008.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[10] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 57–72, 2001.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security. *Communications of the ACM*, 55(3):40–44, 2012.

[12] A. L. Goel. Software Reliability Models: Assumptions, Limitations, and Applicability. *IEEE Transactions on Software Engineering*, 11(12):1411–1423, 1985.

[13] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated Test Data Generation Using An Iterative Relaxation Method. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 231–244, 1998.

[14] A. D. Householder and J. M. Foote. Probability-Based Parameter Selection for Black-Box Fuzz Testing. Technical Report August, CERT, 2012.

[15] B. D. Jovanovic and P. S. Levy. A Look at the Rule of Three. *The American Statistician*, 51(2):137–139, 1997.

[16] C. Labs. zzuf: multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf.

[17] R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: The State of the Art. Technical Report DSTO–TN–1043, Defence Science and Technology Organisation, 2012.

[18] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[19] D. Molnar, X. Li, and D. Wagner. Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the USENIX Security Symposium*, pages 67–82, 2009.

[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering*, pages 75–84, 2007.

[21] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 3–17, 2000.

[22] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.