Empirical Analysis and Modeling of Black-Box Mutational Fuzzing

Mingyi Zhao and Peng Liu

College of Information Sciences and Technology
Pennsylvania State University
{muz127,pliu}@ist.psu.edu

Abstract. Black-box mutational fuzzing is a simple yet effective method for finding software vulnerabilities. In this work, we collect and analyze fuzzing campaign data of 60,000 fuzzing runs, 4,000 crashes and 363 unique bugs, from multiple Linux programs using CERT Basic Fuzzing Framework. Motivated by the results of empirical analysis, we propose a stochastic model that captures the long-tail distribution of bug discovery probability and exploitability. This model sheds light on practical questions such as what is the expected number of bugs discovered in a fuzzing campaign within a given time, why improving software security is hard, and why different parties (e.g., software vendors, white hats, and black hats) are likely to find different vulnerabilities. We also discuss potential generalization of this model to other vulnerability discovery approaches, such as recently emerged bug bounty programs.

Keywords: Mutational Fuzzing, Software Vulnerability, Empirical Analysis, Stochastic Modeling

1 Introduction

Software vulnerability is the root cause of many security breaches. However, it has also been observed that discovering software vulnerability is hard. While software companies invest heavily to eliminate vulnerabilities, other parties including white hats [32] and black hats [27] are frequently able to find new vulnerabilities, even when endowed with less resources (e.g. computing power, manpower, information). In addition, investment in software security exhibits diminishing returns [15], which has also been discussed in the field of software reliability growth [6].

Understanding these phenomena has important theoretical and practical implications. Existing work on the economy of security usually involves models of software vulnerability discovery [16, 18, 29]. Such models can be improved by empirical analysis of real vulnerability discovery data. The effort of studying vulnerability discovery also help practitioners. For example, software companies can make better decisions on the level of security investment and the extent of collaboration with outside security researchers (e.g., white hats) [32]. Cyber-insurance organizations might also be able to assess the security of customers more accurately [5].

In this work, we conduct an empirical analysis and propose models for black box mutational fuzzing. Introduced in early 1990s [21], black box mutational fuzzing remains an effective method for discovering real world vulnerabilities [9, 17]. Its basic idea is very simple. Given a program and a set of diverse seed files, the fuzzing tool randomly mutates the files and use the program to process them. Once the program crashes, a triaging tool identifies the underlying bug and determines its properties such as exploitability. This simplicity makes black-box mutational fuzzing not only easy to use, but also easy to analyze and model.

We first apply black-box mutational fuzzing to multiple Linux programs and collect data from each fuzzing campaign, based on the CERT Basic Fuzzing Framework (BFF) [14] (Section 3). Our dataset contains 60,000 fuzzing runs, 4,000 crashes and 363 unique bugs. Then, we empirically analyze the data and discuss the long-tail distribution of discovery probability (Section 4), as well as the distribution of exploitability of bugs (Section 6.3). Motivated by the empirical analysis, we propose a stochastic model of black-box mutational fuzzing (Section 5.1). The model is derived from software reliability growth models [4, 6, 10, 23]. However, one unique contribution of our model is that we assume the arrival rates of individual bugs follow a power law distribution, which is consistent with our data. Together with a simulation model (Section 5.2), we attempt to explain phenomena discussed at the beginning of this section. First, we provide a method to estimate the expected discovery outcome, which sheds light on the diminishing return of security investment (Section 6.1). Next, we explain why it is hard for software companies to eliminate the vulnerability stockpile of black hats (Section 6.2). Finally, we discuss several potential directions for future work, including the generalization of this model to other vulnerability discovery mechanisms (Section 7). All scripts and data are published online[1] for reproducible research.

2 Related Work

Black-box mutational fuzzing has been widely used in software vulnerability discovery since early 1990s, when Miller et al. surprisingly found out that random inputs crash 25% - 33% of Unix utilities [21]. Since then, black-box mutational fuzzing has been used to find numerous real world bugs and security vulnerabilities in various programs [11, 14, 22]. Compared with other forms of more sophisticated fuzzing approaches, such as generational fuzzing [20], whitebox fuzzing [12], taint-based fuzzing [30], etc., black-box mutational fuzzing is simpler and easier to use, but is usually inferior in terms of code coverage.

More recently, various methods were proposed to improve the effectiveness of black-box mutational fuzzing . Householder and Foote studied the problem of selecting seeds and fuzzing ratio using BFF [14]. The basic idea is to have more selection weight on parameter values that yield higher crash density in the past. Woo et al. considered a similar scheduling problem in which the target

[1] `github.com/movingname/fuzzingModel`

program of each fuzzing run is also selected on the fly [31]. They designed several online scheduling algorithms and showed an average of 1.5× improvement over the one used in BFF. Rebert et al. designed and evaluated 6 seed selection algorithms [28]. The motivation of our work is different from but complementary to these work. Instead of optimizing the fuzzing process, we want to understand the fuzzing process better, based on empirical analysis and theoretical modeling.

The stochastic model built in this paper is derived from software reliability models [4, 6, 10, 23], since fuzzing or vulnerability discovery in general is one particular approach to improve the software reliability. However, different from existing work, we assume that the arrival rates of the individual bugs follow power law distribution. This enables us to obtain similar observations on the difficulty of software reliability growth [4, 6], or in other words, diminishing returns of software vulnerability discovery. In addition, we also uniquely use the power law-based stochastic model to explain why other parties (e.g. black hats) seem always being able to discovery unique vulnerabilities in Section 6.2. We further analyze the exploitability of bugs in Section 6.3, which is missing from software reliability growth models.

This paper assumes that the discovery probability of bugs follows power law distribution. Such long-tail distributions have been observed and discussed in various cyber security domains recently. Allodi showed that vulnerability exploitation in several common programs may follow power law distribution [2], which can be used for vulnerability prioritization. Maillart and Sornette showed that the sizes of personal identify theft follow power law distribution [19]. Finally, Edwards et al. found that data breach size is log-normally distributed while the daily frequency of breaches can be described by a negative binomial distribution [8]. These results can be used to predict data breaches and their associated cost.

3 BFF and Data Collection

Figure 1 shows the workflow of black-box mutational fuzzing. We have created several Python scripts for seed collection, code coverage, seed selection and data analysis. The fuzzing tool and triaging tool is from the CERT Basic Fuzzing Framework (BFF) [14]. BFF is shown to be effective in finding real vulnerabilities in various programs, and has been used in previous work on improving black-box mutational fuzzing [28, 31] as well. Next, we outline the details of our experiment.

Step 1. Target Selection By combining the lists of target programs used in the literature [7, 14, 28, 31], we have collected 18 programs that handle various types of video, audio, graphical, and document inputs. Table 1 list all 9 programs in which BFF has successfully found bugs. We have also tried to apply fuzzing to the following programs: a2mp3, eog, gifsicle, mplayer, mp3blaster, mpg123, moc, Outside In Viewer 8.5.2, and pdf2svg. However, for any of these programs, BFF triggers less than 3 or even 0 crashes. We therefore exclude them from the following analysis.

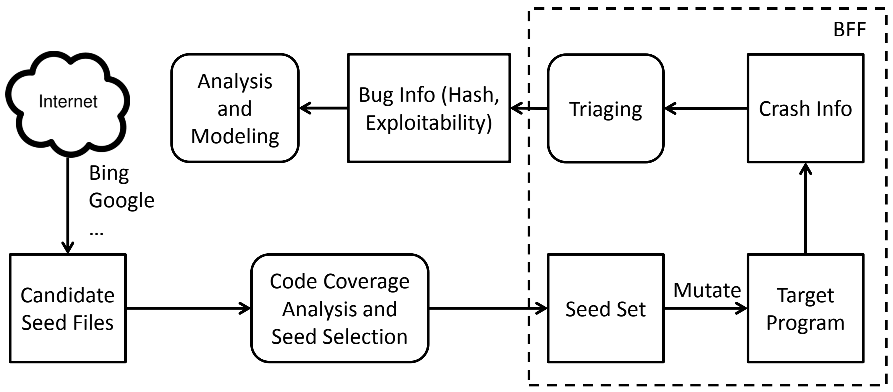

Fig. 1: The fuzzing experiment workflow.

Step 2. Seed Collection and Selection We have collected thousands of candidate seeds files, including pdf documents, mp3 files, videos and images, from search engines like Bing and Google. The # cand columns of Table 1 shows the number of candidate seed files for each program. We then write a script to collect the basic blocks (bbls) covered by each seed using the Intel Pin framework. In general, the higher coverage of seeds, the more vulnerabilities will be found in fuzzing [28]. Next, we select 50 seed files to form the final seed set for each program, using a simple greedy algorithm that maximizes the coverage in each iteration. Table 1 shows that the final seed sets still achieve similar levels of coverage (% bbls column).

Step 3. Fuzzing We use BFF as the fuzzing tool and use its default *fuzzing configuration*. The main configuration parameter is the seed used in each fuzzing run, and the *fuzzing ratio*, which indicates how many bits in the seed will be flipped. We use the default probability-based parameter selection method implemented in BFF [14]. The outcome of a fuzzing run is either a crash or nothing, while the result of a fuzzing campaign is a sequence of crashes caused by software bugs in the program. Since multiple crashes could correspond to the same bug, we need a triaging step to map a crash to the corresponding bug.

Step 4. Triaging Once a crash is encountered, BFF will run the triaging step, which calculates the hash for the underlying bug based on the stack trace[2], minimizes the input that triggers the crash, and determines whether the bug is exploitable or not. Similar to other triaging tools such as the !exploitable

[2] The method used to generate the hash is an extension of the fuzzy stack hash method proposed in the literature [24].

for Windows OS and CrashWrangler for Mac OS X, the CERT Triage Tools in BFF assigns one of the following exploitability levels to each crash: unknown, not_exploitable, probably_not_exploitable, probably_exploitable and exploitable.

Step 5. Data Analysis At the end, we know the , seed file, configuration, and outcome of each fuzzing run, as well as the hash and exploitability of each bug discovered. We then analyze the data and show statistics of the results in Table 1. We present our main analysis results in the next section.

	Seed Selection			**Fuzzing**			
Program	**# cand**	**# bbls**	**% bbls**	**# runs**	**# crashes**	**# bugs**	**max_freq**
xpdf 3.02-2	2,161	188,023	93.1%	4,303	185	37	73
mupdf[a]	-	-	-	9,900	201	25	61
convert 5.2.0[b]	-	-	-	79,636	32,161	134	3,197
ffmpeg 0.8	787	121,875	86.7%	16,055	3,872	96	863
autotrace 0.31.1	149[c]	-	100%	29,729	2,548	23	593
jpegtran 1.2.0	320	6,837	99.4%	303,898	116	33	31
gif2png 2.5.4-2	1,084	12,772	99.8%	136,768	2,305	7	34
feh 2.2	1,332	56,266	94.8%	5,209	159	5	51
mp3gain 1.5.2	214	7,224	99.9%	1,369	1,451[d]	7	861

Table 1: Seed selection and fuzzing statistics of selected programs. # cands is the number of candidate seed files we collected from the Internet. # bbls is the number of unique basic blocks recored when parsing the candidate seed files. % bbls is the percentage of basic blocks covered by the final seed set.

[a] mupdf and xpdf share same seeds.
[b] We use the seeds provided in the default BFF vm image for convert.
[c] Since the size is small, we use all of the seeds in fuzzing.
[d] Here, # crashes is actually larger than # runs. This is caused by a stack corruption bug that confuses triaging process to correlate the same crash into different bugs [28].

4 The Long-tail Distribution of Bugs

The major goal of fuzzing and any bug discovery effort is to find as many bugs as possible. Moreover, it has been observed that the easiness of discovering different bugs is different. In black-box mutational fuzzing, we can quantify easiness of discovering bug i as its *discovery probability* (λ_i):

$$\lambda_i = \frac{c_i}{t} \tag{1}$$

where c_i is the number of crashes caused by bug i, and t is the number of fuzzing runs in the fuzzing campaign. Then the question is, what is the distribution for bug discovery probability?

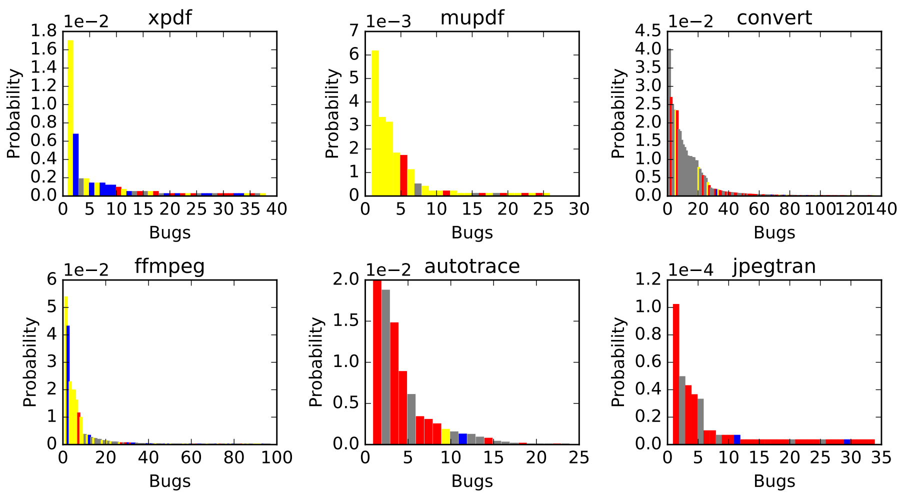

Fig. 2: Probability distributions of bugs triggered in fuzzing campaigns. The color of a bar represents the exploitability of the bug. The meaning of colors are: exploitable (red), probably_exploitable (blue), probably_not_exploitable (yellow), not_exploitable (green), unknown (grey).

In Figure 2, we plot the empirical probability distribution of bugs for all 6 programs with more than 20 bugs discovered. We see that these distributions all have *the long-tail shape*; that is, a few bugs trigger a large number of crashes, while most bugs have only triggered crashes a few times. Many such distributions [26], including vulnerability exploitation [2], have been proposed to follow the power law distribution. We thus propose the following hypothesis:

Hypothesis 1 *The discovery probability of bugs in a program follows a power law distribution.*

More specifically, we assume the following discrete power law distribution [26]:

$$P(\text{discover bug } i \text{ in a fuzzing run}) = \lambda_i = \frac{i^{-\alpha}}{\zeta(\alpha)} \tag{2}$$

where α is the scaling factor of the power law distribution, $\zeta(\alpha)$ is the Riemann ζ-function as the normalizer. As we will show in Section 6, a smaller α leads to more bugs discovered in the same number of fuzzing runs. i is the *rank id* of the bug among all bugs sorted by their discovery probability inside the program. A bug with a larger rank id (lower rank) has lower probability to be discovered, as Equation 2 tells. To complete the probability distribution, we also use λ_1 to represent the probability of no crash. We can think about no crash as a special bug, and it has the highest probability in these 6 fuzzing campaigns.

We next need to estimate the scaling factor α of a power law distribution from the empirical distribution. The most common approach is to use Maximum Likelihood Estimators (MLEs) [3, 26]. However, we could not apply these estimators because *we do not know the true rank id of a bug discovered in fuzzing.* We only know a bug's rank among all discovered bugs. For example, the 20th bug in the empirical data could have the true rank id of 100.

We propose a simulation method to estimate α. We could think a fuzzing campaign with t runs as generating t values form the corresponding power law distribution. We then choose the α that minimizes the difference between the number of unique bugs discovered in the experiment and the number of unique values generated from the distribution. Table 2 shows the estimates of α.

Program	α
xpdf	2.39
mupdf	2.88
convert	2.38
ffmpeg	2.21
autotrace	3.25
jpegtran	3.53

Table 2: Estimates of α.

Because we do not know the true rank id of bugs discovered, it is also difficult to apply goodness-of-fit tests, either through bootstrapping or by comparing with alternative distributions [2, 3]. In this work, we will test the estimates of α by comparing the predicted number of bugs discovered with the actual number of bugs discovered in Section 6.1. More rigorous methods of estimating α and testing the goodness-of-fit are left as future work. In the following sections, we will show that this power law hypothesis enables us to answer some interesting questions related to vulnerability discovery and software security.

5 Modeling a Fuzzing Campaign

We then build models for a fuzzing campaign. First, we propose a stochastic model based on existing software reliability literature [4, 6, 10, 23], in Section 5.1. Although expressive, this stochastic model has two assumptions that might not be realistic. We remove one assumption by proposing a simulation model in Section 5.2.

5.1 A Stochastic Model

Since each fuzzing run is independent from other runs, and the outcome of a fuzzing run is either 1 (crashed) or 0 (not crashed), it is natural to consider the fuzzing process as a Poisson Process $\{N(t), t \geq 0\}$, where $N(t)$ is the number of

Variable	**Explanation**
$N(t)$	The Poisson process for number of crashes in a fuzzing campaign.
$N_i(t)$	The Poisson process corresponding to bug i.
λ_i	The rate for $N_i(t)$ and the discovery probability of bug i.
n	Total number of bugs in the program.
$N'(t)$	The non-homogeneous Poisson process for number of unique bugs.
$\mathcal{D}(t)$	The set of discovered bugs by time t.
$D(t)$	Number of discovered bugs by time t. $D(t) = \lvert\mathcal{D}(t)\rvert$.
$\mathcal{L}(t)$	The set of remaining bugs by time t.

Table 3: Notations.

crashes seen till time t. Furthermore, since crashes are caused by different bugs, we can expressed $N(t)$ as:

$$N(t) = \sum_{i=2}^{\infty} N_i(t) \tag{3}$$

Here, i is the rank id of a bug and t is the number of fuzzing runs. $\{N_i(t), t \geq 0\}$ is the corresponding Poisson process for the i-th bug, and $N_i(t)$ is the number of crashes for the i-th bug we have seen till time t. We can see that the discovery probability of the i-th bug we have discussed in the previous subsection is actually the rate λ_i of the Poisson process $N_i(t)$. A larger λ_i means that bug i causes crashes more frequently.

In a fuzzing campaign, we are mostly interested in the first crash of a bug. This is equivalent to the assumption in the software reliability models that a bug is found and instantly fixed, while the fix does not influence the discovery of other remaining bugs [4]. We define $\mathcal{D}(t)$ as the set of bugs that have already been found by time t, and $\mathcal{L}(t)$ as the set of remaining bugs. So we have:

$$\lambda'(t) = \sum_{i \in \mathcal{L}(t)}^{\infty} \lambda_i = \sum_{i=2}^{\infty} \lambda_i - \sum_{i \in \mathcal{D}(t)}^{\infty} \lambda_i \tag{4}$$

Therefore, we obtain a new non-homogeneous Poission process, $N'(t)$, for the discovery of unique bugs. $\lambda'(t)$ is the arrival rate of new bugs, and the expected time to discover the next bug is $1/\lambda'(t)$.

We currently do not know how to solve Equation 4 analytically. Thus when doing calculation, we replace ∞ with n, in order to obtain an approximate result. Intuitively, we assume there are n bugs in total inside the program. By choosing a larger n, we can further approximate the true result. In our following analysis, we set $n = 1000$. The probability that $i > 1000$ is only 1.3e-4 for ffmpeg ($\alpha = 2.21$), and 9.03e-9 for jpegtran ($\alpha = 3.53$).

In addition, this stochastic model relies on the following two assumptions:

Assumption 1 *In one fuzzing run, multiple bugs can be triggered.*

However, in BFF, each fuzzing run stops at the first crash, which is then triaged to one bug. Thus, with Assumption 1, the model will slightly overestimate the number of bugs discovered, as we will see in Section 6.1. But we expected that this effect is small because most bugs have low discovery probability (Figure 2), and the chance that multiple bugs are triggered in the same fuzzing run is even lower.

Assumption 2 *The discovery probability distribution is the same for all fuzzing runs in a fuzzing campaign.*

This assumption also oversimplifies the reality. Since the fuzzing seeds and fuzzing ratio are different among fuzzing runs, each fuzzing run will explore a unique input space and be able to trigger a different subset of all latent bugs. We will discuss this more in Section 7.

Improving this stochastic model by relaxing these two assumptions is challenging, which is left as a future work. In the next sub section, we propose a simulation model that removes Assumption 1.

5.2 A Simulation Model

Similar to the discussion in Section 4, we could think a fuzzing campaign with t runs as generating t values form the corresponding power law distribution. Algorithm 1 returns a simulated bug discovery sequence as well as unique bugs discovered, given α and t as the inputs . Step 1 and 2 can be implemented using existing software package [3]. In step 5, we add the condition $id > n$ because we will compare the simulation model with the stochastic model.

Algorithm 1: Simulate a fuzzing campaign.

```
  input : $\alpha$ of the bug distribution, and $t$, the number of fuzzing runs
  output: Simulated bug discovery sequence and unique bugs discovered
1 dist = powerlaw($\alpha$, xmin=1, discrete=True) ;
2 seq = dist.gen_random($t$);
3 bugs = {};
4 foreach $id \in bugs$ do
5     if $id == 1$ or $id > n$ then
6         continue;
7     if $id \notin bugs$ then
8         bugs.add(id);
9 return seq, bugs;
```

In this simulation model, we remove Assumption 1 since each fuzzing run only yields at most one bug discovery. In Section 6.1, we will compare the predicted numbers of bugs discovered by these two models, and the actual number of bugs discovered.

6 Analysis Results

We present 3 analysis results in this section. We first use the models presented in the last section to calculate the expected number of bugs discovered, and discuss the diminishing returns in software security. We then examine the order of bug discovery to explain why different parties are likely to find different bugs. Finally, we empirically study the exploitability of bugs and discuss its implications.

6.1 Expected Number of Bugs Discovered

The first question is, what is the expected number of unique bugs find by time t? Under the stochastic model proposed in Section 5.1, we know that the time of the first occurrence of each bug follows the exponential distribution with parameter λ_i. Therefore, the probability of bug i undiscovered by time t is $e^{-\lambda_i t}$, and the expected number of undiscovered bug at time t is $\sum_{i=2}^{n} e^{-\lambda_i t}$. We then know that the number of expected bugs discovered by time t is:

$$E[D(t)] = n - \sum_{i=2}^{n} e^{-\lambda_i t} = n - \sum_{i=2}^{n} e^{-\frac{i^{-\alpha}}{\zeta(\alpha)} t} \tag{5}$$

We also use the simulation model proposed in Section 5.2 to obtain $E[D(t)]$. We repeat the simulation 10 times and take the average number of bugs discovered by time t as $E[D(t)]$. As we have discussed in Section 5.1, we set $n = 1000$ for both models.

In Figure 3, we show the plots of expected bugs discovered based on the Poisson process and the simulation, and the real trajectory, of 6 fuzzing campaigns. We see that the predicted curves from both models are close to the real curve, except for autotrace. We suspect that the the large prediction error for autotrace is due to a poor fit of power law to its empirical distribution. We plan to further investigate this in the future. In addition, the curve of the stochastic model is generally above the other two. This can be partly explained by Assumption 1, as we have discussed in Section 5.1. In general, the simulation model gives more accurate prediction for the 6 fuzzing campaigns than the stochastic model.

The concave shape of all curves show the *diminishing returns*: as the fuzzing campaign enters the long tail, the rate of discovery ($\lambda'(t)$) decreases, and the number of bugs discovered in the same amount of time reduces. This diminishing of return is consistent with our experience of fuzzing and software reliability growth [10, 23]. A software company can use the two models to decide how long the fuzzing campaign shall run. First, the company need to run a fuzzing campaign for a limited amount of time, in order to estimate α. Then, the company needs to define the reliability and security utility gain of finding a bug, and

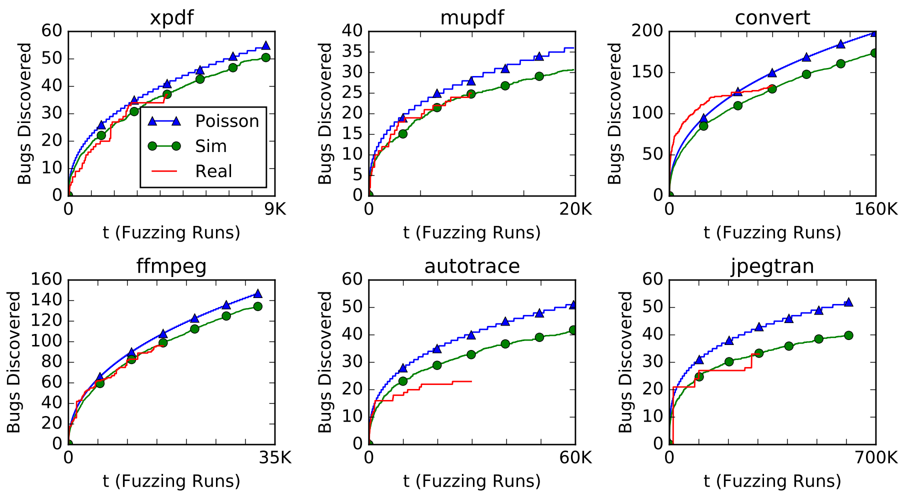

Fig. 3: Plots of expected number of bugs discovered ($E[D(t)]$) and actual number of bugs discovered (D) overtime. We have doubled the number of fuzzing runs in order to observe how two models predict.

the fuzzing cost, which might include computing resource consumption, delayed product release, etc. Next, the company can generate the accumulated utility curve and the accumulated cost curve based on the curve of expected bug discovery ($E[D(t)]$) proposed in this section. At the point when the utility of fuzzing is below the cost, the fuzzing campaign should be terminated.

6.2 The Order of Bug Discovery

The diminishing of return discussed in Section 6.1 might appear to be a good thing for security. If there is a strong order of bug discovery, then bugs with larger discovery probability will almost always be eliminated first. Thus, as long as the software company invests more resources than other parties, including black hats [27] and white hats [32], in vulnerability discovery, these other parties will not likely to find new vulnerabilities.

However, in reality, we see that many vulnerabilities of famous software have been discovered by outside parties, many of whom are just individuals [1, 13, 25, 32]. There are multiple reasons to explain this. In this work, we propose one explanation based on the power law hypothesis. The basic idea is that the order of bug discovery is weak in the long-tail part of the distribution.

To further explain this, we first define *the order of bug discovery*. At the end of a fuzzing campaign, the expected sequence of rank id (S) of discovered k bugs is $2, 3 \ldots k$, because a higher ranked bug has higher discovery probability, and thus is expected to be discovered earlier. However, due to the randomness, the actual id sequence ($\hat{S}$) would be different from the expected sequence S. We can calculate the edit distance $D(S, \hat{S})$ between these two sequences, and define the order of bug discovery as:

$$order(\hat{S}) = k - D(S, \hat{S}) \tag{6}$$

Intuitively, the bug discovery is strongly/weakly ordered if the distance between S and $\hat{S}$ is small/large. However, since we do not know the true rank id of bugs discovered, we cannot calculate the order of empirical sequences directly. Instead, we run simulation to generated 5 sequences in Table 4. We see bugs discovered in the beginning are more ordered, and tend to be rediscovered in other sequences.

Seq 1:	**19**	3	2	9	4	5	12	14	6	**84**	10	7	**85**	**95**	**24**
Seq 2:	2	3	7	4	5	**17**	10	13	**40**	8	6	**49**	12	11	9
Seq 3:	2	4	5	**28**	3	6	7	**18**	9	12	13	20	11	10	**21**
Seq 4:	2	5	6	3	4	9	**15**	12	**99**	10	8	**46**	7	**225**	20
Seq 5:	3	2	4	7	8	5	**27**	10	11	6	9	**23**	**82**	14	12

Table 4: Simulated bug discovery sequence based on the ffmpeg case ($\alpha = 2.21$). Bug ids in the bold font are unique to that sequence.

We can use the stochastic model to explain this. The probability that the next new discovery is bug i (assuming $i \in \mathcal{L}(t)$) is:

$$P(\text{bug } i \text{ is the next one after time } t) = \frac{\lambda_i}{(\sum_{j \in \mathcal{L}(t)} \lambda_j)} \propto \lambda_i \propto i^{-\alpha} \tag{7}$$

For bug i and bug $i+1$ (assuming $i+1 \in \mathcal{L}(t)$), we have:

$$P_i - P_{i+1} \propto i^{-\alpha} - (i+1)^{-\alpha} \tag{8}$$

which decreases to 0 as $i \to \infty$. This means that when i is small (the fuzzing process is in the "head part" of the distribution), a bug with higher discovery probability is much more likely to be discovered first, and the fuzzing process has a stronger order. However, as i increases and the fuzzing process enters the long-tail, which vulnerability will come next is harder to predict. In addition, a smaller α will make the fuzzing outcome less ordered, while a larger α makes the process more ordered.

To understand its implication, we consider a "fuzzing competition" between a software company and a black hat. Both sides run fuzzing and try to find as many bugs as possible. We assume that the software company has a resource advantage A over the black hat. That is, while the black hat can conduct a fuzzing campaign

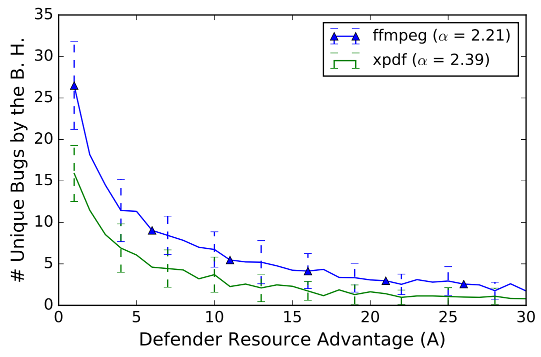

Fig. 4: Simulated number of unique bugs discovered in two programs by the black hat under different resource advantage A of the software company.

with t runs, the company can do At runs, by having a larger fuzzing server farm. We want to know how many unique bugs can the black hat find.

We simulate 10,000 fuzzing runs for the black hat, and simulate $10,000 \times A$ runs for the software company. The two curves in Figure 4 show the number of unique bugs found by the black hat for two programs. We observe that although in the beginning, the software company can quickly reduce the bug pool of the black hat by investing more resources, the return of investment quickly diminishes as A further grows. When the software company has 30 times more fuzzing resources, the black hat is still able to find 2 unique bugs for ffmpeg and 1 unique bug for xpdf on average. Intuitively, it means that when the fuzzing enters the long-tail, the outcome is more random, so the company is less capable of interfering the black hat's outcome. This partly explains why in the reality, outside parties such as black hats and white hats are able to find security holes, despite software companies have already spent significant effort in software security . From Figure 4, we can also see that when α is smaller, more unique bugs can be found by the black hat.

In summary, the power law hypothesis favors attackers, since they are able to find vulnerabilities even if the defender has much more resources. In addition, there is an asymmetry between attackers and defenders: the attackers only need to find a few exploitable bugs to succeed, while the defenders have to patch all holes. On the other hand, this result also encourages software companies to collaborate with outside benign white hats, through vulnerability disclosure and bug bounty programs [32]. We will discuss this more in Section 7.2. But before that, we need to ask one more question: are these unique bugs discovered by by the black hat exploitable?

6.3 Exploitability

Table 5 shows the distribution of bug exploitability in the data. We see that a significant portion of the bugs are either exploitable or probably exploitable.

Program	Exp.	Prob. Exp.	Prob. Not Exp.	Not Exp.	Unknown	Corr.	p-value
xpdf	27%	32%	22%	0	19%	**-0.35**	**0.06**
mupdf	24%	0	64%	0	12%	-0.21	0.35
convert	33%	3%	7%	0	57%	-0.05	0.68
ffmpeg	8%	17%	29%	0	46%	-0.16	0.25
autotrace	39%	4%	4%	0	52%	0.02	0.95
jpegtran	79%	6%	0	0	15%	-0.03	0.87

Table 5: Percentages of exploitability, and correlation between bug discovery probability (log) and exploitability. A correlation is significant if the p-value is less than 0.1.

Then, we further ask the question: is there any correlation between discovery probability and exportability? If there is a positive correlation, then it means that harder to be discovered bugs are harder to be exploited, which favors the software company side. To answer this question, we calculate the Pearson correlation between the logarithm of discovery probability, and the exploitability which is mapped to a 1-4 scale, with 1 meaning not exploitable and 4 means exploitable. We exclude bugs of unknown exploitability. The result is shown in Table 5. We see that although 5 out of 6 programs have a weak negative correlation (i.e., harder to be discovered bugs are easier to be exploited), there is only one that is statistically significant (xpdf). We thus propose the following hypothesis:

Hypothesis 2 *Bug discovery probability and exploitability do not have a strong correlation.*

Hypothesis 2 has several implications. First, it indicates that the next bug to be found could be exploitable, no matter how many runs have been conducted before. This gives an additional advantage for black hats, who not only are likely to find unique bugs, but are also able to find exploitable ones. Second, by assuming the independence between discovery probability and exploitability, one can predict the exploitability of the next bug based on the empirical exploitability distribution in Table 5. For example, in the case of xpdf and $A = 30$ in Figure 4, we can predict that the 1 unique bug discovered by the black hat has roughly 25% probability of being exploitable. By combining the vulnerability discovery models and the exploitability distribution, the software company can thus better forecast potential attacks and allocate defense resources accordingly.

7 Discussion and Future Work

7.1 Apply Our Analysis to Larger Datasets

Although our dataset includes most of the programs studied in previous work [7, 14, 28, 31], it is still not enough to fully test the hypotheses we proposed. Therefore, an important future work is to increase the scope of analysis to other programs, other platforms (e.g., Microsoft Windows and Mac OS), and other fuzzing frameworks [20]. It would also be helpful to run the fuzzing campaign for much longer time.

Another important direction is to apply our analysis to different fuzzing configurations, which include the selection of fuzzing ratio, seeds, etc. It is possible that the same bug's discovery probability might be significantly different in different configurations. This diversity gives an additional explanation to why other parties are likely to find unique bugs, in addition to our discussion in Section 6.2. That is, different parties tend to have different configurations, and thus the discovery probability distribution is distinct to each of them. However, although the discovery probability of a bug might be different under different fuzzing configurations, we hypothesize that the discovery probability distribution will still be a power law distribution:

Hypothesis 3 *The bug discovery probability under different fuzzing configurations follow power law distributions .*

7.2 Generalization to Other Vulnerability Discovery Approaches

We choose to study black-box mutational fuzzing first because it is probably the simplest vulnerability discovery method. However, black-box mutational fuzzing is just one method in the vulnerability discovery toolbox. Other methods include code review, static analysis, symbolic execution, dynamic analysis, etc. We hypothesize that these approaches might resemble fuzzing, and thus the bug discovery "easiness", a generalization of the discovery probability, could also follow the power law distribution. Collecting empirical data from these approaches and applying similar analysis would be an interesting future work.

Some other vulnerability discovery paradigms also share similarities with black box mutational fuzzing. For example, many companies today collaborate with a large number of outside security researchers (or white hats) through vulnerability disclosure and bug bounty programs [32]. Actually, our discussion in Section 6.2 provides one explanation of why such collaboration is necessary. In addition, these white hats, with diverse background and skill levels, will often test different parts of the system, or using various testing payload. This is similar to the seed mutation in a black-box mutational fuzzing, although the distribution of inputs might be more complex than random bit flipping. Therefore, we could possibly generalize the proposed models to understand and analyze data from these bug bounty programs.

8 Conclusion

Understanding the process of vulnerability discovery and why software security is hard has important practical implications. In this work, we have collected empirical data of black-box mutational fuzzing. We show that the fuzzing process can be modeled as a non-homogeneous Poisson process with the rates of individual bugs following a power law distribution. We then show how to calculate the expected outcome of a fuzzing campaign. We further show that once the vulnerability discovery enters the long-tail, there will be significant diminishing returns, and less order in the bug arrival. These effects pose challenge for the software companies that try to eliminate vulnerabilities before the black hats, and call for collaboration with white hats. Finally, we show that the model can potentially be extended to other vulnerability discovery mechanisms, such as bug bounty programs, that have diversity and randomness.

Acknowledgment

We sincerely thank our shepherd and the anonymous reviewers for their valuable comments and suggestions on early versions of this paper. This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CCF-1320605, NSF CNS-1422594, NSF CNS-1505664, ARO W911NF-15-1-0576, and NIETP CAE Cybersecurity Grant.

References

1. A. Algarni and Y. Malaiya. Software vulnerability markets: Discoverers and buyers. *International Journal of Computer, Information Science and Engineering*, 8(3):71–81, 2014.
2. L. Allodi. The heavy tails of vulnerability exploitation. In *Engineering Secure Software and Systems*, pages 133–148. Springer, 2015.
3. J. Alstott, E. Bullmore, and D. Plenz. powerlaw: a python package for analysis of heavy-tailed distributions. *PLoS ONE*, 2014.
4. P. Bishop and R. Bloomfield. A conservative theory for long-term reliability-growth prediction [of software]. *Reliability, IEEE Transactions on*, 45(4):550–560, 1996.
5. R. Böhme and G. Schwartz. Modeling cyber-insurance: Towards a unifying framework. In *The Workshop on the Economics of Information Security (WEIS)*, 2010.
6. R. M. Brady, R. Anderson, and R. C. Ball. *Murphy's law, the fitness of evolving species, and the limits of software reliability.* Number 471. University of Cambridge, Computer Laboratory, 1999.
7. S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *36th IEEE Symposium on Security and Privacy*, 2015.
8. B. Edwards, S. Hofmeyr, and S. Forrest. Hype and heavy tails: A closer look at data breaches. In *The Workshop on the Economics of Information Security (WEIS)*, 2015.
9. C. Evans, M. Moore, and T. Ormandy. Fuzzing at scale. Google Online Security Blog.
10. N. Fenton and J. Bieman. *Software metrics: a rigorous and practical approach.* CRC Press, 2014.
11. J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68. Seattle, 2000.
12. P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *The Network and Distributed System Security Symposium*, volume 8, pages 151–166, 2008.

13. M. Hafiz and M. Fang. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering*, pages 1–40, 2015.
14. A. D. Householder and J. M. Foote. Probability-based parameter selection for black-box fuzz testing. In *CERT*, 2012.
15. W. Jackson. Has secure software development reached its limits? GCN.
16. B. Johnson, A. Laszka, and J. Grossklags. Games of timing for security in dynamic environments. In *6th Conference on Decision and Game Theory for Security (GameSec)*, 2015.
17. M. Jurczyk and G. Coldwind. Ffmpeg and a thousand fixes. Google Online Security Blog.
18. A. Laszka and J. Grossklags. Should cyber-insurance providers invest in software security? In *20th European Symposium on Research in Computer Security*, 2015.
19. T. Maillart and D. Sornette. Heavy-tailed distribution of cyber-risks. *The European Physical Journal B*, 75(3):357–364, 2010.
20. R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: The state of the art. Technical report, DTIC Document, 2012.
21. 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.
22. C. Miller. Babysitting an army of monkeys. In *CanSecWest*, 2010.
23. D. R. Miller. Exponential order statistic models of software reliability growth. *IEEE Transactions on Software Engineering*, (1):12–24, 1986.
24. D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, 2009.
25. R. Naraine. Teenager hacks google chrome with three 0day vulnerabilities. ZDNet.
26. M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.
27. J. Radianti. Eliciting information on the vulnerability black market from interviews. In *Proc. SECURWARE*, pages 154–159, 2010.
28. A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 861–875, 2014.
29. R. Rue and S. L. Pfleeger. Making the best use of cybersecurity economic models. *IEEE Security and Privacy*, (4):52–60, 2009.
30. T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.
31. M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *ACM conference on Computer and communications security*, 2013.
32. M. Zhao, J. Grossklags, and P. Liu. An empirical study of web vulnerability discovery ecosystems. In *ACM conference on Computer and communications security*, 2015.