Hunting Bugs with Lévy Flight Foraging

Konstantin Böttinger
Fraunhofer Institute for Applied and Integrated Security
85748 Garching (near Munich), Germany
Email: konstantin.boettinger@aisec.fraunhofer.de

***Abstract*—We present a new method for random testing of binary executables inspired by biology. In our approach we introduce the first fuzzer based on a mathematical model for optimal foraging. To minimize search time for possible vulnerabilities we generate test cases with Lévy flights in the input space. In order to dynamically adapt test generation behavior to actual path exploration performance we define a suitable measure for quality evaluation of test cases. This measure takes into account previously discovered code regions and allows us to construct a feedback mechanism. By controlling diffusivity of the test case generating Lévy processes with evaluation feedback from dynamic instrumentation we are able to define a fully self-adaptive fuzzing algorithm.**

I. Introduction

As software ever increases in size and complexity we face the significant challenge to validate the systems surrounding us. Penetration testing of software has come a long way from its origins and nowadays shows an extensive diversity of possible strategies. All of them have the common aim to achieve maximal code coverage by generating suitable program inputs, also called test cases. Possible approaches range from dynamic symbolic [1], [2] and concolic [3], [4], [5] execution to more or less random testing using generational, mutational, black-box, or white-box fuzzers [6], [7]. Within the latter domain of random test generation current strategies for input generation basically rely on heuristics and sophisticated guessing. It is still an open question how to optimally generate inputs that trigger a maximum number of bugs in a finite amount of time.

In the course of researching new effective search strategies we find similar problems in biology, particularly in the field of optimal foraging. A variety of biological systems let us observe optimal strategies for finding energy sources by simultaneously avoiding predators. When we identify sources of food with possible vulnerabilities in binary executables and predators with the overhead of execution runtime, we are inspired to adapt mathematical models of optimal foraging to test case generation. This approach enables us to take stochastic models of optimal foraging as a basis for input mutation. In particular we rely on Lévy flights to search for bug triggering test cases in input space.

Before summarizing our contributions we first give some short background on fuzzing, optimal foraging, and the Lévy flight hypothesis.

a) Fuzzing: There exists a substantial diversity of test case generation strategies for random testing binaries. All these approaches have in common to a greater or lesser extent the random generation of test cases with the aim of driving the targeted program to an unexpected and possibly exploitable state. The most significant advantage of fuzzing is its ease of use. Most executable binaries that process any input data are suitable targets for random test generation and effective fuzzers are implemented in a short time.

b) Optimal Foraging: Observing biological systems has led to speculation that there might be simple laws of motion for animals searching for sources of energy in the face of predators. Regardless of whether we look at bumblebees[8], fish and hunting marine predators in the sea [9], [10], grey seals [11], spider monkeys [12], the flight search patterns of albatrosses [13], the wandering of reindeer [14], the reaction pathways of DNA-binding proteins [15], or the neutralisation of pathogens by white blood cells [16], we can discover emerging movement patterns all those examples have in common. Mathematical modelling such common patterns is an active field of research in biology and is more generally referred to as *movement ecology*. While the physics of foraging [17] provides us several possible models our choice is not guided by accuracy with respect to the biological process but by minimization of software bug search time. This leads us to the special class of stochastic processes called *Lévy flights* which we discuss in more detail in Section III.

c) Lévy Flight Hypothesis: Within the variety of models for optimal foraging Lévy flights have several characteristic properties that show promise for software testing. In particular, the Lévy flight hypothesis accentuates the most significant property of these kinds of stochastic processes for our purposes. It states that Lévy flights minimize search time when foraging sources of food that are sparsely and randomly distributed, resting, and refillable. These assumptions match to the properties of bugs in software (with the obvious interpretation that *refillable* translates to the fact that software bugs stay until fix). In addition to the mathematical Lévy flight hypothesis, the Lévy flight *foraging* hypothesis in theoretical biology states that these processes actually model real foraging behavior in certain biological systems due to natural selection. The Lévy flight hypothesis constitutes the major connection link between optimal foraging theory and random software testing.

In this paper we propose a novel method for random software testing based on the theory of optimal foraging. In summary, we make the following contributions:

- We introduce a novel fuzzing method based on Lévy flights in the input space in order to maximize coverage of execution paths.

DOI 10.1109/SPW.2016.9

- We define a suitable measure for quality evaluation of test cases in input space with respect to previously explored code regions.
- In order to control diffusivity of the test generation processes we define a feedback mechanism connecting current path exploration performance to the test generation module.
- We enable self-adaptive fuzzing behavior by adjusting the Lévy flight parameters according to feedback from dynamic instrumentation of the target executable.
- We implement the presented algorithm to show the feasibility of our approach.

The remainder of this paper is organized as follows. In Section II we discuss related work. In Section III we present necessary background on Lévy flights and show how to construct them in input space. We define a quality measure for generated test cases in Section IV and introduce our self-adapting fuzzing algorithm in Section V. Next, we give details regarding our implementation in Section VI and discuss properties, possible modifications, and expansions of the proposed algorithm in Section VII. The paper concludes with a short outlook in Section VIII.

II. Related work

The prevalent method used for binary vulnerability detection is random test generation, also called fuzzing. Here, inputs are randomly generated and injected into the target program with the aim to gain maximal code coverage in the execution graph and drive the program to an unexpected and exploitable state. There is a rich diversity of fuzzing tools available, each focusing on specialized approaches. Multiple taxonomies for random test generation techniques have been proposed, the most common is classification into mutational or generational fuzzing. Mutation fuzzers are unaware of the input format and mutate the whole range of input variables blindly. In contrast, generation fuzzers take the input format into account and generate inputs according to the format definition. For example, generation fuzzers can be aware of the file formats accepted by a program under test or the network protocol definition processed by a network stack implementation. We can further classify random test generation methods into black-box or white-box fuzzing, depending on the awareness of execution traces of generated inputs. We refer to [6] and [7] for a comprehensive account.

For definition of our quality measure for test cases we built upon executable code coverage strategies. The idea to generate program inputs that maximize execution path coverage in order to trigger vulnerabilities has been discussed in the field of test case prioritization some time ago, see e.g. [18] and [19] for a comparison of coverage-based techniques. Rebert et al. [20] discuss and compare methods to gain optimal seed selection with respect to fuzzing and their findings support our decision to select code coverage for evaluating the quality of test cases. The work of Cha et al. [21] is distantly related to a substep of our approach in the sense that they apply dynamic instrumentation to initially set the mutation ratio. However, they use completely different methods based on symbolic execution. Since symbolic preprocessing is very cost-intensive they further compute the mutation ratio only once per test.

Lévy flights have been studied extensively in mathematics and we refer to Zaburdaev et al. [22] and the references therein for a comprehensive introduction to this field. Very recently Chupeau et al. [23] connected Lévy flights to optimal search strategies and minimization of cover times.

III. Lévy Flights in Input Space

In this section we give the necessary background on Lévy flights and motivate their application. With this background we then define Lévy flights in input space.

A. *Lévy Flights*

Lévy flights are basically random walks in which step lengths exhibit power law tails. We aim for a short and illustrative presentation of the topic and refer to Zaburdaev et al. [22] for a comprehensive introduction. Pictorially if a particle moves stepwise in space while randomly choosing an arbitrary new direction after each step, it describes a Brownian motion. If in addition the step lengths of this particle vary after each step and are distributed according to a certain power low, it describes a Lévy flight.

Formally, Lévy processes comprise a special class of Markovian stochastic processes, i.e. collections of random variables

$$(L_t),\ t \in T \tag{1}$$

defined on a sample space Ω of a probability space $(\Omega, \mathcal{F}, P)$, mapping into a measurable space $(\Omega', \mathcal{F}')$, and indexed by a totally ordered set T. In our case Ω' refers to the discrete input space of the program and the index *time* T models the discrete iterations of test case generation, so we can assume $T = \mathbb{N}$. The process $(L_t)_{t \in T}$ is said to have *independent increments* if the differences

$$L_{t_2} - L_{t_1}, L_{t_3} - L_{t_2}, ..., L_{t_n} - L_{t_{n-1}} \tag{2}$$

are independent for all choices of $t_1 < t_2 < ... < t_n \in T$. The process $(L_t),\ t \in T$ is said to be *stationary*, if

$$\forall t_1, t_2 \in T, h > 0 :\ L_{t_1+h} - L_{t_1} \sim L_{t_2+h} - L_{t_2}, \tag{3}$$

i.e. increments for equal time intervals are equally distributed. A Lévy process is then formally defined to be a stochastic process having independent and stationary increments. The additional property

$$L_0 = 0\ a.s. \tag{4}$$

(i.e. almost surely) is sometimes included in the definition, but our proposed algorithm includes starting points other than the origin.

To construct a Lévy process $(L_n)_{n \in \mathbb{N}}$ we simply sum up independent and identically distributed random variables $(Z_n)_{n \in \mathbb{N}}$, i.e.

$$L_n := \sum_{i=1}^{n} Z_i. \tag{5}$$

The process $(L_n)_{n\in\mathbb{N}}$ is Markovian in the sense that

$$P(L_n = x_n | L_{n-1} = x_{n-1}, ..., L_0 = x_o) \tag{6}$$
$$= P(L_n = x_n | L_{n-1} = x_{n-1}), \tag{7}$$

which simplifies a practical implementation. If the distribution of step lengths in a Lévy process is heavy-tailed, i.e. if the probability is not exponentially bounded, we call the process a *Lévy flight*. Such processes generalize Brownian motion in that their flight lengths l are distributed according to the power law

$$p(l) \sim |l|^{-1-\alpha}, \tag{8}$$

where $0 < \alpha < 2$. They exhibit infinite variance

$$< l^2 >= \infty \tag{9}$$

which practically results in sometimes large jumps during search process. In fact, the ability to drive a particle very long distances within a single step gives Lévy flights their name. While Brownian motion is a suitable search strategy for densely distributed targets, Lévy flights are more efficient than Brownian motion in detecting widely scattered (software) bugs. Although there is much to say about the theoretical aspects of this class of stochastic processes we basically refer to the power law in equation (8) in the following. Smaller values of α yield a heavier tail (resulting in frequent long flights and super-diffusion) whereas higher values of α reveal a distribution with probability mass around zero (resulting in frequent small steps and sub-diffusion). In Section V we adapt α according to feedback information from dynamic instrumentation of the targeted binary.

As indicated in Section I Lévy flights are directly connected to the minimal time it takes to cover a given search domain. We refer to [23] for recent results regarding minimization of the mean search time for single targets.

B. Input Space Flights

Next we construct Lévy flights in the input space of binary executables under test. Therefore, assume the input to be a bit string of length N. If we simply wanted an optimal search through the input space without any boundary conditions, we would construct a one-dimensional Lévy flight in the linear space $\{0, ..., 2^N\}$. However, our aim is not input space coverage but execution code coverage of the binary under test. In this section we construct a stochastic process in input space with the properties we need for the main fuzzing algorithm presented in Section V.

First, we divide the input into n segments of size $m = \frac{N}{n}$ (assuming without loss of generality that N is a multiple of n). We then define two Lévy processes, one in the space of offsets $\mathcal{O} = \{1, ..., n\}$ and one in the space of segment values $\mathcal{S} = \{1, ..., 2^m\}$. With underlying probability spaces $(\Omega_1, \mathcal{F}_1, P_1)$ and $(\Omega_2, \mathcal{F}_2, P_2)$ we define the one-dimensional Lévy flights

$$L_t^1 : \ \Omega_1 \to \ \mathcal{O} \tag{10}$$
$$L_t^2 : \ \Omega_2 \to \ \mathcal{S} \tag{11}$$

with index space $t \in \mathbb{N}$ and corresponding power law distribution of flight lengths l

$$p_j(l) \sim |l|^{-1-\alpha_i}, \ j = 1, 2 \tag{12}$$

where $0 < \alpha_i < 2$. While $(L_t^1)_{t\in\mathbb{N}}$ performs a Lévy flight in the offset parameter space, $(L_t^2)_{t\in\mathbb{N}}$ performs Lévy flights within the segment space indicated by the offset. Regarding the initial starting point (L_0^1, L_0^2) we assume a given seed input. We choose an arbitrary initial offset $L_0^1 \in \mathcal{O}$ and set the initial value of L_0^2 according to the segment value (with offset L_0^1) of the seed input.

By setting different values of α we can control the diffusivity of the stochastic processes $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$. If we find a combination of offset and segment values of high quality the fuzzer should automatically explore nearby test cases, which is realized by higher values of $0 < \alpha_i < 2$. Similarly if the currently explored region within input space reveals low quality test cases, the fuzzer should automatically adapt to widen its search pattern by decreasing α. Therefore, we first have to define a quality measure for test cases.

IV. Quality Evaluation of Test Cases

In this section we define a quality measure for generated test cases. We aim for maximal possible code coverage in a finite amount of time, so we evaluate a single input by its ability to reach previously undiscovered execution paths. In other words, if we generate an input that drives the program under test to a new execution path, this input gets a high quality rating. Therefore we have to define a similarity measure for execution traces. We will then use this measure in Section V as feedback to dynamically adapt diffusivity of the test case generation process.

The field of test case prioritization provides effective methods for coverage-based rating (see [18] and [19] for a comparison). We adapt the method of prioritizing test cases by additional basic block coverage. As introduced in Section III we assume inputs for the program under test to be bit strings of size N and denote the space of all possible inputs as $\mathcal{I} = \{0, ..., 2^N\}$. Our challenge can then be formulated as follows. Given a subset of already generated input values $\mathcal{I}' \subset \mathcal{I}$, how do we measure the quality of a new input $x_0 \in \mathcal{I}$ with respect to maximal code coverage? For a given $x_0 \in \mathcal{I}$ let c_{x_0} denote the execution path the program takes for processing x_0. Intuitively we would assign a high quality rating to the new input x_0 if it drives the targeted program to a previously undiscovered execution path, i.e. if c_{x_0} differs significantly from all previously explored execution paths $\{c_x | x \in \mathcal{I}'\}$. To measure this path difference we take the amount of newly discovered basic blocks into account. Here we refer to a *basic block* as a sequence of machine instructions without branch instructions between block entry and block exit. Let $B(c_x)$ denote the set of basic blocks of execution path c_x. The set of newly discovered basic blocks while processing a new test case x_0 given already executed test cases $\mathcal{I}' \subset \mathcal{I}$ is then

$$B(c_{x_0}) \setminus \left(\bigcup_{x \in \mathcal{I}'} B(c_x) \right). \tag{13}$$

We define the number $E(x_0, \mathcal{I}')$ of these newly discovered blocks as

$$E(x_0, \mathcal{I}') := \left| B(c_{x_0}) \setminus \left(\bigcup_{x \in \mathcal{I}'} B(c_x) \right) \right|, \tag{14}$$

where $|A|$ denotes the number of elements within a set A. The number $E(x_0, \mathcal{I}')$ indicates the number of newly discovered basic blocks when processing x_0 with respect to the already known basic blocks executed by the test cases within $\mathcal{I}'$. Intuitively $E(x_0, \mathcal{I}')$ gives us a quality measure for input x_0 in terms of maximization of basic block coverage. In order to construct a feedback mechanism we will use a slightly generalized version of this measure to control diffusivity of the input generating Lévy processes in our fuzzing algorithm in Section V.

V. Fuzzing Algorithm

In this section we present the overall fuzzing algorithm. Our approach uses stochastic processes (i.e. Lévy flights as introduced in Section III) in the input space to generate test cases. To steer the diffusivity of test case generation we provide feedback regarding the quality of test cases (as defined in Section IV) to the test generation process in order to yield self-adaptive fuzzing.

We first prepend an example regarding the interplay between input space coverage and execution path coverage to motivate our fuzzing algorithm. Consider a program which processes inputs from an input space $\mathcal{I}$. Our aim is to generate a subset $\mathcal{I}' \subset \mathcal{I}$ of test cases (in finite amount of time) that yields maximal possible execution path coverage when processed by the target program. Further assume the program to reveal deep execution paths (covering long sequences of basic blocks) only for 3% of the inputs $\mathcal{I}$, i.e. 97% of inputs are inappropriate test cases for fuzzing. Since we initially cannot predict which of the test cases reveals high quality (determined by e.g. the execution path length or the number of different executed basic blocks), one strategy to reach good code coverage would be black-box fuzzing, i.e. randomly generating test cases within $\mathcal{I}$ hoping that we eventually hit some of the 3% high quality inputs. We could realize such an optimal search through input space with highly diffusive stochastic processes, i.e. Lévy flights as presented in Section III.

As mentioned above the Lévy flight hypotheses predicts an effective optimal search through input space due to their diffusivity properties. On the one hand this diffusivity guarantees us reaching the 3% with very high probability. On the other hand, once we have reached input regions within the 3% of high quality test cases, the same diffusivity also guarantees us that we will leave them very efficiently. This is why we need to adapt the diffusivity of the stochastic process according to the quality of the currently generated test cases. If the currently generated test cases reveal high path coverage, the Lévy flight should be localized in the sense that it reduces its diffusivity to explore nearby inputs. In turn, if the currently generated test cases reveal only little coverage, diffusivity should increase in order to widen the search for more suitable input regions. By instrumenting the binary under test and applying the quality evaluation of test cases introduced in Section IV we are able to feedback coverage information of currently explored input regions to the test case generation algorithm. In the following we construct a self-adaptive fuzzing strategy that automatically expands its search when reaching low quality input regions and focuses exploration when having the feedback of good code coverage.

a) Initial Seed: We start with an initial non-empty set of input seeds $X_0 \subset \mathcal{I}$. As described in Section III we assume the elements $x \in X_0$ to be bit strings of length N and divide each of them into n segments of size $m = \frac{N}{n}$ (assuming without loss of generality that N is a multiple of n). Practically the input seeds X_0 can be arbitrary files provided manually by the tester, they may not even be valid with regard to the input format of the program under test. We further set two initial diffusive parameters $0 < \alpha_1, \alpha_2 < 2$ and an initial offset $q_0 \in \{1, ..., n\}$.

b) Test Case Generation: The test case generation step takes as input a test case x_0, diffusion parameters α_1 and α_2, an offset number $q_0 \in \{1, ..., n\}$, and a natural number $k_{gen} \in \mathbb{N}$ of maximal test cases to be generated. It outputs a set X_{gen} of k_{gen} new test cases $X_{gen} \in \mathcal{I}$.

As introduced in Section III we refer to the offset space as $\mathcal{O} = \{1, ..., n\}$ and to the segment space as $\mathcal{S} = \{1, ..., 2^m\}$. We denote with $x_0(q_0)$ the segment value of input x_0 at offset q_0. For the Lévy flights

$$L_t^1 : \ \Omega_1 \to \ \mathcal{O} \tag{15}$$

in the offsets $\mathcal{O}$ and

$$L_t^2 : \ \Omega_2 \to \ \mathcal{S} \tag{16}$$

in $\mathcal{S}$ with flight lengths l distributed according to the power law

$$p_j(l) \sim |l|^{-1-\alpha_j}, \ j = 1, 2 \tag{17}$$

we set the initial conditions

$$L_0^1 = q_0 \text{ and} \tag{18}$$
$$L_0^2 = x_0(q_0), \tag{19}$$

respectively. Let $R(x_0, q_0, s_0)$ denote the bit string generated by replacing the value $x_0(q_0)$ of bit string x_0 at offset q_0 by a new value s_0. Both stochastic processes $(L_t^1)_{t \in \mathbb{N}}$ and $(L_t^2)_{t \in \mathbb{N}}$

are then simulated for k_{gen} steps to generate the k_{gen} new test cases

$$x_1 := R\big(x_0, L_0^1, L_1^2\big) \tag{20}$$
$$x_2 := R\big(x_1, L_1^1, L_2^2\big) \tag{21}$$
$$\dots$$
$$x_{j+1} := R\big(x_j, L_j^1, L_{j+1}^2\big) \tag{22}$$
$$\dots$$
$$x_{k_{gen}} := R\big(x_{k_{gen}-1}, L_{k_{gen}-1}^1, L_{k_{gen}}^2\big). \tag{23}$$

For simplicity of notation in this definition we identify the values L_t^j with their respective binary representations (as bit string). In words, we start with the initial test case x_0 and replace its segment content at offset $L_0^1 = q_0$ with the new value L_1^2, which is the value in segment space $\mathcal{S} = \{1, ..., 2^m\}$ that we get when taking a first random step with the Lévy flight $(L_t^2)_{t\in\mathbb{N}}$. This yields x_1. We get the next test case x_2 by considering the just generated x_1, setting the offset according to $(L_t^2)_{t\in\mathbb{N}}$, and then replacing the content of the segment indicated by this offset by a new segment value chosen by $(L_t^2)_{t\in\mathbb{N}}$. We proceed with this algorithm until the set

$$X_{gen} := \{x_1, ..., x_{k_{gen}}\} \tag{24}$$

of k_{gen} new test cases is generated.

c) Quality Evaluation: The quality evaluation step takes as input two sets of test cases $X_{gen}, \mathcal{I}' \subset \mathcal{I}$ and outputs a quality rating $\tilde{E}(X_{gen}, \mathcal{I}')$ of X_{gen} with respect to $\mathcal{I}'$. We already defined the number $E(x_0, \mathcal{I}')$ of newly discovered basic blocks for a single test case x_0 with respect to a given subset $\mathcal{I}' \subset \mathcal{I}$ in Equation (14). To generalize this definition to a quality rating $\tilde{E}(X_{gen}, \mathcal{I}')$ of a set of test cases X_{gen} (with respect to $\mathcal{I}'$) we define the mean

$$\tilde{E}(X_{gen}, \mathcal{I}') := |X_{gen}|^{-1} \sum_{x\in X_{gen}} E(x, \mathcal{I}'). \tag{25}$$

d) Adaptation of Diffusivity: The diffusivity adaptation step takes as input a quality rating $\tilde{E}(X_{gen}, \mathcal{I}') \in \mathbb{N}$, two parameters $b_1, b_2 \in \mathbb{R}^+$ (controlling the switching behavior from sub-diffusion to super-diffusion) and outputs two adapted parameters $0 < \alpha_1, \alpha_2 < 2$, which according to the power law (17) regulate the diffusivity of the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$.

Our aim (as motivated at the beginning of this section) is to adapt the diffusion parameters in such a way that the algorithm automatically focuses its search (by decreasing diffusivity of the generating Lévy flights) when generating high quality (i.e. high coverage) test cases and in turn automatically widens its search (by increasing diffusivity) in the case of low quality (i.e. low coverage) test cases. As discussed in Section III we can control diffusivity by setting suitable values of α_1 and α_2. Smaller diffusivity parameters result in frequent long flights and super-diffusion whereas higher parameters reveal frequent small steps and sub-diffusion. To achieve this we select a monotonically increasing function $f : \mathbb{R} \to (0, 2)$ with $f(0) \leq \epsilon$ (for $\epsilon > 0$ sufficiently small) and $\lim_{t\to\infty} f(t) = 2$. Any such function will provide self adaptation of diffusivity of the Lévy flights and we simply choose two functions

$$f_i(t) := \frac{1}{1 + e^{b_i - t}}, \; i = 1, 2 \tag{26}$$

where $b_i \in \mathbb{R}^+$ are fixed parameters that determine at which point within the quality rating spectrum (i.e. at which mean number of newly discovered basic blocks) the search behavior of $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ switches from sub-diffusion to super-diffusion. With this function we adapt diffusivity to

$$\alpha_i = f\big(\tilde{E}(X_{gen}, \mathcal{I}')\big), \; i = 1, 2. \tag{27}$$

The next iteration of test case generation is then executed with adapted Lévy flights.

e) Test Case Update: This step takes as input two sets of test cases $X_{old}, X_{gen} \subset \mathcal{I}$ and outputs an updated set of test cases X_{new}. During the fuzzing process we generate a steady stream of new test cases which we directly evaluate with respect to the set of previously generated inputs (as discussed in the quality evaluation step). However, if we archive every single test case and for each generation step evaluate the k_{gen} currently generated new test cases against the whole history of previously generated test cases, fuzzing speed decays constantly with increasing duration of the fuzzing campaign. Therefore we define an upper bound $k_{max} \in \mathbb{N}$ of total test cases that we keep for quality evaluation of new test cases. Small values of k_{max} may cause the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ to revisit already explored input regions without being adapted (by decreasing the parameters α_i) to perform super-diffusion and widen their search behavior. However, this causes no problem due to the Lévy flight hypothesis (discussed in Section I).

The update of X_{old} with X_{gen} simply follows a *first in first out* strategy. Initially if $|X_{old}| + |X_{new}| < k_{max}$ we append all newly generated test cases so that $X_{new} = X_{old} \cup X_{gen}$. Otherwise we first delete the oldest k_{old} entries in X_{old}, where

$$k_{old} = |X_{old}| + |X_{new}| - k_{max}, \tag{28}$$

and then take the union.

f) Joining the Pieces: Now that we have presented all individual parts we can combine them. The overall fuzzing algorithm is depicted in Figure 1.

The initial seed generation step outputs a non-empty set of test cases $X_0 \subset \mathcal{I}$, two diffusivity parameters α_1 and α_2, and an initial offset q_0. The inputs X_0 are added to the list of test cases X_{all}. Then the fuzzer enters the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update. The first step within the loop (referred to as *Last*(X_{all})) is selecting the most recently added test case x_0 in X_{all}, which will then be used as initial condition in the generation step. Starting at $L_0^1 = q_0$ and $L_0^2 = x_0(q_0)$ the Lévy flights $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ generate the set of new inputs X_{gen} by diffusing through input space with diffusivity α_1 and α_2, respectively. The quality of X_{gen} is then evaluated against the previous test cases in X_{all}. Depending on the quality rating outcome, the diffusivity of $(L_t^1)_{t\in\mathbb{N}}$ and $(L_t^2)_{t\in\mathbb{N}}$ is then

```
**Input:** Parameters $b_1,\ b_2,\ k_{gen},\ k_{max}$

$X_{all} = \emptyset$
$X_0,\ \alpha_1,\ \alpha_2,\ q_0 \leftarrow$ Seed()
append $X_0$ to $X_{all}$
**do:**
    $x_0 \leftarrow \mathrm{Last}(X_{all})$
    $X_{gen} \leftarrow \mathrm{Gen}(x_0, \alpha_1, \alpha_2, q_0, k_{gen})$
    $\tilde{E} \leftarrow \mathrm{Eval}(X_{gen}, X_{all})$
    $\alpha_1, \alpha_2 \leftarrow \mathrm{Adapt}(\tilde{E}, b_1, b_2)$
    $X_{all} \leftarrow \mathrm{Update}(X_{gen}, X_{all}, k_{max})$

**while** (true)
```

Fig. 1. Overall fuzzing algorithm with parameters $b_1,\ b_2,\ k_{gen}$, and k_{max}. After initial seed generation the fuzzer enters the loop of test case generation, quality evaluation, adaptation of diffusivity, and test case update.

adapted correspondingly by updating α_1 and α_2 according to the sigmoid functions f_i in Equations (26). Then the current list of test cases X_{all} is updated with the just generated set X_{gen} and the fuzzer continues to loop.

Regarding complexity of the fuzzing algorithm we note that all of the individual parts are processed efficiently in the sense that their time complexity is bound by a constant. Especially the evaluation step *Eval()* is designed to scale: In the first iterations of the loop the cost of evaluating X_{gen} against X_{all} is bound by $\mathcal{O}(|X_{all}|^2)$. To counter this growth we defined an upper bound $k_{max} \in \mathbb{N}$ for $|X_{all}|$ in the *test case update* step above.

VI. Implementation

To show the feasibility of our approach we implemented a prototype for the proposed self-adaptive fuzzing algorithm (as depicted in Figure 1). Our implementation is based on Intel's dynamic instrumentation tool Pin [24] and a custom Lévy flight simulation programmed in the statistical computing language R [25]. While test cases are generated in the R module, evaluation of current path exploration performance is realized in the custom Pin tool. A simple Python script handles the quality feedback from dynamic instrumentation to the test case generating R module.

In our implementation we omit the first step *Last*(X_{all}) within the loop and instead always keep the current position of the processes $(L_t^i)_{t\in\mathbb{N}}$. This is due to the construction of new test cases in Equations (20)-(23) so that the last test case within X_{all} is simply the most recently generated $x_{k_{gen}}$ which will be used as starting position within the subsequent loop iteration. Therefore it suffices to stop the Lévy flights after k_{gen} steps, save their current position, and proceed with adapted diffusivity parameters in the subsequent invocation of the *Gen()* function.

VII. Discussion

In this section we discuss properties, possible modifications, and expansions of our proposed fuzzing algorithm.

As demonstrated in Section V our algorithm is self-adaptive in the sense that it automatically focuses its search when reaching high quality regions in input space and widens exploration in case of low quality input regions. One possible pitfall of such a self-adaptive property is the occurence of attracting regions: If the Lévy flights $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ enter regions of high quality and get the response from the quality evaluation step to focus their search (by decreasing their diffusivity), an improper quality rating mechanism might cause the Lévy flights to stay there forever. However, our evaluation method (as defined in Section IV) avoids this by favoring test cases that lead the target binary to execute undiscovered basic blocks and in turn devaluates inputs that lead to already known execution paths. Therefore, if the test case generation module gets feedback that it is currently exploring a region of high quality it focuses its search as long as new execution paths are detected. As soon as exploration of new execution paths stagnates, the feedback from the evaluation module switches to a low rating. Such a negative feedback again increases diffusivity according to Equations (26) and (27), which again causes the processes $(L_t^i)_{t\in\mathbb{N}}$ $(i = 1, 2)$ to diffuse into other regions of the input space.

One main modification of our algorithm would be interchanging the aim of maximizing code coverage with an adequate objective. In Section IV we defined a quality measure for generated test cases based on the number of new basic blocks we reach with those inputs. Although this is the most common strategy when searching for bugs in a target program of unknown structure, we could apply other objectives. For example, we could aim for triggering certain data flow relationships, executing preferred regions of code, or reach a predefined class of statements within the code. Our fuzzing algorithm is modular and flexible in that it allows to interchange the quality measure according to different testing objectives. More examples of such testing objectives are discussed in the field of test case prioritization (e.g. in [18] and [19]).

VIII. Conclusion

Inspired by moving patterns of foraging animals we introduce the first self-adaptive fuzzer based on Lévy flights. Just like search patterns in biology have evolved to optimal foraging strategies due to natural selection, so have evolved mathematical models to describe those patterns. Lévy flights are emerging as successful models for describing optimal search behavior, which leads us to their application of hunting bugs in binary executables. By defining corresponding stochastic processes within the input space of the program under test we achieve an effective new method for test case generation. Further, we define an algorithm that dynamically controls diffusivity of the defined Lévy flights depending on actual quality of generated test cases. To achieve this we construct a measure of quality for new test cases that takes already explored execution paths into account. During fuzzing the quality of actually generated test cases is constantly forwarded to the test case generating Lévy flights. High quality test case generation with respect to path coverage causes the Lévy flight to enter sub-diffusion and focus its search on nearby inputs, whereas a low quality rating results in super-diffusion and expanding search behavior. This feedback loop yields a fully self-adaptive fuzzer. Our proposed algorithm is modular in the sense that it allows integration of other fuzzing goals beyond code coverage, which is subject to future work.

References

[1] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[2] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International journal on software tools for technology transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *European Software Engineering Conference*, 2005, pp. 263–272.

[4] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *ACM SIGPLAN Notices*, vol. 40. ACM, 2005, pp. 213–223.

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, p. 40, Mar. 2012.

[6] A. Takanen, J. D. Demott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[7] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[8] F. Lenz, T. C. Ings, L. Chittka, A. V. Chechkin, and R. Klages, "Spatiotemporal dynamics of bumblebees foraging under predation risk," *Physical review letters*, vol. 108, no. 9, p. 098103, 2012.

[9] G. M. Viswanathan, "Ecology: Fish in Lévy-flight foraging," *Nature*, vol. 465, no. 7301, pp. 1018–1019, 2010.

[10] N. E. Humphries, N. Queiroz, J. R. Dyer, N. G. Pade, M. K. Musyl, K. M. Schaefer, D. W. Fuller, J. M. Brunnschweiler, T. K. Doyle, J. D. Houghton *et al.*, "Environmental context explains Lévy and Brownian movement patterns of marine predators," *Nature*, vol. 465, no. 7301, pp. 1066–1069, 2010.

[11] D. Austin, W. D. Bowen, and J. I. McMillan, "Intraspecific variation in movement patterns: modeling individual behaviour in a large marine predator," *Oikos*, vol. 105, pp. 15–30, 2004.

[12] G. Ramos-Fernández, J. L. Mateos, O. Miramontes, G. Cocho, H. Larralde, and B. Ayala-Orozco, "Lévy walk patterns in the foraging movements of spider monkeys (ateles geoffroyi)," *Behavioral Ecology and Sociobiology*, vol. 55, no. 3, pp. 223–230, 2004.

[13] G. M. Viswanathan, V. Afanasyev, S. Buldyrev, E. Murphy, P. Prince, H. E. Stanley *et al.*, "Lévy flight search patterns of wandering albatrosses," *Nature*, vol. 381, no. 6581, pp. 413–415, 1996.

[14] A. Mrell, J. P. Ball, and A. Hofgaard, "Foraging and movement paths of female reindeer: insights from fractal analysis, correlated random walks, and Lévy flights," *Canadian Journal of Zoology-revue Canadienne De Zoologie*, vol. 80, pp. 854–865, 2002.

[15] O. Bénichou, C. Loverdo, M. Moreau, and R. Voituriez, "Intermittent search strategies," *Reviews of Modern Physics*, vol. 83, no. 1, p. 81, 2011.

[16] T. H. Harris, E. J. Banigan, D. A. Christian, C. Konradt, E. D. T. Wojno, K. Norose, E. H. Wilson, B. John, W. Weninger, A. D. Luster *et al.*, "Generalized Lévy walks and the role of chemokines in migration of effector CD8+ T cells," *Nature*, vol. 486, no. 7404, pp. 545–548, 2012.

[17] G. M. Viswanathan, M. G. Da Luz, E. P. Raposo, and H. E. Stanley, *The physics of foraging: an introduction to random searches and biological encounters*. Cambridge University Press, 2011.

[18] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *International Symposium on Software Reliability Engineering*, 2003, pp. 442–456.

[19] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *International Conference on Software Maintenance*, 1999, pp. 179–188.

[20] 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.

[21] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy (S&P)*, May 2015, pp. 725–741.

[22] V. Zaburdaev, S. Denisov, and J. Klafter, "Lévy walks," *Rev. Mod. Phys.*, vol. 87, pp. 483–530, Jun 2015.

[23] M. Chupeau, O. Bénichou, and R. Voituriez, "Cover times of random searches," *Nature Physics*, 2015.

[24] 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 *ACM Sigplan Notices*, vol. 40. ACM, 2005, pp. 190–200.

[25] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0.