Program-Adaptive Mutational Fuzzing

Sang Kil Cha, Maverick Woo, and David Brumley
Carnegie Mellon University
Pittsburgh, PA
{sangkilc, pooh, dbrumley}@cmu.edu

***Abstract*—We present the design of an algorithm to maximize the number of bugs found for black-box mutational fuzzing given a program and a seed input. The major intuition is to leverage white-box symbolic analysis on an execution trace for a given program-seed pair to detect dependencies among the bit positions of an input, and then use this dependency relation to compute a probabilistically optimal mutation ratio for this program-seed pair. Our result is promising: we found an average of 37.2% more bugs than three previous fuzzers over 8 applications using the same amount of fuzzing time.**

I. Introduction

Mutational fuzzing [47] (a.k.a. black-box mutational fuzzing) is one of the most effective testing methodologies in finding security bugs and vulnerabilities in Commercial Off-The-Shelf (COTS) software. It has been a huge success in practical security testing and it has been widely used by major software companies such as Adobe [51] and Google [49] for quality assurance purposes [48].

The effectiveness of fuzzing largely depends on *fuzz configuration* [53], which is the set of parameters for running a fuzzer. Recent studies [39, 44], for example, showed that the number of bugs found for a single program given the same computing resources may vary significantly depending on the seed files used. The key challenge is how to find a combination of fuzzing parameters that maximizes the number of bugs found given a limited resource.

The state of the art for maximizing the fuzzing outcome is to search over the parameter space of fuzzing [26, 53], which is called *Fuzz Configuration Scheduling* (FCS). That is, fuzzers explore possible combinations of parameters, and exploit the partial information obtained from the exploration to maximize the fuzzing outcome. This is the classic "exploration vs. exploitation" trade-off, and is often stated as a Multi-Armed Bandit (MAB) problem [5] as noted in [53].

Unfortunately, FCS is challenging when the parameter space is large because there are too many possible parameter combinations to consider. For example, the *mutation ratio*—the rate between the number of bits to modify and the number of total bits of a seed, which is used to confine the Hamming distance from the seed to generated test cases—is a *continuous* parameter, and thus it can have arbitrary many values. The key question is how to discretize the continuous parameter (mutation ratio); in what granularity should we discretize the parameter? This problem has been left open in [53].

Current mutational fuzzers circumvent this problem by selecting just a single mutation ratio, or by using random ratios from a range. However, there is a fundamental challenge in the existing methods: they all involve either manual or non-adaptive parameter selection. First, an analyst has to choose the fuzzing parameters based on their expertise. For example, zzuf [30] runs with either a single or a range of mutation ratios, but the analyst must specify those parameters. Second, if not manual, the parameters are derived non-adaptively regardless of the program under test. BFF [26], for instance, splits a set of all possible nonzero mutation ratios into a predefined set of intervals, and performs scheduling (FCS) over the intervals. FuzzSim [53] and zzuf uses a predefined mutation ratio if a user does not specify a value. AFL-fuzz (American Fuzzy Lop) [58] also employs several bit-flipping mutation strategies that only mutate a fixed number of bits, e.g., flip only a single random bit, regardless of the program under test.

The key question motivating our research is—can adaptive mutation ratio selection help in maximizing the bug finding rate of mutational fuzzing for a given program and a time limit? If so, can we automatically find such a mutation ratio? This is further inspired by a preliminary study we performed: we fuzzed 8 applications for one hour with each of 1,000 distinct mutation ratios from 0.001 to 1.000. We found that the number of bugs found varies significantly over different mutation ratios. Also the distribution of the number of bugs in 1,000 ratios was indeed biased towards several mutation ratios, and the best mutation ratio was different for each distinct program-seed pair (§VI-B). This result provides evidence that adaptive mutation ratio selection can benefit fuzzing efficiency. Hence, the question is how to compute these mutation ratios.

In this paper, we introduce a system called SymFuzz, which determines an optimal mutation ratio from a given program-seed pair based on the probability of finding crashes. SymFuzz augments black-box mutational fuzzing by leveraging a white-box technique, which analyzes a program execution to realize an effective mutation ratio for fuzzing. It then performs traditional black-box mutational fuzzing with the derived mutation ratio. Although the white-box technique often entails heavy cost analysis, it is required only once per program-seed pair as a preprocessing step.

The primary intuition of our work is that a desirable mutation ratio that maximizes the fuzzing efficiency can be deduced from the dependence relations between the input bits of a seed for a program. Suppose we are given a program and a 96-bit seed that consists of a 32-bit magic number followed by two consecutive 32-bit integer fields. We also assume that the magic number is 42424242_{16} and two integer values are zero. The program crashes when an input value satisfies the following two conditions: (1) the magic number remains 42424242_{16}; and (2) the third field is negative integer. To trigger the crash, one needs to flip the most significant bit in the third field, but never touch the bits in the first field of the seed. The value of the second field does not affect the crash. In this case, there exists a dependence relation between the first and the third field: the third field depends on the first field. That is, even though we have a negative value for the third field, we will never be able to trigger the crash if we flip any of the bits in the first field. In this paper, we show that the dependence between the input bits indeed decides the best mutation ratio for this crash, which is about 0.031.

Throughout this paper, we use mutational fuzzing to refer to a software testing technique that consists of the following two steps. First, it generates test cases by flipping input bits of a seed. Second, it evaluates a program under test using the generated test cases to determine whether they crash the program. We assume we know how to run the program under test and what types of input seeds it takes. These pieces of information are commonly available in security testing. We also expect the mutation ratio to specify the number of bit positions to flip in a seed input. In other words, a fuzzer chooses bit positions at random without replacement when mutating a seed. Finally, we consider only three fuzzing parameters including a program, a seed, and a mutation ratio.

We evaluated our system on 8 Linux utilities on Debian 7.4 Wheezy (the version of May 2014). SymFuzz found 39.5% more bugs than BFF [26]—the state-of-the-art scheduling-based mutational fuzzer—on 8 applications. In total, we found 114 previously unknown distinct crashes using a stack hash that we developed to reduce the false-positive rate of an existing stack hash (§V-D).

Overall, this paper makes the following contributions:

1) We devise a mathematical framework to formalize mutational fuzzing, and show how to model the failure rate of mutational fuzzing with respect to the dependence relation between input bits.
2) We introduce a novel method, called mutation ratio optimization, to select a mutation ratio that maximizes the probability of finding bugs.
3) We combine both black- and white-box techniques to maximize the effectiveness of fuzzing. To the best of our knowledge, we are the first in leveraging white-box analysis in optimizing parameters for black-box fuzzing.
4) We design a mutational fuzzing framework, SymFuzz, that implements our technique. This is the first mutational fuzzer that admits an easy mathematical analysis.
5) We make our data and source code public in support of open science.

The rest of this paper is organized as follows. §II sets out the definitions and assumptions needed to our mathematical model. §III then discusses how we obtain a list of candidate mutation ratios. Next, §IV introduces our technique called input-bit dependence inference. We then discuss detailed design of our system in §V, and show evaluation of our system in §VI. Finally, we present related works in §VIII, and summarize our conclusion in §IX.

II. Definition

A. Notation

We let an input, a.k.a. test case, be a bit string. In our model, each input has a fixed length of N bits. An input space $\mathcal{I}_N$ denotes the universe of all possible inputs of size N bits. Therefore, the cardinality of the input space $|\mathcal{I}_N|$ is 2^N. There are inputs in the input space that trigger one or more program bugs, which we call *buggy inputs*. We use a subscript to specify a bit position in an input. For example, s_1 means the first bit of the input s. We denote the Hamming distance—the number of bit differences in two input strings—between input i and j in the input space by $\delta(i, j)$.

Definition 1 (*K-Neighbors*)**.** A K-neighbor of an input i of length N is an input whose Hamming distance from i is K. We denote the set of all K-neighbors of i by $\mathcal{N}_K(i)$:

$$\mathcal{N}_K(i) = \{j \in \{0,1\}^N \mid \delta(i,j) = K\}.$$

Given the above definition, observe that two sets of K-neighbors of the same input with a different value of K are disjoint from each other:

$$\forall i, 0 \le A, B \le N : \mathcal{N}_A(i) \cap \mathcal{N}_B(i) = \varnothing \iff A \ne B.$$

We let μ be a function that takes as input a test case and a set of bit positions, and returns a mutated test case where every specified bit is flipped (exclusive-or-ed with 1) from the given test case. For example, $\mu(s, \{3,4\})$ is an input where both the third and the fourth bit of s are flipped, and $\delta(\mu(s, \{3,4\}), s) = 2$.

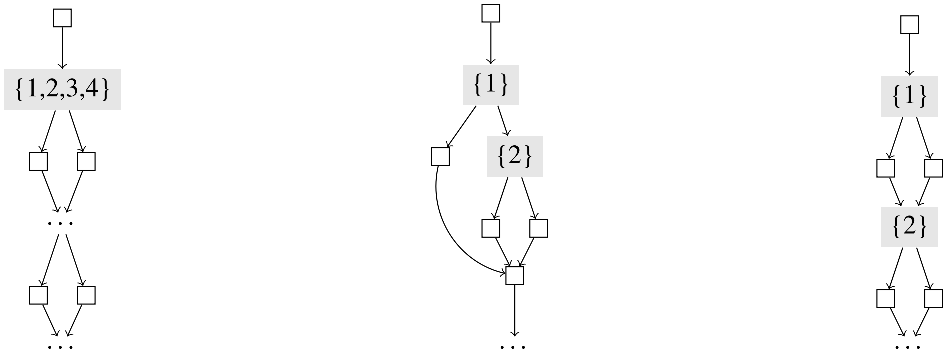
We represent an *execution* as a sequence of instructions. Given a program p and a seed input s to the program, the execution of p using s as an input is $\sigma_p(s)$. An evaluation function ϵ takes in a program execution and outputs either a crash identifier when the program crashes, or $\perp$ if otherwise. We assume that a crash identifier uniquely determines the kind of crashes; see §V-D for discussion on our crash triage technique called safe stack hash. For instance, if a program p crashes when we execute it with a test case i, then $\epsilon(\sigma_p(i)) = c$, where c is the crash identifier.

B. Input-Bit Dependence

We define dependence between input bits using control dependence [2]. Informally, when a node u in a Control-Flow Graph (CFG) decides if another node v is executed or not, then we say v is control-dependent on u. We extend the notion of control dependence to define the relationship between input bits as follows.

Definition 2 (*Input-Bit Dependence*)**.** Given an execution $\sigma_p(s)$, consider two bit positions x and y of s. We say that the bit s_x is *dependent* on s_y, denoted by $s_x \xrightarrow{dep(p)} s_y$, if either of the following conditions holds: (1) there is a conditional branch that reads both s_x and s_y; (2) there are two conditional branches c_1 and c_2 that read s_x and s_y respectively, and c_1 is control-dependent on c_2.

Figure 1 demonstrates input-bit dependence for three



(a) Bits 1, 2, 3, 4 are dependent each other.

(b) Bit 2 is dependent on bit 1.

(c) Bits 1 and 2 are independent.

Fig. 1: Input-bit dependence. Each gray box represents a conditional branch that is controlled by a set of input bit positions.

different cases. Figure 1a and Figure 1b show examples that satisfy the first and the second condition of input-bit dependence respectively. In Figure 1a, every bit involved in the same condition is dependent on each other due to the first condition of Definition 2: $s_1 \xrightarrow{dep(p)} s_1, s_1 \xrightarrow{dep(p)} s_2, s_1 \xrightarrow{dep(p)} s_3, s_1 \xrightarrow{dep(p)} s_4, s_2 \xrightarrow{dep(p)} s_1, \cdots, s_4 \xrightarrow{dep(p)} s_3, s_4 \xrightarrow{dep(p)} s_4$. In Figure 1b, $s_2 \xrightarrow{dep(p)} s_1$. Finally, Figure 1c presents a case where two input bits are not dependent on each other.

As an example, let us consider the following C program.

```
1   char x = input[0]; char y = input[1];
2   if ( x > 42 ) {
3     if ( y > 42 ) {
4       ... /* omitted */
```

Given a program execution that exercises Line 4, the second byte (bits 9 to 16) of the input is dependent on the first byte (bits 1 to 8), all the bits in the first byte are dependent upon each other, and all the bits in the second byte are dependent upon each other.

Based on the definition of the input-bit dependence, we can compute the set of dependency bits for each bit in a seed; we use the term "dependency" in the same vein to the term "library dependencies". We call such a set as a *dependency bitset*, and denote it with a function $\uparrow$. The upward arrow is intended to reflect the direction of the dependence relation in a CFG. For instance, $\uparrow^p_s(\{3,4\})$ is the set of bits that are depended on either by s_3 or by s_4 in the execution $\sigma_p(s)$.

Definition 3 (*Dependency Bitset*)**.** Given a set of bit positions X of a seed s for a program p, the dependency bitset of X is defined as

$$\uparrow^p_s(X) = \left\{ y \;\middle|\; x \in X, s_x \xrightarrow{dep(p)} s_y \right\}.$$

Intuitively, we have defined input-bit dependence to reveal the approximate syntactic structure of an input. Most input structures consist of a series of input fields. For instance, a PNG file has a series of data chunks each of which consists of four input fields. Intuitively, every bit in an input field should depend on each other, because all the bits together decide the control-flow of the program. Indeed, a notion similar to input-bit dependence has been used in recovering the format of an input from a given program execution [32]. More precisely, bits in a dependency bitset can be a superset of a bits in an input field: input-bit dependence can involve multiple input fields. In this paper, we use the input-bit dependence to infer the optimal mutation ratio for mutational fuzzing (§III-B).

C. Mutational Fuzzing

Mutational fuzzing is a software testing technique where test cases are derived from a *seed*—typically a well-formed input—by partially mutating the seed. The number of bit mutations is determined by a parameter called the *mutation ratio* r, which is the rate between the number of bit positions to flip and the total number of bit positions in a seed. We assume that mutational fuzzing selects a set of bit positions at random *without* replacement, and hence the Hamming distance between an N-bit seed and a mutated input of it is always $\lceil N \cdot r \rceil$, i.e., r is always chosen such that $N \cdot r$ is an integer. This is different from all current fuzzers that use sampling *with* replacement. See §V-C for more discussion on mutational fuzzing and its implementation.

We view mutational fuzzing as a more targeted version of random testing, which generates only test cases from a subspace of the input space. The key assumption is that the distribution of buggy inputs are often biased towards a subspace where the inputs are not far from a seed in terms of the Hamming distance. For example, a crashing test case for an MP3 player is likely to be a nearly-valid MP3 file instead of being a random string. If a random string is given, the program will likely reject the input before it reaches a buggy code. More formally, we define mutational fuzzing as follows.

Definition 4 (*Mutational Fuzzing*)**.** Given a seed input s of size N bits and a mutation ratio r, mutational fuzzing generates and evaluates a set of test cases chosen uniformly at random from $\mathcal{N}_K(s)$, where $K = \lceil N \cdot r \rceil$.

We model each fuzz trial as a probabilistic experiment that randomly selects a test case from an input space $\mathcal{I}_N$. Let $\mathcal{B}^p_N$ be the set of N-bit buggy inputs for program p. Given a sample space $\mathcal{I}_N$, the probability of a randomly chosen input being in $\mathcal{B}_N^p$ is the failure rate θ [13], which is defined as

$$\theta = \frac{|\mathcal{B}_N^p|}{|\mathcal{I}_N|} = \frac{|\mathcal{B}_N^p|}{2^N}.$$

D. Failure Rate based on Mutation Ratio

In this paper, we are interested in finding a mutation ratio r that maximizes the failure rate of mutational fuzzing. Therefore, we need to represent the failure rate in terms of r. To do so, we first categorize bit positions in a seed into several kinds, and approximate the failure rate in terms of mutation ratio and input-bit dependence.

Given a program p and a seed s, suppose the program crashes when it is executed on a mutated s, i.e., there is an input among the K-neighbors of s that triggers the crash. Specifically, there is a set of bits in s that, when flipped, generates a buggy input for p. We call such a set a *buggy bitset* of s.

Definition 5 (*Buggy Bitset*)**.** Given a program p and an N-bit seed s, a buggy bitset is a set of bit positions $B \subseteq \{1, 2, \ldots, N\}$ where $\epsilon(\sigma_p(\mu(s, B))) \neq \perp$.

Some of the bits in a buggy bitset may not need to be flipped to generate an input that triggers the bug. Among all subsets of a buggy bitset, there exists a combination of bits with a minimum cardinality while still producing a buggy input for the same bug. We call such a subset a *minimum buggy bitset*.[1] Notice that there may be multiple minimum buggy bitsets with the same size. Suppose there is an 8-bit seed, and flipping both the first and second bits of the seed leads a program p to crash. The buggy bitset is therefore $\{1, 2\}$, and $\epsilon(\sigma_p(\mu(s, \{1, 2\}))) \neq \perp$. Now, suppose flipping only the second bit of the seed produces a buggy input that leads to the same crash, i.e., $\epsilon(\sigma_p(\mu(s, \{1, 2\}))) = \epsilon(\sigma_p(\mu(s, \{2\})))$. Since we assume that a seed does not produce a program crash by itself, a minimum buggy bitset is $\{2\}$.

Definition 6 (*Minimum Buggy Bitset*)**.** Given a buggy bitset B for a program p and a seed s, a minimum buggy bitset B' of B is an element of the set

$$\underset{\{B^* \subseteq B \mid \epsilon(\sigma_p(\mu(s,B^*))) = \epsilon(\sigma_p(\mu(s,B)))\}}{\operatorname{argmin}} |B^*| .$$

A minimum buggy bitset B' includes a set of bits that must be flipped to generate a buggy input. Any bit position other than B', i.e., any element of $\{1, 2, \ldots, N\} \setminus B'$, either (1) does not affect the crash regardless of its values; or (2) thwarts the crash when it is flipped. To find a set of bits that must *not* be flipped for triggering the crash, we overapproximate a set of bits that changes the program execution with respect to the bits in B', which is a dependency bitset of B' by definition. If any of the bits in $\uparrow_s^p(B')$ are flipped, it will change the execution of the program. Since all the bits in B' must be flipped, the other bits in $(\uparrow_s^p(B') \setminus B')$ must *not* be flipped to maintain the same execution path for the crash.[2]

The above argument can be intuitively explained by an example. A bug is typically triggered when one or more input fields have specific values, e.g., one needs to set an integer field to be greater than the size of a program buffer to trigger a buffer overflow. However, even though the integer field has a value greater than the buffer size, the program might take an execution path that does not even read the values. This happens when the program checks the value of another input field f before it reaches the buffer overflow, and jumps to another execution path. Therefore, the integer input field is dependent on f. This is the key intuition of approximating the failure rate of mutational fuzzing in terms of the input-bit dependence.

We now compute a failure rate for each minimum buggy bitset. For simplicity, let b be the cardinality of a minimum buggy bitset ($b = |B'|$), and let d be the cardinality of the dependency bitset of B' ($d = |\uparrow_s^p(B')|$). We also let r be the mutation ratio. The failure rate of mutational fuzzing for B' follows a multivariate hypergeometric distribution [7], where the population size is N and the number of draws is $(N \times r)$. Therefore, the failure rate of a minimum buggy bitset that has b elements is:

$$\theta_b = \frac{\binom{b}{b}\binom{N-d}{N\cdot r-b}}{\binom{N}{N\cdot r}} = \frac{\binom{N-d}{N\cdot r-b}}{\binom{N}{N\cdot r}}, \quad \text{when } N \cdot r \geq b \tag{1}$$

This formula can be explained as follows. Given an N-bit seed, the total number of possible inputs that mutational fuzzing can generate is $\binom{N}{N\cdot r}$. To generate a buggy input from a seed, we need to flip all the bits in the minimum buggy bitset (this is the $\binom{b}{b}$ term), while not flipping the bits in the dependency bitset of B' (this is the $\binom{N-d}{N\cdot r-b}$ term). Since $\binom{b}{b} = 1$, the term can be eliminated.

The failure rate is only meaningful when the number of flipped bits is not less than the size of B', i.e., $N \cdot r \geq b$. When $N \cdot r < b$, we simply cannot flip every bit in B'. By the definition of the minimum buggy bitset (Definition 6), one needs to flip all the bits in B' in order to generate a buggy input. Therefore, the failure rate is effectively 0 in this case.

E. Mutation Ratio Optimization Challenge

Now that we have a formal definition of mutational fuzzing and its failure rate, we address *mutation ratio optimization challenge* as follows.

Definition 7 (*Mutation Ratio Optimization Challenge*)**.** Given a program p and an N-bit seed s, consider a crash that is identified by a minimum buggy bitset B', and let $b = |B'|$. The mutation ratio optimization challenge is to derive a mutation ratio r that maximizes the failure rate θ_b of p.

Notice the cardinality of a minimum buggy bitset (b) is not known unless we have found the corresponding bug. Moreover, we may have multiple optimal mutation ratios for different values of b. Therefore, several questions remain: How do we solve the mutation ratio optimization challenge? How do we

[1] Deriving a minimum buggy bitset is often called bug minimization [25].

[2] The dependency bitset of B' is an over-approximation of the immutable bit positions for the crash, because flipping some bits in $(\uparrow_s^p(B') \setminus B')$ may still trigger the same crash.

compute the cardinality of the dependency bitsets (d) for a given program-seed pair? We address these questions in the following sections.

III. Mutation Ratio Optimization

In this section, we introduce a systematic way of deciding a set of mutation ratios for a given program and a seed, which we call *mutation ratio optimization*. Our technique automatically adapts to a given program-seed pair, and it enables efficient bug finding for mutational fuzzing.

A. Solving for an Optimal Mutation Ratio

Recall in §II-D we described the failure rate θ_b of mutational fuzzing with respect to three variables: the bit size of a seed (N), the cardinality of a minimum buggy bitset (b), and the cardinality of a dependency bitset of the minimum buggy bitset (d). One of the primary challenges is to find a mutation ratio $0 < r \leq 1$ that maximizes θ_b. When $d = b$, i.e., $B' = \upharpoonleft_s^p(B')$, it is trivial to show that the maximum failure rate is achieved with $r = 1$: we simply let $d = b$ and $r = 1$ from Equation 1, and then the failure rate θ_b becomes always 1 regardless of the value of b. When $d = N$, there is no bit position to flip other than the ones in $\upharpoonleft_s^p(B')$, and, as a result, the only way to trigger the crash is to flip exactly b bit positions. That is, the optimal mutation ratio is b/N. When $b < d < N$, we solve the mutation ratio optimization problem by modeling it as a classic nonlinear programming problem (NLP) [6] as follows.

For N, b, and d, find r to

$$\begin{array}{ll} \textbf{maximize} & \theta_b = \dfrac{\binom{N-d}{N\cdot r-b}}{\binom{N}{N\cdot r}} \\ \textbf{subject to} & (0 < r \leq 1) \\ & \wedge \quad (b < d < N) \\ & \wedge \quad (b \leq N \cdot r \leq N - d + b). \end{array}$$

The first constraint of the NLP is from the definition of mutation ratio: mutation ratio must be between zero and one. The second constraint ($b < d < N$) is to restrict the range of the d value. When $d = b$, the optimal mutation ratio is 1, and when $d = N$, the optimal mutation ratio becomes b/N as we discussed above. The third constraint ($b \leq N \cdot r \leq N - d$) is due to our problem definition: (1) we should flip more than the cardinality of a minimum buggy bitset in order to generate an input that trigger the bug ($b \leq N \cdot r$); (2) we should not flip any bits in ($\upharpoonleft_s^p(B') \setminus B'$), hence the maximum number of bit flips is $(N - d + b)$.

We now solve the above NLP to obtain an optimal mutation ratio for a given minimum buggy bitset. The solution to it is the optimal mutation r with respect to b, d and N. See Appendix A for a complete proof.

Theorem 1 (Optimal Mutation Ratio)**.** *Given a minimum buggy bitset B' and the corresponding $\upharpoonleft_s^p(B')$ for a program p and a seed s, let $b = |B'|$ and $d = |\upharpoonleft_s^p(B')|$. The optimal mutation ratio r for finding the bug $\epsilon(\sigma_p(\mu(s, B')))$ is*

$$r = \frac{b \times (N+1)}{d \times N} \textit{ when } N \cdot r > b. \qquad (2)$$

We find an optimal mutation ratio r that maximizes the failure rate as follows when $b < d < N$. First, when $b = N \cdot r$, we compute a failure rate θ_1 by letting $r = b/N$ from Equation (1). Next, we obtain another mutation ratio r_2 for the case of $b < N \cdot r$ using Equation (2). We then compute a failure rate θ_2 for r_2 from Equation (1). Finally, we compare the two failure rates and return an optimal mutation ratio as follows: if θ_1 is greater than θ_2 then the optimal ratio is b/N; otherwise it is r_2. We note, when computing r_2, N is given from the seed, but b and d are unknown. We know neither buggy bitsets nor minimum buggy bitsets prior to fuzzing the program under test: our goal is to pre-compute the optimal mutation ratio before fuzzing. That is, we cannot know the corresponding minimum buggy bitset before hitting a bug. This problem suggests that we must find a way to estimate the value of b and d without the prior knowledge about the buggy bitsets.

B. Estimating r

Suppose there exist M unique crashes that can be produced by mutating an N-bit seed s for a program p. Since each crash can have its own distinct minimum buggy bitsets, the value of b and d may differ depending on the crash, and thus, each crash may have different optimal mutation ratios. Ideally, one may find a set of distinct mutation ratios for all M crashes, but knowing exact b and d for every unique crash is infeasible in practice.

To estimate effective mutation ratios in finding all M crashes, we use the averaged values of b and d. Although buggy bitsets are unknown in advance, we can still compute dependency bitsets of every bit in the seed: we can obtain all possible d values from the given program-seed pair, which will expose the trend of the input-bit dependence of the seed. We then use this information to estimate d. Let $\mathcal{P}(S)$ be the powerset of S. We then denote $\bar{d}_{\text{all}}$ as the average cardinality of every possible dependency bitsets for the program-seed pair:

$$\bar{d}_{\text{all}} = \frac{\sum_{x \in \mathcal{P}(\{1,2,\ldots,N\})} |\upharpoonleft_s^p(x)|}{2^N}.$$

Recall from §II-B, the input-bit dependence indicates the overall input structure for a given program-seed pair: it reveals which chunk of the input bits together affects the control flow of the program. Therefore, the average input-bit dependence $\bar{d}_{\text{all}}$ is an approximate indicator that shows how many bits are dependent on each other for a given program-seed pair. When $\bar{d}_{\text{all}}$ is high, that means many input bits in the seed are dependent on each other, and thus, there are likely to be more input bits that should not be mutated to trigger the crashes. That is, a larger $\bar{d}_{\text{all}}$ corresponds to a smaller r and vice versa. This relationship between $\bar{d}_{\text{all}}$ and r is evident from the above NLP formulation. The optimal mutation ratio ranges from b/N (when $d = N$) to 1.0 (when $d = b$), and as we have smaller d, i.e., less dependence between input bits, we have a higher optimal mutation ratio.

Although $\bar{d}_{\text{all}}$ shows the trend of input-bit dependence, there

Algorithm 1: Computing $\bar{d}$ using adaptive sampling.

```
   input  : A distribution of $b$ values ($\beta$)
   output : $\bar{d}$
1  prevN ← 0
2  prevSum ← 0
3  while $true$ do
4      $b$ ← WeightedRand($N$, $\beta$)
5      $S$ ← RandomK($N$, $b$)
6      newN ← prevN + b
7      newSum ← prevSum + $|\uparrow_s^p(S)|$
8      if prevN $\neq 0 \wedge |$newSum/newN − prevSum/prevN$| < \epsilon$
       then break
9      else prevN ← newN; prevSum ← newSum
10 end
11 return prevSum/prevN                          /* Returns $\bar{d}$ */
```

are two remaining problems in using it. First, just averaging the cardinality of all possible dependency bitsets is not necessarily the best way to represent the trend, because the cardinality of minimum buggy bitsets (b) may be biased towards several values. Second, the number of dependency bitsets to consider is exponential in N, and N is typically not small.

To mitigate both challenges, we incorporate the distribution of b (β) into the average input-bit dependence by using adaptive sampling [50], which also helps in computing an approximate average efficiently. The algorithm is shown in Algorithm 1. First, we select a random cardinality b with the probability associated with each cardinality in β (Line 4, `WeightedRand`). Next, we sample a set of random bit positions S of cardinality b (Line 5, `RandomK`). `RandomK` takes in N and b, and returns b distinct random numbers from the interval $[1, N]$. We then compute $|\uparrow_s^p(S)|$, and use the cardinality to compute a new cardinality sum. Then, we check the difference between the previous and the new mean values to see if it is smaller than a threshold ϵ (Line 8). We repeat the process until the difference is negligible (we use $\epsilon = 10^{-7}$ in our experiment). After breaking out of the while loop, the algorithm returns the final average input-bit dependence denoted as $\bar{d}$.

Since $\bar{d}$ relies on the distribution of b (β), it is important to note how the distribution looks like. We obtained a large scale fuzzing dataset from a previous work, and computed the cardinality of minimum buggy bitsets for each unique crash found in [44]. The average b value was 9 and the standard deviation was 18. This result conforms to the observations from practitioners [25]. See §VI-C for further discussion on how we obtained b values from the dataset.

Now that we have a distribution of b values from a large-scale experiment, we need to estimate r using the averaged d value ($\bar{d}$). Since $\bar{d}$ is the average cardinality of dependency bitsets per each bit in minimum buggy bitsets, we can estimate the cardinality of a dependency bitset for a minimum buggy bitset of cardinality b using $b \times \bar{d}$. By letting $d = \bar{d} \times b$, we can simplify the Equation 2 as follows.

$$r = \frac{b \times (N+1)}{b \times \bar{d} \times N} = \frac{1}{\bar{d}} \cdot \frac{N+1}{N}. \tag{3}$$

The value of b is now included in $\bar{d}$, and we only need to consider the value of $\bar{d}$ to estimate the optimal mutation ratio r. Given the distribution of b in crashes, $\bar{d}$ provides a way to estimate the cardinality of dependency bitsets for the crashes, which, in turn, helps in estimating r.

p	::=	`stmt`*
`exp`	::=	get_input(src) $\mid$ load(`exp`) $\mid$ `var`
		$\mid$ `exp` $\Diamond_b$ `exp` $\mid$ $\Diamond_u$ `exp` $\mid$ v
`stmt`	::=	`var:=` `exp`
	$\mid$	goto `exp`
	$\mid$	if `exp` then goto `exp`$_1$ else goto `exp`$_2$
	$\mid$	call `exp`
	$\mid$	ret `exp`
	$\mid$	store(`exp`,`exp`)
v	::=	⟨unsigned integer, a set of affecting bits⟩
$\Diamond_b$	::=	binary operators
$\Diamond_u$	::=	unary operators

TABLE I: A simple language for IBDI.

IV. Input-Bit Dependence Inference

At a high level, Input-Bit Dependence Inference (IBDI) is a process of computing the input-bit dependences for every bit in a seed from a program execution. We then use these dependence relations to compute $\bar{d}$ as in Algorithm 1. From the perspective of program analysis, IBDI is a symbolic analysis that is more specific than the traditional taint analysis [14, 42, 57], and more abstract than the traditional symbolic execution [8, 27, 29]. Our approach is inspired by several automatic input format recovery approaches including [10, 16, 17, 32], where they share a common theme as us: they use a program execution to reveal the structure of an input. However, our focus is on figuring out the input-bit dependence rather than precise input formats.

A. The Algorithm

Input-Bit Dependence Inference (IBDI) takes as input a program and a seed, and outputs the input-bit dependence for every bit of the seed. Similar to dynamic symbolic execution [11, 22], IBDI runs the program under test both concretely and symbolically. The key difference between IBDI and dynamic symbolic execution is that IBDI operates on a set of dependent bits instead of generating bit-vector-level path formulas, hence it does not rely on SMT solvers [18]. As in dynamic symbolic execution, IBDI introduces symbolic values whenever reading from a user input, e.g., `read` system call. It then symbolically evaluates program statements on a program execution. It also constructs a CFG while symbolically executing the program in order to compute control dependences between variables and the corresponding input bits.

We describe our IBDI algorithm using the formal runtime semantics over a simple language shown in Table I. In our language, a program is a sequence of statements. There are four different jump statements including `goto`, `if-else`, `call`, and `ret`. The first two are regular jump statements: `goto` is an unconditional jump statement, and `if-else` is a conditional jump statement. The last two kinds, `call` and `ret`, are special jump instructions that represent calls and

Context	Meaning
s	a list of program statements
m_c	mapping from an address to concrete value
r_c	mapping from a variable to concrete value
m_a	mapping from an address to abstract value
r_a	mapping from a variable to abstract value
Δ	current dependence predicate
c	current input-dependence stack
l	current delay queue
pc	current program counter
i	current statement

TABLE II: The execution context of our analysis.

returns respectively. Notice, however, we do not allow call/ret instructions to implicitly manipulate call-stacks in our language. For example, call instruction in x86 will be jitted into into stack manipulation statements followed by a `call` statement.

Since we execute a program both concretely and symbolically, expressions in our language evaluate to a value v, which is a tuple of a concrete value and an abstract value. A concrete value is an unsigned integer, and an abstract value is a set of input bits that affects either directly or indirectly the value, which is often called data lineage [32, 33]. We denote data lineage of a variable as a set of bit positions. For example, if a variable x evaluates to $\langle 1, \{2, 3, 4\}\rangle$, it means the variable x has a concrete value of 1, and is also affected by the three other input bits.

We use $\Diamond_b$ to denote binary operators such as addition, subtraction, etc. Similarly, $\Diamond_u$ represents unary operators such as minus. When we evaluate $\Diamond_b$ over abstract values (data lineages), we apply set union between them. For example, when we evaluate a subtraction between $\{1\}$ and $\{1, 2, 3, 4\}$, we obtain $\{1, 2, 3, 4\}$. For $\Diamond_u$, we simply propagate abstract values from a source to a destination.

The execution context of our analysis consists of ten fields as shown in Table II. We store abstract and concrete values for variables in r_a and r_c respectively in a map.[3] Similarly, we store abstract and concrete values of memory addresses in m_a and m_c respectively in a map. To access maps, we use a bracket notation. For example, $r_a[x]$ returns the current abstract value of x, and $r_c[x \leftarrow 1]$ returns a new map, which is equivalent to the previous map except that the value of x is 1. We use $\Downarrow$ to represent evaluation of an expression under a given context. For example, $m_c, r_c, m_a, r_a \vdash e \Downarrow v$ is an evaluation of an expression e to a value v in the context given as 4-tuples (m_c, r_c, m_a, r_a).

We encode the input-bit dependence for every bit in an input using a data structure that we call *dependence predicate* (Δ). The dependence predicate is essentially a map from a bit of an input to a set of bit positions that the bit is dependent on. As we execute the program under test, we update Δ using a `merge` function. For example, suppose Δ has a mapping from the first bit to $\{3, 4\}$. Then, merge($\Delta, \{1, 2\}, \{5, 6\}$) will return a new dependence predicate which contains two mappings: (1) from the first bit to $\{3, 4, 5, 6\}$, and (2) from the second bit to $\{5, 6\}$. Algorithm 2 describes the merge function. Notice, in Line 4 of the algorithm, we compute the relative complement of $\{x\}$ in order to exclude the dependence relations that self-referencing.

Algorithm 2: Dependence predicate update algorithm.

```
1 Function merge($\Delta, X, Y$)
2     $\Delta' \leftarrow \Delta$
3     for $x \in X$ do
4         $\Delta' \leftarrow \Delta'[x \leftarrow (\Delta[x] \cup Y \setminus \{x\})]$
5     end
6     return $\Delta'$
7 end
```

Algorithm 3: Delay queue update algorithm.

```
Function delayedUpdate($\Delta, l$)
    for $\langle X, Y\rangle \in l$ do
        if $\langle X, Y\rangle$ is not memoized then
            $\Delta \leftarrow$ merge($\Delta, X, Y$)
            memoize $\langle X, Y\rangle$
        end
    end
    return $\langle \Delta, [\,]\rangle$
end
```

One may call the merge function for every instruction encountered on the fly. However, we delay the predicate update until we reach a return instruction—thus, update the dependence predicate per control-dependent region [54]—for two reasons. First, it is more cache-efficient to perform updates once in a while. Second, we can employ a heuristic to eliminate unnecessary updates: there are many duplicated updates, thus we can memoize the last updates to speed up the process. To perform delayed update, we employ an additional field in our execution context, which we call a delay queue (l). l stores a tuple of the current data lineage and the current set of dependent bits from the control stack. To add an entry to a delay queue, we use an `add` method.

We maintain an input-bit dependence stack (c) to store a set of bit positions that the current instruction is control-dependent on. The idea is similar to dynamic control-dependence analysis in [54], but input-bit dependence stack maintains a set of control-dependent bits instead of storing control-dependent statements. We use three methods to access the input-dependence stack (IDS): `top()` returns the top element of a stack, `pop()` returns a tuple of the top element of a stack and a new stack without the top element, and `push`(X) returns a new stack which contains an additional element X.

Each element on the stack is a 2-tuple (an address of an instruction that is beyond the scope of the current control dependence, a set of control-dependent bits). In our analysis, control dependence of a conditional branch is valid in two scenarios: (1) until we reach an immediate post-dominator of the conditional branch; (2) until we reach a return instruction. Therefore, we need to update the input-bit dependence stack either when we enter a function (call instructions), or when we encounter a conditional branch. However, to efficiently handle

[3] Variables at the machine-code level are really ***r***egisters.

$$\frac{v_c = \text{a concrete value from } src \quad v_a = \{\text{set of input bit positions}\}}{m_c, r_c, m_a, r_a \vdash \texttt{get\_input(src)} \Downarrow \langle v_c, v_a\rangle}\ \textsc{Input} \qquad \frac{}{m_c, r_c, m_a, r_a \vdash \texttt{var} \Downarrow \langle r_c[\texttt{var}], r_a[\texttt{var}]\rangle}\ \textsc{Var}$$

$$\frac{m_c, r_c, m_a, r_a \vdash e \Downarrow \langle v_c, v_a\rangle \quad v'_c = m_c[v_c] \quad v'_a = m_a[v_c]\Diamond_b v_a}{m_c, r_c, m_a, r_a \vdash \text{load } e \Downarrow \langle v'_c, v'_a\rangle}\ \textsc{Load} \qquad \frac{m_c, r_c, m_a, r_a \vdash e \Downarrow \langle v_c, v_a\rangle \quad v'_c = \Diamond_u v_c}{m_c, r_c, m_a, r_a \vdash \Diamond_u e \Downarrow \langle v'_c, v_a\rangle}\ \textsc{Unary-Op}$$

$$\frac{m_c, r_c, m_a, r_a \vdash e_1 \Downarrow \langle v_{c1}, v_{a1}\rangle \quad m_c, r_c, m_a, r_a \vdash e_2 \Downarrow \langle v_{c2}, v_{a2}\rangle}{m_c, r_c, m_a, r_a \vdash e_1 \Diamond_b e_2 \Downarrow \langle v_{c1}\Diamond_b v_{c2}, v_{a1}\Diamond_b v_{a2}\rangle}\ \textsc{Binary-Op} \qquad \frac{}{m_c, r_c, m_a, r_a \vdash v \Downarrow \langle v, \{\}\rangle}\ \textsc{Const}$$

$$\frac{m_c, r_c, m_a, r_a \vdash e \Downarrow \langle v_c, v_a\rangle \quad \begin{array}{l} r'_c = r_c[var \leftarrow v_c] \\ r'_a = r_a[var \leftarrow v_a] \end{array} \quad c' = \text{checkIDS}(c, pc) \quad \boxed{l' = l.\text{add}(\langle v_a, c'.\text{top}()\rangle)} \quad i = s[pc+1]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \texttt{var} := e \rightsquigarrow s, m_c, r'_c, m_a, r'_a, \Delta, c', l', pc+1, i}\ \textsc{Assign}$$

$$\frac{m_c, r_c, m_a, r_a \vdash e \Downarrow \langle v_c, v_a\rangle \quad c' = \text{checkIDS}(c, pc) \quad i = s[v_c]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \text{goto } e \rightsquigarrow s, m_c, r_c, m_a, r_a, \Delta, c', l, v_c, i}\ \textsc{Goto}$$

$$\frac{\begin{array}{l} m_c, r_c, m_a, r_a \vdash e \Downarrow \langle 1, v_a\rangle \\ m_c, r_c, m_a, r_a \vdash e_1 \Downarrow \langle v_{c1}, v_{a1}\rangle \end{array} \quad \begin{array}{l} c = \text{checkIDS}(c, pc) \\ c' = \text{updateIDS}(c, pc, v_a) \end{array} \quad l' = l.\text{add}(\langle v_{a1} \cup v_a, c'.\text{top}()\rangle) \quad i = s[v_{c1}]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow s, m_c, r_c, m_a, r_a, \Delta, c', l', v_{c1}, i}\ \textsc{True-Cond}$$

$$\frac{\begin{array}{l} m_c, r_c, m_a, r_a \vdash e \Downarrow \langle 0, v_a\rangle \\ m_c, r_c, m_a, r_a \vdash e_2 \Downarrow \langle v_{c2}, v_{a2}\rangle \end{array} \quad \begin{array}{l} c = \text{checkIDS}(c, pc) \\ c' = \text{updateIDS}(c, pc, v_a) \end{array} \quad l' = l.\text{add}(\langle v_{a2} \cup v_a, c'.\text{top}()\rangle) \quad i = s[v_{c2}]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow s, m_c, r_c, m_a, r_a, \Delta, c', l', v_{c2}, i}\ \textsc{False-Cond}$$

$$\frac{m_c, r_c, m_a, r_a \vdash e \Downarrow \langle v_c, v_a\rangle \quad c = \text{checkIDS}(c, pc) \quad c' = c.\text{push}(\langle pc+1, c.\text{pop}()\rangle) \quad i = s[v_c]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \text{call } e \rightsquigarrow s, m_c, r_c, m_a, r_a, \Delta, c', l, v_c, i}\ \textsc{Call}$$

$$\frac{m_c, r_c, m_a, r_a \vdash e \Downarrow \langle v_c, v_a\rangle \quad c' = \text{returnIDS}(c, pc) \quad \langle \Delta', l'\rangle = \text{delayedUpdate}(\Delta, l) \quad i = s[v_c]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \text{ret } e \rightsquigarrow s, m_c, r_c, m_a, r_a, \Delta', c', l', v_c, i}\ \textsc{Ret}$$

$$\frac{\begin{array}{l} m_c, r_c, m_a, r_a \vdash e_1 \Downarrow \langle v_{c1}, v_{a1}\rangle \\ m_c, r_c, m_a, r_a \vdash e_2 \Downarrow \langle v_{c2}, v_{a2}\rangle \end{array} \quad c' = \text{checkIDS}(c, pc) \quad \begin{array}{l} m'_c = m_c[v_{c1} \leftarrow v_{c2}] \\ m'_a = m_a[v_{c1} \leftarrow v_{a1}\Diamond_b v_{a2}] \end{array} \quad i = s[pc+1]}{s, m_c, r_c, m_a, r_a, \Delta, c, l, pc, \text{store}(e_1,\ e_2) \rightsquigarrow s, m'_c, r_c, m'_a, r_a, \Delta, c', l, pc+1, i}\ \textsc{Store}$$

Fig. 2: Operational semantics of input-bit dependence inference.

recursions, we use the same intuition as [54]: either when we enter the same function more than once or when we have the same immediate post-dominator, we replace the top element of the stack instead of pushing a new one.

Algorithm 4 illustrates three major functions to access the IDS. For every conditional branch, we call `updateIDS` to register a control-dependent region [54] from a conditional branch to an immediate post-dominator. For every return statement, we call `returnIDS` to clear up the IDS. We also update the IDS for every call instruction. Finally, we call `checkIDS` for every statement to check whether we have encountered the end of control-dependent region. When a CFG is incomplete, i.e., indirect jumps, we might not be able to find an immediate post-dominator. In this case, we use a conservative approach: we always merge the current set of control-dependent bits with the top element of the IDS.

We formulate the algorithm of input-bit dependence inference in an operational semantics in Figure 2. In the rule of assignment statement, we highlight the delay queue update using a box, because we optionally disable the function. In fact, even though we do not update the delay queue in assignment statements, we can still capture most of the input-bit dependence conditional branches. If we do not take the input-bit dependence for assignment statements, we may miss some dependence due to implicit data flow [28].

B. Example

Figure 3 is our running example showing a typical PNG parsing algorithm. It parses the first 8 characters using a series of conditional branches—which is an unrolled version of a for loop—from Line 2 to 14. It then reads the next 4 bytes as an integer in Line 16, and checks the value in Line 17. Figure 3b shows a control flow of the program, where each node is annotated with a line number of a branch instruction and a set of input bits affecting the condition of the branch at runtime. We use C to describe IBDI algorithm, but our system runs on raw binary executables. Additionally, we represent input positions in a byte-level granularity in our example for brevity.

Suppose we provide a valid PNG file to the parser, and follow the execution path of 1, 2, 5, 8, 14, 16, 17, and 19. On Line 1, Δ, l, and c are empty. When the first conditional branch is encountered on Line 2, we check which input bytes

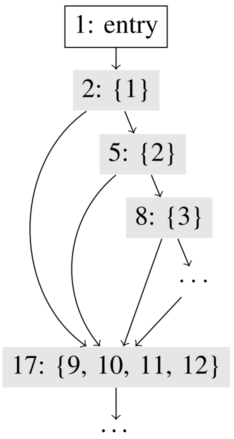
Algorithm 4: Input-dependence stack update algorithm.

```
1  Function updateIDS($c, pc, v_a$)
2      $pd \leftarrow$ immediate post-dominator of the current instruction at $pc$
3      $\langle toppd, topdep \rangle \leftarrow c$.top()
4      if $toppd = pd$ then
5          $\langle \cdot, c \rangle \leftarrow c$.pop()
6      end
7      $c' \leftarrow c$.push($\langle pd, v_a \cup topdep \rangle$)
8      return $c'$
9  end
10 Function returnIDS($c, pc$)
11     $\langle toppd, \cdot \rangle \leftarrow c$.top()
12     while $c.isNotEmpty() \wedge toppd \neq pc$ do
13         $\langle \cdot, c \rangle \leftarrow c$.pop()
14     end
15     if $c.isNotEmpty()$ then
16         $\langle \cdot, c \rangle \leftarrow c$.pop()
17     end
18     return $c$
19 end
20 Function checkIDS($c, pc$)
21     $\langle toppd, \cdot \rangle \leftarrow c$.top()
22     if $c.isNotEmpty() \wedge toppd = pc$ then
23         $\langle \cdot, c \rangle \leftarrow c$.pop()
24     end
25     return $c$
26 end
```

are affecting the condition. Since the first byte is affecting the condition, we call updateIDS($c, 2, \{1\}$), which first statically expands the control flow from the instruction on Line 2, and computes the immediate post-dominator of the instruction, which is Line 14 in this case. Then it updates c to have an element of $\langle 14, \{1\} \rangle$. Next, we push the input byte information $\langle \{1\}, \{1\} \rangle$ into l, which represent a dependence relation from the first byte to the first byte itself.[4]

On Line 5, we reach another conditional branch, which has a condition affected by the second byte. Since the top element of c has the same address as the immediate post-dominator of the branch, we replace the top element of c with $\langle 14, \{1, 2\} \rangle$ (due to Line 4-7 of updateIDS). The delayed queue is also updated with the updated control-dependence, which will call merge($\Delta, \{2\}, \{1, 2\}$) later in the delayedUpdate function. Similarly, we update the delay queue and the IDS until we reach the Line 14. Since the current instruction has the same address as in the top element of the IDS, we pop one element from the IDS (Line 23 of Algorithm 4), and then we call delayedUpdate of Algorithm 3 to update Δ. After executing Line 14, Δ has a mapping from each byte to the byte positions that the byte is dependent on. To be more precise, Δ should represent bit-level dependences, but we show byte-level dependences for simplicity. We perform the similar steps along the execution.

```
1   /* input read in buf */
2   if (buf[0] != '\x89'){
3     error();
4   }else{
5     if (buf[1] != '\x50'){
6       error();
7     }else{
8       if (buf[2] != '\x4e'){
9         error();
10      }else{
11        ...
12      }
13    }
14  }
15  /* next field */
16  len = *((*int32)&buf[8]);
17  if (len > PNG_MAX)
18    error();
19  ...
```



(a) A PNG parser in C (b) A control-flow graph.

Line	Δ	c	l
1	·	·	·
2	·	$\langle 14, \{1\} \rangle$	$\langle \{1\}, \{1\} \rangle$
5	·	$\langle 14, \{1, 2\} \rangle$	$\langle \{1\}, \{1\} \rangle$; $\langle \{2\}, \{1, 2\} \rangle$
8	·	$\langle 14, \{1, 2, 3\} \rangle$	$\langle \{1\}, \{1\} \rangle$; $\langle \{2\}, \{1, 2\} \rangle$; $\langle \{3\}, \{1, 2, 3\} \rangle$
14	$1 \mapsto \{1\}$ $2 \mapsto \{1, 2\}$ $3 \mapsto \{1, 2, 3\}$ ...	·	·
17	$1 \mapsto \{1\}$ $2 \mapsto \{1, 2\}$ $3 \mapsto \{1, 2, 3\}$ ...	$\langle 19, \{9, 10, 11, 12\} \rangle$	$\langle \{9, 10, 11, 12\},$ $\{9, 10, 11, 12\} \rangle$
19	$1 \mapsto \{1\}$ $2 \mapsto \{1, 2\}$ $3 \mapsto \{1, 2, 3\}$ ... $9 \mapsto \{9, 10, 11, 12\}$ $10 \mapsto \{9, 10, 11, 12\}$ ...	·	·

(c) The state transition table where each row is the execution context after executing the corresponding line. For delay queue l, each item is separated with a semicolon. The second column contains a mapping from a byte position to a set of byte positions.

Fig. 3: A PNG parser example. We represent the input positions using a *byte*-level granularity in this figure for brevity.

V. System Design

In this section, we describe SymFuzz, a system that automatically finds an optimal mutation ratio for mutational fuzzing based on the input-bit dependence inference. Figure 4 summarizes our system design, which consists of two major components: symbolic analysis and mutational fuzzing. The symbolic analysis module takes in a program and a seed, and returns a recommended optimal mutation ratio. The mutational fuzzing module then uses the mutation ratio to perform fuzzing, and outputs buggy inputs found. Finally, we triage buggy inputs using our safe stack hash technique described in §V-D.

[4] In a bit-level granularity, this represents the input-bit dependences between the first eight input bits, where each of the bits is dependent on each other.

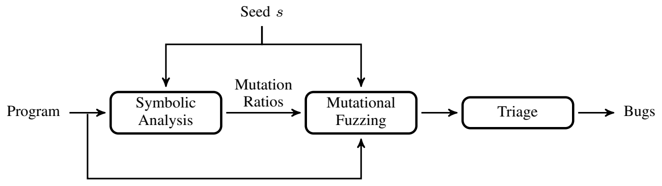


Fig. 4: SymFuzz architecture.

A. Implementation & Open Science

We base our system for input-bit dependence inference on BAP [9], an open-source binary analysis framework. BAP converts an x86 executable to an intermediate language suitable for program analysis. SymFuzz consists of 5,300 lines of OCaml for symbolic analysis and 1,600 lines of C++ code for instrumentation. We leverage PIN [34] to instrument a target binary. We also implement our mutational fuzzing framework in about 1900 lines of OCaml and 700 lines of C++. In support of open science, we release the source code of SymFuzz at http://security.ece.cmu.edu/symfuzz/.

B. Symbolic Analysis

The symbolic analysis module implements the operational semantics described in Figure 2 on top of the BAP [9] intermediate language. We employ several optimizations to our analysis including (1) tainted-block optimization, (2) JIT and PD caching, and (3) set memoization.

First, we use the taint information of each basic block to reduce the cost of symbolic analysis as follows. For each instrumented basic block, we perform a lightweight taint analysis. When a basic block does not involve any tainted instructions, we do not perform the symbolic analysis, and proceed to the next block. Notice our symbolic analysis inherently provides precise taint information for each block. Therefore, we do not need to maintain additional data structure for storing taints. In fact, the data-lineage tracking [32, 33] part of our symbolic analysis is more specific than the traditional taint analysis [14, 42, 57], and more abstract than the traditional symbolic execution [8, 27, 29].

Second, we employ several caches to improve the performance. The JIT cache is to cache recently-used BAP ILs, and the PD cache is to store the recently-computed immediate post-dominator nodes. We note that expanding a static CFG for each conditional branch to compute an immediate post-dominator is an expensive operation compared to symbolic evaluation. This is because it involves not only jitting but also recursive disassembling and graph analysis.

Finally, we note that IBDI uses significant amount of memory footprint, because each bit in a seed needs to store a pointer to a set of bit positions, and each memory bit that is touched by the program under test also stores such a set for in m_a. Although we perform byte-level analysis in our implementation, this problem still remains. The crux of the problem is that there can be multiple instances of the same set representation. Therefore, we memoize every set throughout the analysis, and make sure that there exists physically a single distinct set throughout the analysis.

C. Mutational Fuzzing

It is important to discuss how to design an algorithm for mutational fuzzing, because it is not straightforward as it seems[5], and none of the previous works discuss this problem. For random fuzzing, there is a trivial $O(N)$ algorithm, namely, generating a random number between 0 and $2^N - 1$ using a Pseudorandom Number Generator (PRNG) [36, 37], where each bit takes $O(1)$ time. However, designing an algorithm for mutational fuzzing is not trivial, because we want to generate only inputs in a K-neighbor of the input space for a fixed K.

Particularly, we need an ability to selecting test cases that have the exact Hamming distance K from the seed s, for any given K. This is equivalent to selecting K-bit positions from the seed and flipping them, because flipping K bits results in a test case i such that $\delta(i, s) = K$.

This is the classic random k-subset selection problem. The crux of the problem is to devise an algorithm to select K elements at random from N bit positions. A straightforward algorithm such as computing a random permutation and taking the top K elements requires $O(N)$ space and time complexity. One might consider using reservoir sampling [52] to reduce the space complexity to $O(K)$, but it still requires $O(N)$ time complexity. There are several known algorithms that have $O(K)$ space complexity while requiring only $O(K)$ time complexity in expectation [4, 43]. In this work, we use Floyd-Bentley's algorithm [4, Algorithm F1] to compute the subset. This algorithm outperforms permutation-based algorithms in terms of both time and space complexity when $K < N$. Once we obtain K random bits to modify from the Floyd-Bentley's algorithm, we simply flip the selected K bits (by XORing the bits with s), and generate a new test case.

Someone may argue that the difference between random mutation with or without replacement is negligible when the input size is reasonably large, and it is even easier to analyze the probability when we assume the random process with replacement. However, note that the effect of using random mutation with replacement can be significant in mutational fuzzing as the mutation ratio does not precisely determine the number of bits to be flipped. Of course, it is possible to use a rejection sampling to resolve the problem, but the problem becomes worse when the mutation ratio gets bigger, especially when it is close to 1.0. Suppose the mutation ratio is 0.9, then it is probabilistically infeasible to get the desired number of bit flips with a rejection sampling.

D. Safe Stack Hash

Security practitioners use a call-stack trace or a part of a call-stack trace [39], e.g., taking only top five entries of a full stack-trace as in the fuzzy stack hash [41]. The rationale is that if two crashes have the same call-stack traces, then they

[5] In fact, we are not aware of any practical fuzzers that implement exact mutational fuzzing. For example, even calculating the exact number of bits flipped by these fuzzers is difficult, because there is no closed-form expression for their algorithm.

Program	**#Crashes**	**#Bugs**	**Seed Size (bits)**	**Seed Type**
abcm2ps	299,204	34	35,040	abc
autotrace	15,848	23	16,304	bmp
bib2xml	603	2	177,152	bib
catdvi	2,045,327	8	1,632	dvi
figtoipe	443,301	38	8,016	fig
gif2png	21,600	3	1,816	gif
pdf2svg	125	1	23,368	pdf
mupdf	30	5	23,368	pdf
Total	2,826,038	114		

TABLE III: The ground truth data.

are likely to have an equivalent final program state, and thus, it is an evidence of having the same root cause. This approach works for many cases, but it exhibits a false bucketing problem: it can put a single bug into multiple buckets.

We note that this false bucketing problem can significantly increase the number of bugs found for fuzzing especially when a buffer overflow mangles the return addresses on the stack. For example, suppose a mutated input data overwrites a return address of a call stack. The return address of the stack trace may contain any arbitrary values due to the mutation. In the worst case, we can have 2^{32} distinct call-stack traces on 32-bit machine just because of the mangled return address.

To mitigate this problem, we employ a technique, called *safe stack hash*, which stops traversing the call stack when it finds an unreachable return address. Specifically, we check for each return address of a call-stack trace starting from the top, i.e., the crashing stack frame, whether each return address falls in a mapped page. If not, we assume that the stack is mangled in the corresponding stack frame, and discard the rest of the return addresses in the call-stack trace. We also use the same heuristic as the fuzzy stack hash, and consider only the top five stack entries when computing the hash. Notice the number of bugs found from safe stack hash can only be less than the one from regular stack hash techniques such as the fuzzy stack hash. We implemented our safe stack hash using a GDB script written in Python.

VI. Evaluation

We now evaluate our system SymFuzz on 8 real-world applications in order to answer the following questions.

1) Does it make sense to optimize the mutation ratio in mutational fuzzing? How does the number of bugs differ per mutation ratio? (§VI-B)
2) What is the cardinality of minimum buggy bitsets? Is the conventional wisdom about choosing small mutation ratios correct? (§VI-C)
3) How effective is it to use the SymFuzz's adaptive strategy in terms of number of bugs found? (§VI-D)
4) Does SymFuzz work well in practice? How many bugs did we find compared to the practical fuzzers such as BFF, zzuf, AFL-fuzz? (§VI-E)

A. *Experimental Setup*

We ran experiments on a private cluster consisting of 8 virtual machines. Each VM was running Debian Linux 7.4 on a single Intel 2.68 GHz Xeon core with 1GB of RAM, and all the applications that we tested were up-to-date as of May 2014. Each VM in our cluster was committed to only a single application throughout the experiments. The number of bugs reported from this paper is based on our safe stack hash introduced in §V-D. We also make our source code publicly available at http://security.ece.cmu.edu/symfuzz/.

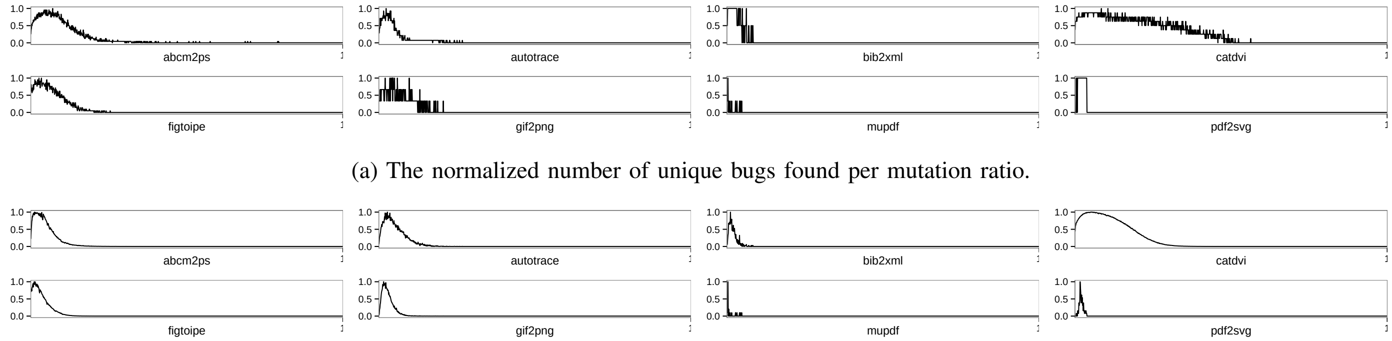
Collecting Ground Truth. We ran the mutational fuzzing module of SymFuzz individually to gather the ground-truth data of mutational fuzzing. We initially obtained a list of 100 file-conversion applications of Debian as in [53], and manually created a seed file for each application. We then fuzzed all 100 program-seed pairs with BFF [26] to know which programs exhibit crashes. We found 8 programs that resulted in at least one crash. We first ran our tool on each of the programs for 1,000 hours using 1,000 distinct mutation ratios from 0.001 to 1.000, i.e., 1 hour per each mutation ratio. Table III summarizes our ground truth experiment. In total, we have spent 8,000 CPU hours fuzzing the applications, and found 114 previously unknown bugs based on our safe stack hash. Since all the applications that we tested read in an input file, all the bugs found are potentially on the attack surface. For example, an attacker can craft a malicious file and upload it to the Internet, or send it as an email attachment in order to compromise users that run the applications with the file. We leave it as future work to check the exploitability of the bugs found [3, 12, 20].

B. *Mutation Ratio Optimization*

To justify our research, we first studied our ground truth data from fuzzing, and measured how the effectiveness of fuzzing changes with respect to the mutation ratio. We answer two specific questions as follows. First, is it meaningful to optimize the mutation ratio? Second, what is the potential benefit of using an adaptive optimization for fuzzing?

1) Is Mutation Ratio Optimization Useful?: Optimizing mutation ratio is useful when the result of fuzzing varies significantly depending on a mutation ratio, and when there is a clear bias in the distribution of mutation ratios in terms of fuzzing efficiency. Figure 5a and Figure 5b illustrate respectively the normalized number of bugs and crashes found for each of the 8 programs in our ground truth dataset, that is, the number of bugs (crashes) divided by the maximum attainable number of bugs (crahses). Both figures show that the effectiveness of fuzzing largely depends on the mutation ratio. For example, we found the maximum number of bugs from `abcm2ps` using the mutation ratio of 0.071, but did not find any bug from the same program using the mutation ratio of 0.262. However, using the mutation ratio of 0.071 on `pdf2svg`, we found no bug in our dataset.

We note that the optimal mutation ratios differ across the programs. Figure 6 shows an empirically optimal mutation ratio per program based on the number of bugs found. The optimal ratios range from 0.003 to 0.085 depending on the program under test. We also notice fuzzing efficiency is biased towards the optimal mutation ratios from both the figures. Thus,



(a) The normalized number of unique bugs found per mutation ratio.

(b) The normalized number of crashes found per mutation ratio.

Fig. 5: The effectiveness of fuzzing per mutation ratio evaluated over 1,000 mutation ratios from 0.001 to 1.000.

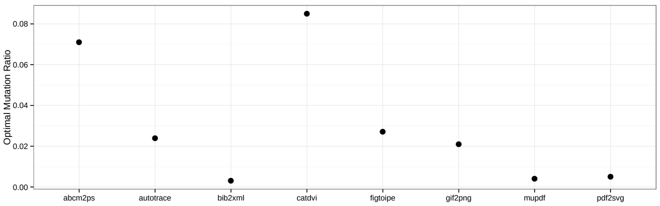


Fig. 6: Empirically best mutation ratios for 8 programs.

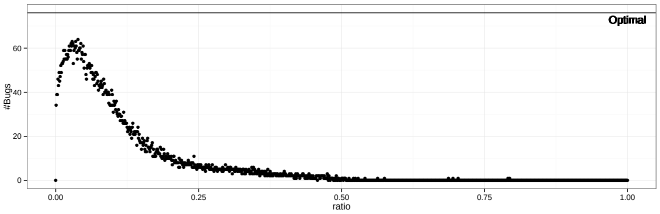


Fig. 7: Comparison between a non-adaptive method, which is to choose a single default mutation ratio, and an adaptive (optimal) method, which is to select an empirically optimal mutation ratio per program.

our data suggest that mutation ratio optimization is useful in fuzzing.

2) How Much Better to Use Adaptive Optimization?: An immediate follow-up question of the first question is: how much better can adaptive optimization strategies be compared to non-adaptive strategies? In particular, we want to know what is the potential gain of using an adaptive strategy over non-adaptive strategies such as selecting either (1) a single default mutation ratio, or (2) multiple ratios at random from a given range. Both the approaches are indeed employed in zzuf [30]. To answer the question, we first computed the maximum possible number of bugs that can be obtained by an optimal adaptive strategy for each program from our dataset; it was 76. We then compared this number against the number of bugs that can be found from the non-adaptive strategies.

The first non-adaptive strategy we checked is to choose a single default mutation ratio throughout an entire fuzzing campaign. Figure 7 shows the comparison. For all the mutation ratios in our dataset, the optimal adaptive strategy—represented as the horizontal line at the top of the figure—always found more bugs than the non-adaptive way. Moreover, even for the best case of the non-adaptive strategy, which is to choose the ratio of 0.039, the adaptive optimization found 18.8% more bugs compared to the non-adaptive method. Additionally, we notice that if we consider only a single mutation ratio per program, even a perfect adaptive strategy can only find 76 bugs out of 114 from our dataset. This result suggests the need for inferring multiple instead of a single mutation ratio, although this is outside the scope of this paper (see §VII for discussion).

The second strategy that we evaluated is to select a fixed range of mutation ratios throughout a fuzzing campaign. We used three different ranges suggested by the zzuf manual for this comparison, namely, $[0.00001, 0.01]$, $[0.00001, 0.02]$, and $[0.00001, 0.10]$. We fuzzed each application in our dataset for 1 hour with each of the ranges. In this experiment, we used the same algorithm that zzuf employs for selecting mutation ratios from a given range, which works as follows. We first discretize the given range into a set of uniformly distributed mutation ratios, where the cardinality of the set is 65,535. We then select a mutation ratio from the set uniformly at random for each fuzzing iteration. The best range was $[0.00001, 0.02]$, which results in 44 bugs from our dataset. This number of bugs was indeed 57.9% of the optimal adaptive case. From the two experiments, we conclude that optimizing the mutation ratio benefits fuzzing in practice.

C. Distribution of b Values

In this subsection, we answer the following two questions. First, how do we compute b from a crash? Second, what is the distribution of b in crashes of real-world programs?

Recall from §III-B, we estimate an optimal mutation ratio r using $\bar{d}$, which depends upon the distribution of b values. To obtain the distribution, we first collected 4,255 distinct pairs of a crashing input and a seed from our previous study [44]. The crashing inputs are gathered by fuzzing a variety of applications that take in a file as an input for over 650 CPU days. The size of the seeds in the dataset ranged from 43B

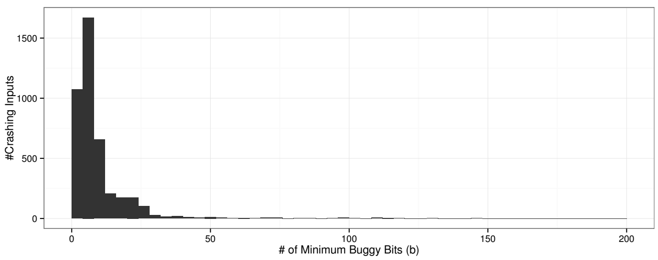


Fig. 8: The number of minimum buggy bits for 4,255 crashing inputs derived from previous studies. The average was 9 and the median was 6.

Program	$\bar{d}$	**Seed Size** N	**#Bugs**	**Max. #Bugs**	**Diff.**
abcm2ps	164	35,040	22	24	2
autotrace	69	16,304	10	14	4
bib2xml	484	177,152	2	2	0
catdvi	24	1,632	7	8	1
figtoipe	44	8,016	14	21	7
gif2png	144	1,816	3	3	0
pdf2svg	434	23,368	0	1	1
mupdf	201	23,368	1	3	2

TABLE IV: The number of bugs found with IBDI.

to 31MB, and the average seed size was 954KB. For each crashing input, we computed the Hamming distance from them to the corresponding seeds. The average Hamming distance was 151,721; the median was 11,430; and the standard deviation was 862,055. Notice that the Hamming distance in this case does *not* represent the size of a minimum buggy bitset: it represents the size of a buggy bitset instead.

To compute the size of a minimum buggy bitset (b), we used a delta debugging technique [15, 59] called bug minimization, presented by Householder et al. [25]. The idea is simple: given a crashing input and a corresponding seed, bug minimization iteratively restores bits in the crashing input to the original value of the seed, and determines which bit flips are necessary to crash the program. After the minimization, we compute the Hamming distance from each minimized crashing input to its corresponding seed, which is essentially the value of b.

We used the above algorithm in order to compute the distribution of b in the 4,255 distinct crashes that we collected. Figure 8 shows the distribution of b values from our dataset. We found that it is enough to flip 9 bits of a seed on average to trigger crashes in our dataset. The median Hamming distance was 6, and the standard deviation was 18. More than 80% of the b values were less than or equal to 10. In addition, we performed the same experiment on our ground truth data. As a result, we obtained the Hamming distance of 5 on average (median 3), and the standard deviation of the Hamming distance was 10. The result shows that most of the crashes can be triggered by flipping only few bits—less than a byte size in our dataset—from the corresponding seeds.

How Many Bits to Flip? It is important to note that the above result does not necessarily mean that we need to flip only few bits of a seed to effectively trigger program crashes in mutational fuzzing. For example, there may be an input field that is independent from crashes: no matter what value the field has, we can still crash the program. Therefore, in this case, we want to flip more than b bits to increase the likelihood of finding crashes. We indeed found the most number of bugs in `abcm2ps` using a mutation ratio of 0.071, which corresponds to about $2,500$ bit flips. This result highlights the key idea of this paper: a good mutation ratio depends on the input-bit dependence of a seed.

D. Estimating r

Recall from §III-B, the core part of SymFuzz is to derive the average number of dependent bits ($\bar{d}$) from a distribution of b in estimating r. We used Algorithm 1 to compute $\bar{d}$ and obtained r for each program. Table IV summarizes the result. The second column of the table shows $\bar{d}$. The third column of the table is the size of the seed N that is used for each of the programs. The fourth column is the number of bugs found using the obtained r for 1 hour of fuzzing. The fifth column is the maximum attainable number of bugs in each program for 1 hour of fuzzing when the empirically optimal mutation ratio is selected. The last column is the difference between the number of bugs found with SymFuzz and the optimal number of bugs.

SymFuzz successfully estimated effective mutation ratios for each program, and found 77.6% of the bugs that can be found from the optimal adaptive strategy. Most mutation ratios that we obtained was close to optimal mutation ratios except for the case of `figtoipe`. To investigate the problem, we first ran bug minimization on every unique crash that we obtained for `figtoipe`. We then checked the cardinality of the minimum buggy bitsets (b) for the crashes, and found that $\bar{d}$ was 5× greater than the average input-bit dependence for the minimum buggy bitsets, which results in a smaller mutation ratio than the optimal one. This is a corner case where buggy bits are not close to the other bits in a seed, in which our algorithm can perform poorly.

E. SymFuzz *Practicality*

In this subsection, we test the practicality of mutation ratio optimization by comparing the number of bugs found with existing mutational fuzzers such as BFF, zzuf, and AFL-fuzz.

1) Comparison against BFF and zzuf: The closest practical mutational fuzzers in terms of the underlying mutation process are BFF and zzuf: they use bit-flipping-based mutation for fuzzing. We fuzzed each of the programs in our dataset for 1 hour using zzuf, BFF, and SymFuzz, and compared the number of bugs found. To run zzuf, we used a single mutation ratio of 0.004, which is a default ratio. Notice BFF uses dynamic scheduling algorithm to automatically find good mutation ratios to use, whereas zzuf requires an analyst to specify either a mutation ratio or a range of mutation ratios. In total, BFF found 43 bugs; zzuf found 38 bugs; and SymFuzz found 59 bugs. The result indicates that SymFuzz's adaptive strategy found 37.2%

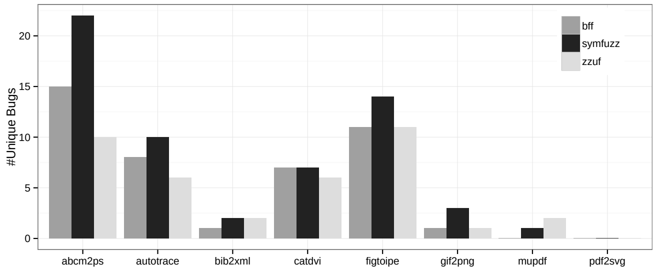


Fig. 9: Final comparison in the number of bugs found.

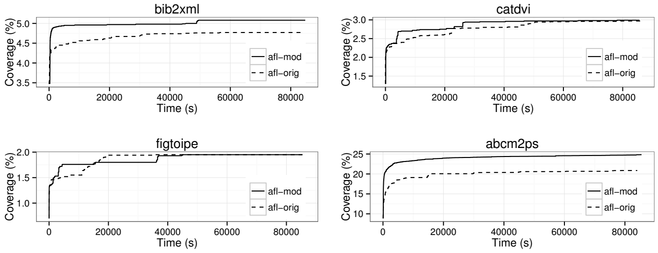


Fig. 10: Branch coverage difference between AFL-fuzz and modified AFL-fuzz (with mutation-ratio-based mutation logic).

and 55.3% more bugs than BFF and zzuf respectively. For further analysis, we show a head-to-head comparison against BFF and zzuf for each program in Figure 9. Notice that SymFuzz found equal or more number of bugs compared to BFF for all the programs. SymFuzz was also superior than zzuf except for one program: `mupdf`. The primary reason is because the performance of fuzzing for `mupdf` is sensitive to the mutation ratio from Figure 5a. SymFuzz obtained a ratio of 0.003, which is just 0.001 off from the empirically optimal mutation ratio (0.004). zzuf's default mutation ratio happened to be the same as the optimal one.

2) *Comparison against AFL-fuzz:* AFL-fuzz [58] is the state-of-the-art mutational fuzzer that is used by many security practitioners. The mutation process of AFL-fuzz consists of two major phases. First, it performs a series of deterministic bit-flipping algorithms based on several heuristics. Second, it uses a random combination of the algorithms in order to non-deterministically generate test cases. These two steps are applied for every seed during a fuzzing campaign. If any one of the generated test cases exhibits a new execution path (based on branch coverage), AFL-fuzz uses it as a new seed.

Since AFL-fuzz uses radically different mutation algorithms than SymFuzz, we cannot measure the effectiveness of mutation ratio optimization by directly comparing AFL-fuzz with SymFuzz. Instead, we replaced the first phase of AFL-fuzz with SymFuzz's mutation algorithm with mutation ratio optimization, which allows us to compare the effect of using our algorithm over their bit-flipping mutation algorithm. We downloaded AFL-fuzz 1.45b for this experiment. We ran both the modified AFL-fuzz and the original AFL-fuzz on 7 programs (excluding `mupdf` because AFL-fuzz does not support GUI application) for 24 hours. After 24 hours of fuzzing we triaged all the crashes found using our safe stack hash. As a result, we found 54 bugs from the original AFL-fuzz, and 64 bugs from the modified AFL-fuzz. In other words, we found 18.5% more bugs by applying our technique on AFL-fuzz. We also computed the branch coverage per time during the 24 hours of fuzzing. Figure 10 shows the coverage differences in 4 applications that present the most significant differences; we did not observe significant coverage differences from the rest. We conclude that AFL-fuzz can benefit from our technique in our dataset.

VII. Limitation & Future Work

Statistical Significance. Currently, SymFuzz outperforms previous fuzzers in our dataset. However, the result may change with other applications that have different statistical properties in terms of b and d values. Furthermore, our ground truth dataset is based only on fuzzing campaigns of one hour. Since fuzzing usually runs for several weeks in practice, fuzzing longer would allow a stronger conclusion. We leave fuzzing for more time as future work.

Multiple Mutation Ratios. Two distinct bugs can have significantly different d values, although our current strategy focuses on finding a single $\bar{d}$ from the overall average of d values. This is a fundamental limitation of SymFuzz because we do not know exact minimum buggy bitsets prior to fuzzing. One potential future direction is to consider multiple d values and perform scheduling over the derived mutation ratios.

Seed File. Currently, we assume a seed file for a program is given by an analyst. This is a common assumption for most of the fuzzers in practice. Recent work [44] partially addresses the problem using a coverage-based inference. We leave combining the seed selection algorithm with SymFuzz as future work. Additionally, our analysis only analyzes a single execution path based on a given seed. Therefore, it is possible to miss several input-bit dependence relations that manifest only when a different execution path is taken. Furthermore, our operational semantics (§IV-A) do not differentiate bit-level operators such as logical-AND from other operators. This may result in an over-approximated results for our analysis, i.e., some bits may have more dependent bits than it is supposed to be. Guaranteeing a bit-level accuracy is out of scope of this work.

Input-Format-Aware Fuzzing. Although IBDI is inspired by automatic input format recovery [10, 17, 32], our technique currently does not leverage the input field information. One may use the existing input format recovery techniques to find a set of input fields, and mutate a set of specific input fields instead of fuzzing the entire seed file. One potential research direction is to derive the optimal time allocation for each of the input fields in order to maximize the number of bugs found.

VIII. Related Work

A. *Automatic Reversing*

Automatic input format recovery (reversing) is close to IBDI in a sense that both techniques try to find the relationship between input bits. However, IBDI does not try to recover the exact structural format of the input. Instead, IBDI tries to find the input-bit dependence to compute the optimal mutation ratio. Several researches on automatic input format recovery use taint analysis to recover the basic structure of an input [10, 16, 17]. Lin et al. [32] also combined the dynamic control dependence analysis with data lineage tracing to reverse the input structure from an execution trace. Their work is the closest previous work, but their algorithm is different from ours due to different goals, e.g., they ignore binary operators in their algorithm. There is also a static approach by Lim et al. [31]. We believe automatic reversing and IBDI are complementary. For example, if we are given an input format, one may be able to extract dependence relation from the format by considering every bit in a record field are dependent each other. One potential future direction is to leverage existing input format recovery tools.

B. Combining White-Box and Black-Box

There are several approaches to combining multiple testing techniques. Hybrid concolic testing [35] is the first attempt in combining symbolic execution and mutational fuzzing. It interleaves mutational fuzzing and concolic testing when there is no more coverage increase with the hope that it can discover novel execution paths. In contrast, IBDI uses a white-box analysis as a pre-processing step of a black-box testing. Yang et al. [55] attempted to use symbolic execution to figure out which input vectors are related. Then they utilized this information to perform combinatorial testing. Unlike their approach, IBDI uses an abstract analysis that does not require an SMT solver, and focuses on the dependence relationship between input bytes.

C. Fuzzing

The term fuzz testing (fuzzing) was coined in the 90s [38] to mean a software testing technique that executes the program under test using a series of random inputs. Since then, the term has been overloaded to mean different types of testing techniques including mutational fuzzing and grammar-based fuzzing. Our main focus is on mutational fuzzing. Most of existing mutational fuzzing algorithms [1, 19, 23, 30, 46, 58] rely on ad-hoc heuristics, which make it hard to mathematically model and analyze them. For example, zzuf [30] uses an arbitrary interval to discretize a continuous parameter such as mutation ratio, and notSPIKEfile [23] utilizes a predefined list of string tokens to construct and mutate inputs. One of our contributions is that we designed the first mutational fuzzer which we can mathematically analyze.

Grammar-based fuzzing (a.k.a. generation-based fuzzing) [21, 24, 40, 45, 56] is another class of fuzzing, which differs from mutational fuzzing since it does not require a seed file. Instead, it generates test cases from a given input specification, e.g., network or file format. Grammar-based fuzzing shares the common theme as mutation ratio optimization: exploit the knowledge about the input structure. We leave it as future work to study the opportunities of applying our technique to grammar-based fuzzing.

IX. Conclusion

We designed an algorithm to optimize the mutation ratio in mutational fuzzing given a program and a seed. In particular, we introduced SymFuzz, which runs both black- and white-box analysis to find bugs in a program. We also have formulated the failure rate of mutational fuzzing in terms of the input-bit dependences among bit positions in an input. Our mathematical model led us to design a novel technique for mutation ratio optimization, which estimates a probabilistically optimal mutation ratio from an execution trace. With our data set, we showed that SymFuzz can find 39.5% more bugs than BFF and 57.9% more bugs than zzuf in the same amount of fuzzing time. We have also applied our technique to improve AFL-fuzz. With our modifications, AFL-fuzz was able to find 18.5% more bugs in the same 24-hour experiment.

Acknowledgement

We thank Thanassis Avgerinos for encouragement and fruitful discussions. This work was supported in part by grants from the National Science Foundation under grant CNS-0720790 and the Department of Defense under Contract No. N66001-13-2-4040. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

Appendix A
Solving NLP

Recall from §III-A, we solve the NLP problem to optain an optimal mutation ratio for a given minimum buggy bitset.

Theorem 1 (Optimal Mutation Ratio)**.** *Given a minimum buggy bitset B' and the corresponding $\uparrow_s^p(B')$ for a program p and a seed s, let $b = |B'|$ and $d = |\uparrow_s^p(B')|$. The optimal mutation ratio r for finding the bug $\epsilon(\sigma_p(\mu(s, B')))$ is*

$$r = \frac{b \times (N+1)}{d \times N} \textit{ when } N \cdot r > b. \tag{2}$$

Proof: The goal of the NLP is to maximize the following failure rate

$$\theta_b = \frac{\binom{N-d}{N\cdot r-b}}{\binom{N}{N\cdot r}}.$$

For simplicity, we let u to denote $N \cdot r - b$, and v to denote $N - d$. Then the failure rate is simplified as follows.

$$\frac{\binom{v}{u}}{\binom{N}{u+b}}.$$

By expanding the binomial coefficients, we have

$$\frac{\binom{v}{u}}{\binom{N}{u+b}} = \frac{\frac{v!}{u!(v-u)!}}{\frac{N!}{(b+u)!(N-u-b)!}} = \frac{v!(b+u)!(N-b-u)!}{u!(v-u)!N!}.$$

When $u = 1$, the failure rate is

$$\frac{v!(b+1)!(N-b-1)!}{1!(v-1)!N!}.$$

When $u = 2$, the failure rate is

$$\frac{v!(b+2)!(N-b-2)!}{2!(v-2)!N!}.$$

We note that, as we increase u by one, the failure rate increases by the factor of

$$\frac{(v-u+1)(b+u)}{u(N-b-u+1)} \text{ when } u > 0.$$

Since the factor monotonically decreases in terms of u, the maximum failure rate can be achieved when the factor becomes 1. This relaxation gives us the maximum failure rate when

$$\frac{(v-u+1)(b+u)}{u(N-b-u+1)} = 1 \text{ when } u > 0.$$

Solving the equation with respect to u, we have

$$u = \frac{b(v+1)}{N-v} \text{ when } u > 0.$$

Since $u = N \cdot r - b$ and $v = N - d$, we can further simplify the equation with respect to the mutation ratio r:

$$r = \frac{b \times (N+1)}{d \times N} \text{ when } N \cdot r > b.$$

■

References

[1] P. Amini, A. Portnoy, and R. Sears, "Sulley," https://github.com/OpenRCE/sulley.

[2] A. Appel, *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[3] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014.

[4] J. Bentley and B. Floyd, "Programming Pearls: A Sample of Brilliance," *Communications of the ACM*, vol. 30, no. 9, pp. 754–757, 1987.

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

[6] D. P. Bertsekas and D. P. Bertsekas, *Nonlinear Programming*, 2nd ed. Athena Scientific, 1999.

[7] Y. M. Bishop, S. E. Fienberg, and P. W. Holland, *Discrete Multivariate Analysis: Theory and Practice*. Cambridge: MIT Press, 1975.

[8] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution," *ACM SIGPLAN Notices*, vol. 10, no. 6, pp. 234–245, Apr. 1975.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz, "BAP: A Binary Analysis Platform," in *Proceedings of the International Conference on Computer Aided Verification*, 2011, pp. 463–469.

[10] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007, pp. 317–329.

[11] 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*, 2008, pp. 209–224.

[12] 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*, 2012, pp. 380–394.

[13] T. Y. Chen and Y. T. Yu, "On the Relationship Between Partition and Random Testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 977–980, 1994.

[14] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2007, pp. 196–206.

[15] H. Cleve and A. Zeller, "Locating Causes of Program Failures," in *Proceedings of the International Conference on Software Engineering*, 2005, pp. 342–351.

[16] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009, pp. 110–125.

[17] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic Reverse Engineering of Input Formats," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2008, pp. 391–402.

[18] L. De Moura and N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," *Communications of the ACM*, vol. 54, no. 9, pp. 69–77, 2011.

[19] M. Eddington, "Peach Fuzzing Platform," http://peachfuzzer.com/.

[20] J. Foote, "CERT Linux Triage Tools," http://www.cert.org/blogs/certcc/2012/04/cert_triage_tools_10.html.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based Whitebox Fuzzing," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2008, pp. 206–215.

[22] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008, pp. 151–166.

[23] A. Greene, "notSPIKEfile," http://www.securiteam.com/tools/5NP031FGUI.html.

[24] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the USENIX Security Symposium*, 2012, pp. 445–458.

[25] A. D. Householder, "Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File," CERT, Tech. Rep. CMU/SEI-2012-TN-018, 2012.

[26] A. D. Householder and J. M. Foote, "Probability-Based Parameter Selection for Black-Box Fuzz Testing," CERT, Tech. Rep. CMU/SEI-2012-TN-019, 2012.

[27] W. Howden, "Methodology for the Generation of Program Test Data," *IEEE Transactions on Computers*, vol. C-24, no. 5, pp. 554–560, 1975.

[28] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted

Control-Flow Propagation," in *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[29] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[30] C. Labs, "Zzuf: Multi-Purpose Fuzzer," http://caca.zoy.org/wiki/zzuf.

[31] J. Lim, T. Reps, and B. Liblit, "Extracting Output Formats from Executables," in *Proceedings of the Working Conference on Reverse Engineering*, 2006, pp. 167–178.

[32] Z. Lin and X. Zhang, "Deriving Input Syntactic Structure from Execution," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008, pp. 83–93.

[33] Z. Lin, X. Zhang, and D. Xu, "Convicting Exploitable Software Vulnerabilities: An Efficient Input Provenance Based Approach," in *Proceedings of the International Conference on Dependable Systems Networks*, June 2008, pp. 247–256.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[35] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *Proceedings of the International Conference on Software Engineering*, 2007, pp. 416–426.

[36] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[37] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[38] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[39] C. Miller, "Babysitting an Army of Monkeys," in *CanSecWest*, 2010. [Online]. Available: http://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf

[40] C. Miller and Z. N. J. Peterson, "Analysis of Mutation and Generation-Based Fuzzing," Independent Security Evaluators, Tech. Rep., 2007.

[41] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs," in *Proceedings of the USENIX Security Symposium*, 2009, pp. 67–82.

[42] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[43] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms for Computers and Calculators*, 2nd ed., ser. Computer Science and Applied Mathematics. Academic Press, 1978.

[44] 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*, 2014, pp. 861–875.

[45] J. Ruderman, "jsfunfuzz," http://blog.mozilla.org/security/2007/08/02/javascript-fuzzer-available/, 2007.

[46] M. Sutton, "FileFuzz," http://osdir.com/ml/security.securiteam/2005-09/msg00007.html.

[47] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[48] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[49] C. S. Team, "Clusterfuzz," https://code.google.com/p/clusterfuzz/.

[50] S. K. Thompson and G. A. F. Seber, *Adaptive Sampling*. Wiley, 1996.

[51] P. Uhley, "A Basic Distributed Fuzzing Framework for FOE," https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe.html.

[52] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.

[53] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling Black-box Mutational Fuzzing," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 511–522.

[54] B. Xin and X. Zhang, "Efficient Online Detection of Dynamic Control Dependence," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2007, pp. 185–195.

[55] J. Yang, H. Zhang, and J. Fu, "A Fuzzing Framework Based on Symbolic Execution and Combinatorial Testing," in *IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, 2013, pp. 2076–2080.

[56] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[57] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007, pp. 116–127.

[58] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[59] A. Zeller, "Isolating Cause-effect Chains from Computer Programs," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2002, pp. 1–10.