SLIME: Program-Sensitive Energy Allocation for Fuzzing

Chenyang Lyu
Zhejiang University
China
puppet@zju.edu.cn

Hong Liang
Zhejiang University
China
hongliang@zju.edu.cn

Shouling Ji*
Zhejiang University
China
sji@zju.edu.cn

Xuhong Zhang
Zhejiang University
China
zhangxuhong@zju.edu.cn

Binbin Zhao
Georgia Institute of Technology
USA
binbin.zhao@gatech.edu

Meng Han
Binjiang Institute of Zhejiang University & Zhejiang University
China
mhan@zju.edu.cn

Yun Li
Huawei Technologies Co., Ltd.
China
liyun5@huawei.com

Zhe Wang
Institute of Computing Technology, Chinese Academy of Sciences.
China
wangzhe12@ict.ac.cn

Wenhai Wang
Zhejiang University
China
zdzzlab@zju.edu.cn

Raheem Beyah
Georgia Institute of Technology
USA
rbeyah@ece.gatech.edu

ABSTRACT

The energy allocation strategy is one of the most popular techniques in fuzzing to improve code coverage and vulnerability discovery. The core intuition is that fuzzers should allocate more computational energy to the seed files that have high efficiency to trigger unique paths and crashes after mutation. Existing solutions usually define several properties, e.g., the execution speed, the file size, and the number of the triggered edges in the control flow graph, to serve as the key measurements in their allocation logics to estimate the potential of a seed. The efficiency of a property is usually assumed to be the same across different programs. However, we find that this assumption is not always valid. As a result, the state-of-the-art energy allocation solutions with static energy allocation logics are hard to achieve desirable performance on different programs.

To address the above problem, we propose a novel program-sensitive solution, named *SLIME*, to enable adaptive energy allocation on the seed files with various properties for each program. Specifically, SLIME first designs multiple property-aware queues, with each queue containing the seed files with a specific property. Second, to improve the return of investment, SLIME leverages a customized Upper Confidence Bound Variance-aware (UCB-V) algorithm to statistically select a property queue with the most estimated reward, i.e., finding the most new unique execution paths and crashes. Finally, SLIME mutates the seed files in the selected property queue to perform property-adaptive fuzzing on a program. We evaluate SLIME against the state-of-the-art open source fuzzers AFL, MOpt, AFL++, AFL++HIER, EcoFuzz, and TortoiseFuzz on 9 real-world programs. The results demonstrate that SLIME discovers 3.53×, 0.24×, 0.62×, 1.54×, 0.88×, and 3.81× more unique vulnerabilities compared to the above fuzzers, respectively. We will open source the prototype of SLIME to facilitate future fuzzing research.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Fuzzing, Data-driven technique, Vulnerability discovery

ACM Reference Format:
Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. 2022. SLIME: Program-Sensitive Energy Allocation for Fuzzing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534385

1 INTRODUCTION

Mutation-based fuzzing is one of the most popular solutions to automatically discover vulnerabilities in a program. The general process of a mutation-based fuzzer is to mutate seeds with random operators, and then use the generated test cases to test a program

*Shouling Ji is the corresponding author.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534385

for triggering abnormal behaviors. However, the fuzzing performance is usually limited in practice under the above straightforward idea. Then, following the intuition that testing more paths is likely to discover more unique crashes/vulnerabilities, coverage-based fuzzing is proposed with the goal of improving path exploration in the fuzzing process [11, 32, 33, 57]. Specifically, a coverage-based fuzzer first stores the seeds, each of which triggers a unique execution path of the target program, in a queue. Then, it achieves better fuzzing performance by allocating mutation energy to these seeds. Recently, many researches point out that the seeds of a coverage-based fuzzer have different efficiency in generating *interesting test cases* [30, 49, 50], i.e., the ones that can trigger new unique execution paths or crashes on a target program. Therefore, a carefully designed energy allocation strategy is necessary for effectively generating more interesting test cases. Towards this, multiple energy allocation strategies are proposed, including mutating more times on the seeds with some specific properties [2, 32], focusing on exploring deeper paths [40], mutating more times on the seeds with higher transition probabilities, i.e., the seeds that are more likely to trigger different execution paths after mutation [8, 56], etc.

However, we observe that the seeds with better performance on the defined properties do not always have high efficiency to generate interesting test cases given a program in practice. Thus, existing energy allocation solutions cannot achieve the optimal efficiency on different programs due to the static energy allocation logics. To further demonstrate our observation, we conduct a case study (see the details in Section 2.4), which leads to the following main observation: the seeds with the same property have different efficiency on different programs. Thus, the seeds with the defined key properties cannot achieve the best fuzzing efficiency across different programs, i.e., existing static energy allocation solutions may yield poor performance in practice. Therefore, a program-sensitive energy allocation solution, which adaptively allocates appropriate energy to the seeds with different properties for each target program, is demanded to improve the fuzzing performance.

Towards this, we have the following main challenges: 1) how to define properties that can effectively classify seed files and contain the real-time efficient ones for each target program? 2) how to design a framework that adaptively finds seeds with each property and can periodically update them if necessary? and 3) how to estimate the potential of finding new unique paths and crashes for the seeds with a property and allocate mutation energy to them?

To overcome the above challenges, we present *SLIME* to achieve program-sensitive energy allocation for different programs. To be specific, we propose to define properties from three perspectives related to seed diversity, and identify 17 example properties correspondingly. Then, we design an iterative framework for SLIME to periodically update seeds with each property. In each iteration, SLIME first mutates seeds in the original queue to evaluate and decide whether this seed has the above properties. Second, SLIME constructs a new property queue to store the seeds for each specific property. Thus, a seed with multiple properties will be stored in multiple property queues. Third, SLIME leverages a customized Upper Confidence Bound Variance-aware (UCB-V) algorithm to estimate the potential reward of each property queue, based on the property queue's unique path and crash discovery performance so far. Finally, SLIME statistically selects the property queues guided by the estimated reward and mutates the seeds in these queues to achieve program-sensitive energy allocation.

As a general energy allocation solution, SLIME can be applied to most existing mutation-based fuzzers to enhance their energy allocation logics. In this paper, we implement SLIME based on MOpt [32], which is one of the most popular fuzzers with a static energy allocation component. We compare the fuzzing performance of SLIME with the state-of-the-art fuzzers, including AFL [2], MOpt, AFL++ [17], AFL++HIER [50], EcoFuzz [56], and TortoiseFuzz [52], on 9 representative real-world programs. In total, SLIME finds 3.53×, 0.24×, 0.62×, 1.54×, 0.88×, and 3.81× more unique vulnerabilities reported by AddressSanitizer than AFL, MOpt, AFL++, AFL++HIER, EcoFuzz, and TortoiseFuzz, respectively.

In summary, we make the following contributions.

- We conduct a preliminary case study to demonstrate that the seeds, which have a better performance on the key properties defined by the current energy allocation strategies, do not always have high efficiency given different programs in practice.
- Motivated by our case study, we propose a program-sensitive solution SLIME to adaptively allocate energy to the seeds with different properties, guided by the potential reward estimated by a customized Upper Confidence Bound Variance-aware (UCB-V) algorithm on a program. In particular, we design a flexible framework to implement property queue construction and property-adaptive energy allocation, and propose a new data structure for the original queue to better store property values for each seed.
- We evaluate SLIME compared with 6 state-of-the-art fuzzers on 9 real-world programs. The results show that SLIME performs better than others in terms of vulnerability discovery and edge coverage. We also use the standardized benchmark *FuzzBench* to show the significant coverage performance of SLIME. Furthermore, we utilize the published Common Vulnerabilities and Exposures (CVE) IDs as ground truth, and demonstrate the effectiveness and efficiency of SLIME to find serious vulnerabilities. We experimentally analyze the contribution of each main part of SLIME to the fuzzing performance.
- We open source SLIME at https://github.com/diewufeihong/SLIME to facilitate future fuzzing research.

2 BACKGROUND AND MOTIVATION

2.1 Mutation-based Fuzzing

The core idea of mutation-based fuzzing is to trigger the abnormal behaviors of a target program with randomly mutated test cases. Specifically, a general workflow is as follows. A fuzzer 1) constructs a seed queue to store seeds; 2) mutates a seed in the seed queue to generate new test cases and test the program; 3) stores the test cases locally that trigger abnormal behaviors of the program; and 4) goes to step 2) to iteratively perform the fuzzing process. However, the performance of a mutation-based fuzzer is restricted by the simple execution workflow. To improve fuzzing efficiency, existing works focus on improving the coverage of a mutation-based fuzzer, since a higher coverage means triggering more unique paths, which implies a higher likelihood to find more unique crashes/vulnerabilities.

2.2 Coverage-based Fuzzing

The focus of coverage-based fuzzing is to store a unique seed for each different execution path. In addition to the above steps of a

mutation-based fuzzer, a coverage-based fuzzer maintains a seed queue to store the unique seeds with different path coverage. To trigger new execution paths around the unique path triggered by a seed, the fuzzer generates new test cases based on this seed. For each execution to test a program, if the generated test case triggers a new unique execution path, the fuzzer stores the test case into the queue as a new seed. In this paper, we mainly focus on edge coverage to identify unique paths, which is widely implemented in the state-of-the-art coverage-based fuzzers and balances the trade-off between the identification accuracy and computational overhead.

In order to deduplicate execution paths with edge coverage, an edge coverage guided fuzzer constructs the following framework. 1) The fuzzer instruments a target program and assigns a random value to each basic block in the Control Flow Graph (CFG). After instrumentation, if the program executes an edge between two basic blocks, the program calculates an edge hash according to the blocks' assigned values. Thus, a unique edge in CFG is represented by a unique edge hash as calculated above. Then, the program utilizes each byte in a bitmap named `trace_bits`, whose byte index is edge hash, to count the execution times for each unique edge. For each triggered edge in an execution path, the instrumented program adds 1 on the corresponding byte of `trace_bits`; 2) To measure the edge coverage of an execution path triggered by a test case, the fuzzer leverages the shared memory to read `trace_bits` from the instrumented program. Then, to use different bits to mark different ranges of execution times in a byte of `trace_bits`, the fuzzer initializes `trace_bits` to 0, and turns one bit in one byte to 1 according to the execution times of each triggered edge; 3) Therefore, when using a test case to test a program, the fuzzer obtains the corresponding `trace_bits`. A new bit in `trace_bits` becoming 1 indicates that a new unique execution path is triggered.

Based on the edge coverage statistics provided by `trace_bits`, coverage-based fuzzers can store the seeds, which trigger unique execution paths, into the queue with low computational overhead. To further improve the fuzzing performance, as one of the research directions in coverage-based fuzzing, energy allocation strategies are proposed to spend more computational energy on the seeds that seem more likely to generate interesting test cases.

2.3 Energy Allocation Strategies

The core idea of energy allocation is to spend more energy (i.e., execution time and computing power) on mutating the seed that is more efficient to trigger unique paths and crashes after mutation. Specifically, most state-of-the-art works 1) propose their key properties to estimate the potential of seeds, and 2) design the corresponding algorithms based on the key properties to allocate different energy to each seed. For instance, AFL allocates more mutation energy to the seeds that have faster execution speed, trigger more bits on their `trace_bits`, and are discovered at a later time [2]. Based on AFL, *AFLFast* models the coverage exploration as the state space transition in a Markov chain. Then, AFLFast allocates more mutation energy to the seeds that trigger low-frequency state spaces and have higher transition probabilities, i.e., the probability that a seed triggers different execution paths after mutation [8]. Recently, *EcoFuzz* leverages the adversarial multi-armed bandit model to prioritize and allocate more energy to the seeds that have higher transition probabilities with fewer executions [56].

In summary, existing energy allocation solutions allocate more energy to the seeds that have better performance on their key properties, respectively. However, *the seeds with these key properties do not always have high efficiency to trigger unique paths and crashes given different programs in practice*, and thus the defined key properties would not be suitable to estimate the seeds' potential on different target programs. Different from the existing energy allocation solutions, the main idea of SLIME is to figure out the real-time efficient properties on each program, and adaptively allocate more mutation energy to the seeds with these high efficiency properties.

2.4 Motivation

To illustrate that *the efficiency of seeds with the same property is not always consistent across different target programs*, we conduct a case study to show the fuzzing efficiency of seeds with different properties. The investigated properties are defined as follows.

Definition 1 `fast`. *A seed with the* `fast` *property means that it has a short execution time when testing a program. We sort the seeds in the queue according to the execution time from short to long, and select the front ranked files as the ones with the* `fast` *property.*

Definition 2 `slow`. *The seeds with the* `slow` *property are the front ranked files which are sorted by the execution time from long to short.*

Definition 3 `long`. *The seeds with the* `long` *property mean that they have larger file sizes than other files in the queue.*

Definition 4 `short`. *Contrary to the* `long` *property, the size of the seed with the* `short` *property is smaller than others.*

Definition 5 `depth`. *The* `depth` *of a new seed is the* `depth` *of its parent seed plus 1. For instance, the* `depth` *of a seed in the initial seed set is 0. Then, the* `depth` *of a new seed generated from this is 1. Therefore, a seed with the* `depth` *property means that it is generated based on more rounds of mutations than other seed files.*

Definition 6 `cmp_const_num`. *The seeds with the* `cmp_const_num` *property mean that there are more solved branch constraints in their execution paths.*

Definition 7 `untouch_num`. *The seeds with the* `untouch_num` *property mean that they have more untouched neighbor basic blocks.*

Definition 8 `bit_num`. *If a seed has the* `bit_num` *property, it passes through more unique edges during execution and triggers more bits in its* `trace_bits` *compared to other seeds.*

Definition 9 `loop_num`. *If a seed triggers more program loops in the execution process, it is more likely to have the* `loop_num` *property.*

With the same experiment settings as in Section 4.1, we employ MOpt to fuzz `gdk`, `objdump` and `pdfimages`, three widely-used programs in fuzzing [5, 29, 32], for 48 hours, each of which is repeated 4 times. When finishing the fuzzing process, we 1) sort the seeds according to the definition of each aforementioned property; 2) select the top 40% files as the ones with the specific property; 3) count the total number of unique paths and crashes which are triggered by the selected seeds after mutation; 4) count the total execution times of these selected files; and 5) calculate the fuzzing efficiency of a property as the total number of unique paths and crashes divided by the total execution times. The results are shown in Fig. 1, and we have the following observations.

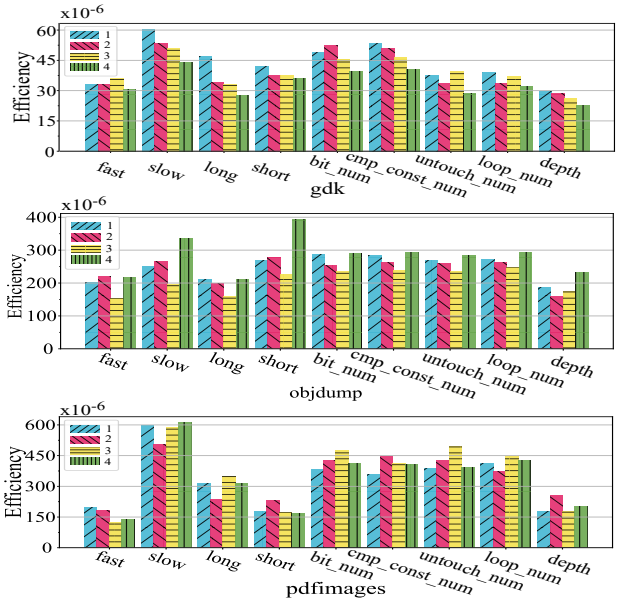

Figure 1: The unique path and crash discovery efficiency of the seeds with different properties in 4 trials. Each color represents one trial on a program. X-axis: The different properties. Y-axis: The efficiency of a specific property.

The seeds with different properties have different efficiency on a program. For instance, the seeds with the `slow`, `bit_num` and `cmp_const_num` properties have higher fuzzing efficiency than the files with other properties on `gdk`. Moreover, the seeds with the same property have stable efficiency in different trials on a program. This demonstrates the validity of property-based energy allocation solutions in fuzzing. However, the implicit assumption in existing energy allocation solutions that the seeds with the same property also achieve high efficiency across different programs is improper.

The seeds with the same property have different efficiency on different programs. As shown in Fig. 1, the seeds with the `untouch_num` property have high efficiency on `pdfimages`, but have low efficiency on `gdk`. The seeds with the `short` property have acceptable efficiency on `objdump`, while they perform poorly on `pdfimages`. In addition, the seeds with the `fast` and `depth` properties perform worse than others on `pdfimages`, which is contrary to the state-of-the-art energy allocation solutions. Therefore, a program-sensitive energy allocation solution is demanded to improve the fuzzing performance given different programs in practice.

In summary, we find that 1) the seeds with different properties perform differently on a program; and 2) the seeds with the same property perform differently on different programs. However, existing energy allocation solutions consider that the seeds having the defined key properties are always efficient to find unique paths and crashes, while ignoring the correlation between different properties and different programs. This may lead to poor performance in practice. Thus, it is necessary to develop a program-sensitive energy allocation solution, which adaptively allocates energy according to the fuzzing efficiency of the properties for each target program.

3 DESIGN OF SLIME

3.1 Framework of SLIME

To solve the problems in existing energy allocation strategies and achieve the program-sensitive energy allocation, SLIME is designed to adaptively allocate mutation energy to the seeds with different properties according to each property's efficiency given a target program. Specifically, SLIME first needs to mutate each seed in the seed queue and record its performance on different properties. Second, SLIME clusters the seeds with the same property, and then allocates appropriate energy to mutate these seeds according to the property's fuzzing efficiency. To periodically update seeds' property and recluster them, the above process is performed iteratively.

To achieve the above design, we construct the framework of SLIME as shown in Fig. 2. Specifically, to cluster the seeds with the same property, we design a *property queue* to store seeds for each property. Correspondingly, we design the *original queue* to store all the seeds discovered in the fuzzing process.

To test a target program and record a newly discovered interesting test case's performance on different properties, SLIME leverages the `Fuzzing Engine` to mutate a seed and detect whether triggering new execution paths and crashes. Then, if SLIME finds a new interesting test case which triggers a new execution path, it utilizes the `Property Record` to 1) store the test case into the original queue as a new seed and 2) record the seed's performance on each defined property. In particular, we present 17 kinds of properties from 3 perspectives related to seed diversity (see Section 3.2). Noting that more properties can be easily included in SLIME. For instance, SLIME can leverage vulnerable locations reported by static analysis tools as a property, and allocates more energy to the seeds triggering these vulnerable locations in order to improve vulnerability discovery.

To adaptively update seeds' property and periodically reconstruct each property queue, we design 3 iterative stages in SLIME: the `Exploration` Stage, the `Update` Stage, and the `Exploitation` Stage. The details of these stages are as follows.

The `Exploration` Stage: As shown in Fig. 2, SLIME enters the `Exploration` Stage at the beginning of fuzzing, in which SLIME mutates all the seeds in the original queue. For each new interesting test case that triggers a new unique path, SLIME stores it in the original queue as a new seed, and records its property values. To save computational overhead and maximize SLIME's fuzzing efficiency at the beginning of the fuzzing process, we define two ending conditions of the `Exploration` Stage: 1) for the first time, if the unique path and crash discovery efficiency of SLIME drops to a preset threshold, SLIME finishes the `Exploration` Stage and enters the `Update` Stage; and 2) after the first time, SLIME mutates all the seeds in the original queue for a fixed number of cycles in the `Exploration` Stage, and then enters the `Update` Stage.

Different from the design of existing fuzzers [2, 32, 42, 56] that only store and mutate the first test case triggering a unique execution path, SLIME leverages the `Seed Replacement` to replace the old seed in the original queue with a higher quality one for each unique path. To achieve this, we first design a new data structure of the original queue to implement the `Seed Replacement` in SLIME, whose details are shown in Section 3.3. Then, after identifying most efficient properties in the `Exploitation` Stage, SLIME can quantify

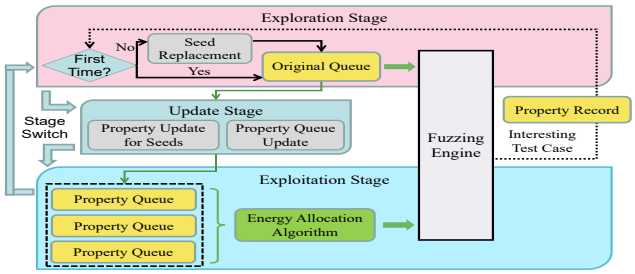

Figure 2: The framework of SLIME.

the estimated quality for each test case based on its performance on these efficient properties. Finally, in the `Exploration` Stage, if a seed has been fuzzed and has never discovered any new unique path or crash, it can be replaced by a test case that triggers the same path and has a higher estimated quality. The details of the `Seed Replacement` are shown in Section 3.4.

The `Update` Stage: In the `Update` Stage, SLIME first leverages the `Property Update for Seeds` to update property values of each seed before constructing property queues, whose reasons are as follows. Although SLIME records the value of each property for a new discovered seed with the `Property Record`, its recorded property values can be changed during the fuzzing process. For instance, the value of the `untouch_num` property will be changed because some untouched neighboring basic blocks are explored as the testing goes on. Therefore, SLIME checks all the seeds in the original queue to update their property values in this step.

Second, in the `Property Queue Update`, SLIME traverses all the seeds in the original queue and reconstructs all the property queues. Each property queue is sorted from high to low by the performance of seeds on this property. Specifically, when checking the value of a specific property for a seed, if 1) the corresponding property queue does not reach the maximum length or 2) the current seed has better performance on this property according to the definition than the last one in this property queue, SLIME adds the seed to this property queue. Then, if the property queue exceeds the maximum length after adding a new seed, SLIME removes the last seed with the worst performance in this property queue. Thus, after traversing all the seeds once, SLIME achieves to construct all the property queues satisfying their definitions, respectively.

The `Exploitation` Stage: After reconstructing all the property queues, SLIME enters the `Exploitation` Stage to perform energy allocation. To be specific, SLIME 1) leverages a property-adaptive energy allocation algorithm to estimate the potential of discovering interesting test cases of each property queue, and 2) statistically selects a property queue according to the estimated potential and mutates the seeds in the property queue. More details are introduced in Section 3.5. The queue selection process repeats a fixed number of times in the `Exploitation` Stage, then SLIME enters the `Exploration` stage and starts the next iteration.

3.2 The Properties of SLIME

To find the real-time key properties contributing to the discovery of interesting test cases given a target program, SLIME needs to define enough properties, each of which can effectively classify seeds. To achieve this, we analyze the methods of existing coverage-based fuzzers [17, 19, 32, 56] to distinguish seeds, and realize that the properties related to seed diversity can be grouped into 3 perspectives: 1) the basic properties related to seeds, 2) the properties of basic blocks triggered by seeds, and 3) the properties of `trace_bits` triggered by seeds. Thus, in our design, we define 17 kinds of properties based on the 3 perspectives, whose details are as follows.

1) The basic properties include `fast`, `slow`, `long`, `short`, and `depth` as shown in Section 2.4, as well as the following 3 properties.

Definition 10 `interesting`. *The seeds with the* `interesting` *property are sorted from high to low according to the number of interesting test cases generated from these seeds.*

Definition 11 `edge_change_eff`. *If a seed changes its execution path after mutation, its edge change number is increased by 1. We calculate the edge change efficiency (the edge change number divided by the number of executions) for each seed, and then select the front ranked seeds as the ones with the* `edge_change_eff` *property.*

Definition 12 `rare_file`. *If a seed has been mutated fewer times than others, it can be regarded as a rare file during the fuzzing process and will be added to the* `rare_file` *property queue.*

2) The properties of basic blocks triggered by seeds include `cmp_const_num` and `untouch_num` as aforementioned, as well as the following 5 properties.

Definition 13 `mem_num`. *The seeds in the* `mem_num` *property queue are sorted from high to low according to the total number of the memory access operations reported by LLVM instruction, e.g.,* `mayReadFromMemory()` *and* `mayWriteToMemory()`*, in their triggered basic blocks.*

Definition 14 `func_num`. *The seeds in the* `func_num` *property queue are sorted from high to low according to the number of the high-risk functions, which are most relevant to vulnerabilities summarized from the CVE dataset [3, 52], in their triggered basic blocks.*

Definition 15 `global_num`. *The seeds in the* `global_num` *property queue are sorted from high to low according to the total number of the global variables in their triggered basic blocks.*

Definition 16 `global_assign_num`. *Similarly, there are more global variables and assignment operators in the basic blocks of the seeds with the* `global_assign_num` *property.*

Definition 17 `crash_num`. *Recent researches point out that, if a program crashes in a basic block, then the program has a higher probability of triggering a crash in its neighboring basic blocks [31]. Thus, if there occurs a crash in a basic block, we mark this basic block and its neighboring blocks as high-risk blocks. Then, the seeds with the* `crash_num` *property are sorted from high to low according to the number of the marked basic blocks in their execution paths.*

3) The properties of `trace_bits` triggered by seeds include `bit_num` and `loop_num` as defined in Section 2.4.

In implementation, we write an LLVM pass to record the corresponding values for the properties of basic blocks during compilation. Then, SLIME can record all the property values for a seed by reading properties related to the seed, statically reading the above values for its triggered basic blocks, and analyzing its `trace_bits`.

3.3 The Queue Structure of SLIME

In order to 1) store the values of the above 17 properties for each seed, and 2) implement a suitable data structure for the `Seed Repla-`

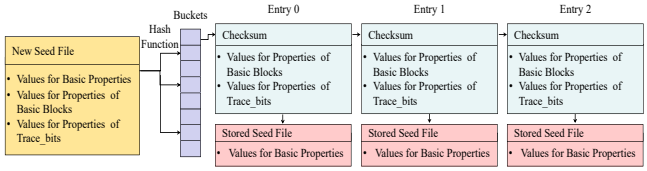

Figure 3: The new data structure of the original queue implemented in SLIME.

cement in order to replace seeds with higher quality ones, we design a new data structure for the original queue in SLIME.

Instead of storing seeds in a linked list like existing fuzzers [2, 8, 40, 56], we use a hash table to construct the original queue as shown in Fig. 3, whose workflow to add a newly discovered seed is as follows. First, SLIME calculates `checksum` by hashing the seed's `trace_bits`. Second, SLIME utilizes a hash function to hash `checksum` in order to locate the index on buckets, each of which can be empty or may contain more than one entry in order to handle a hash collision. Then, to add a new seed into the original queue, SLIME inserts a new `checksum` node, which stores the values for the properties of basic blocks and `trace_bits` triggered by this seed as shown in Fig. 3, at the end of the located bucket. Finally, SLIME creates a `seed file` node under the `checksum` node to store the seed's normal data (e.g., the file name) and the values for its basic properties.

To locate an entry which triggers the same `trace_bits` for a given test case, the workflow is close to the above. After finding the target bucket with the hash function, SLIME traverses all the entries in the bucket in order to find the `checksum` node containing the same `checksum`. Then, the found node and its `seed file` node store the data of the seed triggering the same `trace_bits`.

Based on the new data structure of the original queue, SLIME achieves to fast create and locate nodes for seeds. Moreover, the new data structure provides the infrastructure for the `Seed Replacement`. By locating the entry with the same `checksum` in the bucket and only modifying the data stored in the `seed file` node, SLIME achieves to 1) fast find the node which stores the seed triggering the same execution path, and 2) save the computational overhead of the replacement process since two seeds have the same triggered basic blocks and `trace_bits`.

As for the data structure of property queues, SLIME constructs a linked list for each property queue in the `Update` Stage. In the `Property Queue Update`, the linked list stores the entries for each property following the procedures as described in Section 3.1. To mutate the seeds in a selected property queue in the `Exploitation` Stage, SLIME traverses the corresponding linked list and reads the `seed file` nodes' data to perform the mutation.

3.4 `Seed Replacement` in SLIME

Most state-of-the-art coverage-based fuzzers store the first test case triggering an unique execution path as the seed, and then generate test cases based on this seed. However, the stored seed may not be the optimal one to find new unique paths and crashes. If the stored seed is not suitable, existing fuzzers need to spend more computational overhead generating test cases to trigger a new path.

To solve the above problem and improve the fuzzing efficiency with a better seed, we propose the `Seed Replacement` for SLIME, which solves the following challenges: 1) how to measure the quality for a seed, and 2) when to replace an old seed with a new and higher quality test case.

To overcome the first challenge, in the `Seed Replacement`, SLIME measures a seed's quality based on its performance on most efficient properties, whose workflow is as follows.

First, SLIME identifies which properties are most efficient by finding the most frequent properties on the high-efficiency seeds, which implies the reasons for finding more unique paths and crashes. To be specific, SLIME 1) records the properties of the high-efficiency seeds which have discovered the most unique paths and crashes after mutation; 2) counts the occurrences for each property; and 3) selects the top frequent properties as the efficient properties. In implementation, we refer to the design of the number of key properties of existing energy allocation algorithms as shown in Section 4.3, and empirically define the top 3 frequent properties as the efficient properties. For instance, the efficient properties can be `slow`, `bit_num` and `cmp_const_num` if they appear most frequently on the high-efficiency seeds.

Second, SLIME calculates a temporary score on each efficient property for a seed, which is normalized to the range of $(0, 1]$. To do this, SLIME records the property value of the top ranked seed in the original queue, according to the definition of each property. Then, SLIME measures the performance of a seed on a property by comparing its value with the recorded top value. Specifically, the temporary score is defined as the recorded top value divided by a seed's property value for the properties, whose queues are in the ascending order, e.g., the `fast` property and the `short` property. Oppositely, the temporary score is defined as a seed's property value divided by the top value for the other properties, whose queues are in the descending order, e.g., the `slow` property. As an example, if the `slow` property is found to be one of the efficient properties, SLIME considers that a seed with a longer execution time can find more unique paths and crashes. Then, leveraging the value of the longest execution time as the baseline, SLIME measures the performance of each seed on the `slow` property. The longer a seed's execution time is, the higher its temporary score is.

Third, SLIME sums the temporary scores on the top 3 efficient properties as the estimated quality of a seed. For example, if the efficient properties are `slow`, `bit_num` and `cmp_const_num`, SLIME calculates a temporary score on each of them and uses the sum of the 3 temporary scores as the estimated quality of a seed.

With the above three steps, SLIME achieves to measure seed quality by quantifying the performance of a seed on the top 3 efficient properties. In implementation, for every 3 cycles of stage iterations as shown in Fig. 2, SLIME reidentifies the top 3 efficient properties and remeasures the quality of each seed in the original queue, in order to balance the trade-off between computational overhead and functional requirement.

As for the second challenge, SLIME replaces the seed, which has been fuzzed and has never discovered any new path or crash, with a new test case in the `Exploration` as shown in Section 3.1, whose workflow is as follows. After the first time to enter the `Exploration` Stage, SLIME has identified the top 3 efficient properties, based on which SLIME can quantify the estimated quality of a test case. Thus,

for a test case which triggers a known execution path, SLIME fast locates the seed triggering the same path in the original queue with our new data structure. If the located seed 1) has been added into the original queue for more than 3 cycles of the stage iterations, 2) has been mutated, 3) has never discovered any new unique path or crash, and 4) has a lower estimated quality, SLIME will replace the seed with the test case by modifying the data in the `seed file` node in our new data structure.

3.5 Property-adaptive Energy Allocation

In this subsection, we mainly introduce the property-adaptive energy allocation algorithm to statistically select property queues.

To achieve the energy allocation on the seeds with different properties, in the `Exploitation` Stage, SLIME performs the queue selection process a fixed number of times, and mutates the seeds in each selected property queue. Since SLIME is supposed to select the property queues containing the efficient seeds more times and improve the fuzzing performance, the queue selection problem can be regarded as a multi-armed bandits problem. In other words, the goal of the property-adaptive energy allocation algorithm is to estimate the potential of discovering interesting test cases for each property queue (i.e., the reward), and then statistically select a property queue based on the reward. Meanwhile, to prevent from 1) slowing down the execution speed of a fuzzer and 2) significantly reducing its fuzzing performance, the energy allocation algorithm is supposed to have low computational overhead.

To solve the aforementioned problem, we design the property-adaptive energy allocation algorithm based on the Upper Confidence Bound Variance-aware (UCB-V) algorithm [6], which is one of the most popular algorithms to solve a multi-armed bandits problem. Specifically, the UCB-V algorithm used in SLIME estimates the confidence interval for the number of newly discovered interesting test cases if selecting a property queue in the `Exploitation` Stage. Then, the UCB-V algorithm regards the upper confidence bound of the estimated interval as the reward, which is the basis for property queue selection. Moreover, the more times a property queue is selected, the narrower and more accurate its upper confidence bound about interesting test case discovery is estimated by the UCB-V algorithm. Therefore, the UCB-V algorithm adaptively optimizes the estimated rewards for property queues during the fuzzing process, improving the fuzzing performance. The relevant symbols of the UCB-V algorithm are defined as follows.

Definition 18 G. *$G[i]$ is the number of newly discovered interesting test cases, which trigger new unique paths and crashes, when selecting the ith property queue once and mutating its seeds in the* `Exploitation` *Stage.*

Definition 19 R. *$R[i]$ is the sum of $G[i]$ for the ith property queue.*

Definition 20 R_Sq. *$R\_Sq[i]$ is the sum of squares of $G[i]$ for the ith property queue.*

Definition 21 N. *$N[i]$ is the number of times the ith property queue was selected in the* `Exploitation` *Stage, and N_total is the number of selections for all the queues.*

Definition 22 Variance. *$Variance[i]$ is the variance estimation for the ith property queue, which improves the algorithm's convergence rate and helps find the efficient property queues faster.*

Definition 23 UCB_V. *$UCB\_V[i]$ is the estimated upper confidence bound of the ith property queue.*

$$R[i] += G[i]. \tag{1}$$

$$R\_Sq[i] += G[i] \times G[i]. \tag{2}$$

$$Variance[i] = \frac{R\_Sq[i]}{N[i]} - \frac{R[i] \times R[i]}{N[i] \times N[i]}. \tag{3}$$

$$UCB\_V[i] = \frac{R[i]}{N[i]} + \sqrt{\frac{2 \times Variance[i] \times \log(N\_total)}{N[i]}} + \frac{3 \times \log(N\_total)}{N[i]}. \tag{4}$$

The workflow of the UCB-V algorithm used in SLIME is as follows. In the `Exploitation` Stage, each time to select a property queue to mutate, SLIME will perform Formula 4 to calculate the upper confidence bound for each property queue. If there is a property queue that has never been selected (i.e., $N == 0$), SLIME will select this property queue to perform the following fuzzing process; Otherwise, SLIME selects the property queue with the largest UCB_V (which is one of the most common usages of UCB algorithms in practice). After selecting a property queue, SLIME performs the mutation process with the seeds in this property queue, and counts the number of newly discovered interesting test cases (i.e., G). When all the seeds in the property queue have been mutated, a queue selection process is complete. Then, SLIME updates R and R_Sq for the selected property queue following Formula 1 and 2. After that, SLIME performs the next queue selection process; or SLIME finishes the `Exploitation` Stage and enters the `Exploration` Stage if SLIME performs the queue selection process enough times.

4 EVALUATION

4.1 Experiment Setup

In the evaluation, we aim to answer the following questions:
RQ1 - How effective is the fuzzing performance of SLIME?
RQ2 - What is the contribution of each main part of SLIME to its fuzzing performance?

Compared fuzzers. We compare SLIME with the open source fuzzers AFL [2], MOpt [32], AFL++ [17], AFL++HIER [50], EcoFuzz [56], and TortoiseFuzz [52] for the following reasons. First, AFL is a widely used baseline in the state-of-the-art papers [12, 13, 53, 55]. Second, MOpt is a newly developed fuzzer that focuses on allocating different energy to different mutation operators. AFL++ is a superior fuzzer enhanced by various designs of state-of-the-art fuzzers. Third, AFL++HIER, EcoFuzz and TortoiseFuzz are 3 state-of-the-art energy allocation solutions, which can show the performance difference between different energy allocation strategies.

Real-world target programs and initial seed sets. We follow the guideline of *UniFuzz* [29] to evaluate the above fuzzers with 9 widely-used programs as shown in Table 1, which are randomly selected from the targets of UniFuzz and state-of-the-art fuzzing papers [18, 19, 32, 55, 56]. The initial seed set for each target program is provided by the open-source data sets of UniFuzz, which 1) collects seeds with the corresponding input format from the Internet; 2) excludes the seeds that do not satisfy fuzzers' requirements; and 3) randomly selects 100 seeds as the initial seed set.

Experiment settings. We conduct the evaluation of each fuzzer on a target program under the same experiment settings: a docker container configured with 1 CPU core of 2.40GHz E5-2680 V4 and the 64-bit Ubuntu 16.04 LTS. To reduce the impact of randomness

Table 1: The open-source target programs of the benchmark.

Program	Version	Input format	Test instruction
cflow	1.6	txt	@@
ffmpeg	4.0.1	mp4	-y -i @@ -c:v mpeg4 -c:a copy -f mp4 /dev/null
gdk	gdk-pixbuf 2.31.1	jpg	@@ /dev/null
imginfo	jasper 2.0.12	jpg	-f @@
jhead	3.00	jpg	@@
mp3gain	1.5.2-r2	mp3	@@
objdump	binutils 2.28	binary	-S @@
pdfimages	xpdf 4.00	pdf	@@ /dev/null
tiffsplit	libtiff 3.9.7	tiff	@@

Table 2: The unique vulnerability discovery of SLIME with the different property queue lengths.

Programs	Length ratio of property queues to the original queue			
	1/10	4/10	8/10	10/10
ffmpeg	1	**2**	1	1
jhead	4	**5**	4	4
objdump	15	**16**	15	15
tiffsplit	12	**13**	11	11
total	32	**36**	31	31

and improve the reproducibility of the results, 20 trials are repeated in each evaluation, each of which lasts 120 hours. In total, we spend several months running the evaluations on 8 servers, each of which has two E5-2680 V4 CPUs and 256 GB memory.

Evaluation metrics. To measure the performance of each fuzzer with unified metrics. First, we evaluate the vulnerability discovery performance of each fuzzer, according to the number of the unique vulnerabilities reported by AddressSanitizer (ASan) [1] on target programs. The reasons for using the number of unique vulnerabilities rather than unique crashes are as follows. 1) Not all the crashes discovered by a fuzzer trigger unique vulnerabilities. Thus, the number of unique crashes found by each fuzzer cannot accurately reflect its vulnerability discovery performance; 2) On the contrary, the number of the unique vulnerabilities directly shows the vulnerability discovery performance of each fuzzer. Since ASan can report different types of vulnerabilities on a program and is often used in the state-of-the-art papers [18, 29, 35, 52], we leverage the number of unique vulnerabilities reported by ASan to measure the fuzzer's vulnerability discovery performance. In this paper, we extract the top three functions in the stack trace reported by ASan to deduplicate the vulnerabilities following the guideline of UniFuzz.

To compare the coverage performance of different fuzzers, the second metric is the average edge coverage reported by AFL-cmin [2] in 20 trials, which is a widely used metric in [19, 42, 43, 56].

Length selection for property queues of SLIME. To choose a suitable property queue length for SLIME, we analyze the influence of the length of the property queues on vulnerability discovery. To do this, we conduct experiments to evaluate SLIME with different property queue lengths on `ffmpeg`, `jhead`, `objdump`, and `tiffsplit`. We select 1/10, 4/10, 8/10, and 10/10 of the length of the original queue as the length of the property queues in SLIME. In the `Exploitation` Stage, to eliminate the influence brought by the UCB-V algorithm, SLIME with each length ratio traverses and mutates the seeds in all the property queues, rather than statistically selects the property queue according to the UCB-V algorithm. Each trial lasts 96 hours and is repeated 4 times to reduce randomness. The number of the total unique vulnerabilities after deduplication

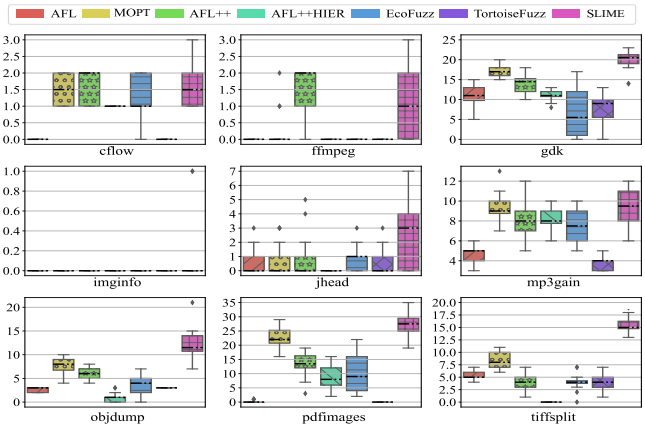

Figure 4: The boxplot generated by the number of unique vulnerabilities in 20 trials, where '— -' represents the median. Y-axis: the number of unique vulnerabilities discovered in each trial.

Table 3: The number of total unique vulnerabilities after deduplication in 20 trials found by each fuzzer.

	AFL	MOPT	AFL++	AFL++HIER	EcoFuzz	TortoiseFuzz	SLIME
cflow	0	2	2	1	2	1	**4**
ffmpeg	0	2	2	0	0	0	**3**
gdk	23	31	26	23	26	20	**32**
imginfo	0	0	0	0	0	0	**1**
jhead	5	6	5	0	5	5	**10**
mp3gain	8	17	16	18	16	5	**23**
objdump	5	30	28	5	14	5	**39**
pdfimages	1	75	49	44	48	0	**87**
tiffsplit	9	24	15	0	12	12	**32**
total	51	187	143	91	123	48	**231**

in 4 trials are shown in Table 2, and we have the following observation. **When using 4/10 of the length of the original queue as the length of the property queues, SLIME performs the best on the vulnerability discovery.** For instance, SLIME with the 4/10 length ratio finds 1 more unique vulnerability than others on `ffmpeg`, `jhead` and `objdump`. In total, SLIME with the 4/10 length ratio finds 4, 5 and 5 more unique vulnerabilities than SLIME with the 1/10, 8/10 and 10/10 length ratio on all the programs, respectively. We analyze the reasons for the results in Table 2 as follows.

The ratio of the length of property queues to the length of the original queue represents the proportion of the seeds which will be mutated in the `Exploitation` Stage. Thus, SLIME with the 10/10 length ratio actually executes the same fuzzing process in the `Exploitation` Stage as in the `Exploration` Stage, since SLIME stores all the seeds of the original queue into each property queue and mutates all the seeds in both stages. Furthermore, SLIME constructs the property queues that have the opposite properties. There can be the same seeds stored in both the opposite property queues if the length ratio is larger than 5/10. For instance, when the length ratio is 8/10, there are several seeds with the medium execution speed stored in both the `fast` and `slow` property queues. These seeds in the opposite property queues can obtain more energy allocation. On the contrary, the small length ratio of SLIME may lead to unbalanced energy allocation, in other words, SLIME spends

Table 4: The number and types of new unique vulnerabilities which are only found by SLIME and are missed by other fuzzers.

	SEGV on unknown address, READ memory access	heap-buffer-overflow	stack-overflow	memory leaks	allocation-size-too-big	total
ffmpeg	1	0	0	0	0	1
gdk	0	3	0	0	0	3
jhead	0	4	0	0	0	4
objdump	7	1	0	0	0	8
pdfimages	1	10	4	0	0	15
tiffsplit	0	3	0	5	2	10
total	9	21	4	5	2	41

Table 5: The properties and values of each original seed of SLIME that triggers a new unique vulnerability on `objdump` after mutation. A value in bold font means that the original seed has the corresponding property.

seed_id	long (file size)	global_num	global_assign_num	func_num
No. 1	**32,391.00**	**660.00**	**108.00**	47.00
No. 2	10,432.00	583.00	**101.00**	70.00
No. 3	13,488.00	564.00	91.00	**78.00**
No. 4	**32,452.00**	**720.00**	**121.00**	47.00
mean among all the seeds	54,116.90	508.08	84.18	64.78
median among all the seeds	13,952.00	577.50	94.00	57.50

the most computational energy mutating a small proportion of the seeds in the `Exploitation` Stage. The small proportion of the seeds can be mutated too many times to find any new unique path and crash in the `Exploitation` Stage after a long fuzzing duration. Therefore, either the small or the large length ratio is not suitable as the length of the property queues in SLIME.

Empirically, we use the 4/10 length ratio in the implementation of SLIME, as it achieves the best vulnerability discovery performance.

4.2 Vulnerability & Coverage Discovery (RQ1)

In this subsection, we analyze the vulnerability discovery and edge coverage performance of each fuzzer on 9 target programs. The boxplot generated by the number of unique vulnerabilities found by each fuzzer in 20 trials is shown in Fig. 4, and the number of total unique vulnerabilities after deduplication in all 20 trials on each target is shown in Table 3.

• As shown in Fig. 4, SLIME achieves the best vulnerability discovery on most programs according to the upper quartile and median of the boxplots. For instance, SLIME is more likely to find more vulnerabilities than others in each trial on `objdump`. On `jhead`, the median of SLIME is significantly larger than other fuzzers' median. The results demonstrate the significant performance of SLIME on vulnerability discovery in one trial of a program.

• Table 3 presents that SLIME finds the most total unique vulnerabilities in 20 trials on all the programs. For instance, SLIME finds 8, 17 and 20 more vulnerabilities than MOpt, AFL++ and EcoFuzz on `tiffsplit`, respectively. In total, SLIME discovers 44 more unique vulnerabilities than the second best fuzzer on 9 programs. SLIME discovers 3.53× more unique vulnerabilities than the baseline AFL.

• We further analyze the unique vulnerabilities found by SLIME, and find out the ones that 1) cannot be found by other fuzzers and 2) are not published on the CVE website [3]. The results are shown in Table 4, from which we can realize that SLIME finds significantly more new unique vulnerabilities than others. Then, we analyze the PoCs of these new vulnerabilities. To be specific, we trace the original seeds in the original queue which generate these PoCs after mutation. Then, we perform the property queue construction of SLIME to classify seeds from the original queue to property queues, and record their properties. As described above, we analyze several PoCs along with their original seeds for the new unique vulnerabilities of `objdump` found by SLIME, and find that the PoCs are generated by the seeds which will be assigned less mutation energy in traditional energy allocation algorithms. The properties and values of 4 original seeds are shown in Table 5. If a seed has a specific property, its corresponding value will be in bold font. For instance, when fuzzing `objdump`, SLIME mutates No. 4 seed having the `long`, `global_num` and `global_assign_num` properties, and then triggers a new unique vulnerability that is missed by all other fuzzers. Since this seed is classified into `long`, `global_num` and `global_assign_num` property queues, it will be assigned more energy in the `Exploitation` Stage of SLIME. On the contrary, due to the `long` property, this seed will be assigned minor mutation energy in the state-of-the-art fuzzers like AFL, MOpt, and AFL++.

Table 6: The published CVE IDs found by each fuzzer.

	CVE ID	AFL	MOpt	AFL++	AFL++HIER	EcoFuzz	TortoiseFuzz	SLIME
cflow	CVE-2020-23856							●
	CVE-2019-16166							●
	CVE-2019-16165		●	●	●	●	●	●
imginfo	CVE-2017-6851							●
jhead	CVE-2020-6624	●	●	●		●	●	●
	CVE-2019-1010302	●	●	●				●
	CVE-2019-19035					●	●	
mp3gain	CVE-2017-14410			●	●			●
	CVE-2017-14409		●	●	●	●		●
	CVE-2017-14407	●	●	●	●	●	●	●
objdump	CVE-2021-3487		●	●		●		●
	CVE-2019-17450							●
	CVE-2019-9072		●	●				●
	CVE-2018-1000876		●	●				●
	CVE-2018-7568		●	●		●		●
	CVE-2017-16831		●	●		●		●
	CVE-2017-16826		●	●		●		●
	CVE-2017-15024		●	●				●
	CVE-2017-14938		●	●				●
	CVE-2017-8396	●	●	●	●	●	●	●
pdfimages	CVE-2020-24999		●					●
	CVE-2019-13291		●			●		●
	CVE-2019-13281		●	●		●		●
	CVE-2019-10022							●
	CVE-2018-18458		●					
	CVE-2018-18455		●	●	●	●		
	CVE-2018-16368	●	●	●	●	●		●
	CVE-2018-7453	●	●	●	●	●		●
	total	6	21	19	8	15	5	**25**

To further verify the validity of the discovered vulnerabilities for each fuzzer, we record the published Common Vulnerabilities and Exposures (CVE) IDs found by each fuzzer on target programs. To do this, we 1) collect the stack traces of the CVE IDs and their Proof-of-Concept (PoC) exploits from [3] for each program; 2) leverage the PoC exploits to reproduce the stack traces of the CVE IDs on our

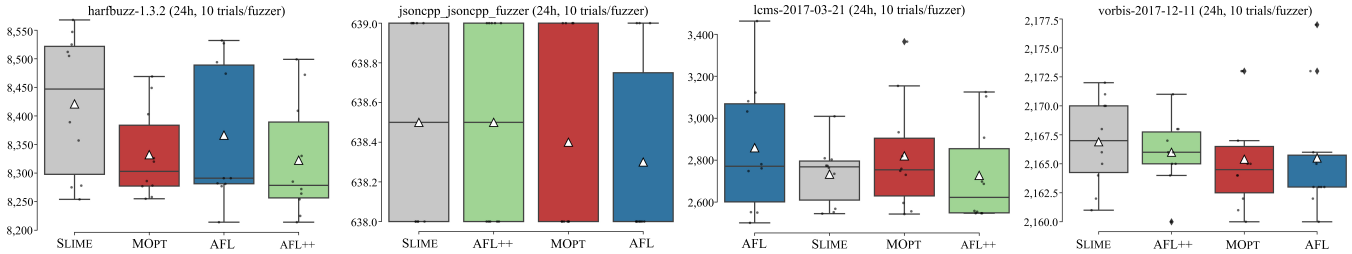

Figure 5: The boxplot of region coverage found in 10 trials on FuzzBench, where '△' and '—' represent the mean and median, respectively. The fuzzer with the highest median coverage is on the left. Y-axis: the region coverage found in each trial.

Table 7: The number of average edge coverage in 20 trials found by each fuzzer.

	AFL	MOpt	AFL++	AFL++HIER	EcoFuzz	TortoiseFuzz	SLIME
cflow	1,760.50	1,784.05	1,785.65	1,804.55	1,784.30	1,762.25	**1,819.75**
ffmpeg	18,046.25	31,343.45	**43,838.75**	33,317.40	21,293.55	17,002.20	32,825.75
gdk	1,458.40	1,985.00	1,577.95	1,505.40	1,236.30	1,335.75	**2,038.85**
imginfo	2,192.15	3,258.90	2,743.35	1,179.55	2,815.85	1,716.05	**3,500.25**
jhead	283.00	283.00	283.00	283.00	283.00	283.00	283.00
mp3gain	1,203.40	1,297.45	1,282.85	1,278.05	1,281.25	1,191.80	**1,300.40**
objdump	5,568.45	7,687.20	**8,071.95**	5,785.40	7,009.00	5,576.85	7,954.95
pdfimages	10,275.55	11,497.80	11,245.80	10,909.60	10,626.35	10,239.35	**11,932.70**
tiffsplit	3,036.35	3,291.75	3,290.40	2,289.60	2,964.20	2,988.90	**3,308.50**

instrumented target program; 3) compare the top three stack traces between the CVE IDs and vulnerabilities found by each fuzzer; and 4) record the triggered CVE IDs found by each fuzzer. In our evaluation, no fuzzer triggers any published CVE ID on `ffmpeg`, `gdk`, and `tiffsplit`. For ease of demonstration, we show the triggered CVE IDs for the remaining target programs in Table 6, from which we can have the following conclusion.

• Table 6 shows that SLIME achieves the best performance on CVE discovery, which contains 5 unique CVE IDs missed by other fuzzers. Specifically, SLIME finds 25 CVE IDs on all the 9 programs, and finds 4 more CVE IDs than the second best fuzzer. SLIME also finds 6 and 10 more CVE IDs than AFL++ and EcoFuzz on all the target programs. These discovered CVE IDs seriously threaten the services and users. For instance, CVE-2017-6851 allows remote attackers to cause a denial of service via a crafted image and affects the normal function of `imginfo`. Thus, the significant performance of discovering published CVE IDs demonstrates the effectiveness and efficiency of SLIME on serious vulnerability discovery.

As for the coverage, the number of average edge coverage in 20 trials achieved by each fuzzer is shown in Table 7.

As for the coverage, the number of average edge coverage in 20 trials achieved by each fuzzer is shown in Table 7, from which we can have the following conclusion. **SLIME performs the best on the edge coverage on 6 of all the 9 programs.** While all the tested fuzzers have the same edge coverage on `jhead`, SLIME achieves the top 3 edge coverage on the other 2 programs. The results in Table 7 also show the significant edge coverage improvement of SLIME compared to AFL, EcoFuzz and TortoiseFuzz on most programs.

To further evaluate fuzzers' coverage performance with the acknowledged benchmark, we utilize *FuzzBench*, one of the most famous standardized benchmarks, to evaluate AFL, MOpt, AFL++, and SLIME on 4 programs, including `harfbuzz`, `jsoncpp`, `lcms`, and `vorbis`. Each evaluation lasts 24 hours and is repeated 10 times to reduce the randomness. The results are shown in Fig. 5, from which we have the following conclusion. **SLIME performs the best on region coverage when using the standardized benchmark FuzzBench to measure the performance**. For instance, the median of SLIME is the highest on `harfbuzz`, `jsoncpp`, and `vorbis`. SLIME performs better than MOpt and AFL++ on `lcms`. The mean of SLIME is higher than other fuzzers on `harfbuzz` and `vorbis`. The results demonstrate the significant coverage performance of SLIME on a standardized benchmark.

RQ1: SLIME is effective and efficient in coverage improvement and vulnerability discovery. SLIME significantly outperforms the state-of-the-art fuzzers in terms of vulnerabilities detection. As for coverage, SLIME achieves more average edge coverage on most programs, and performs the best on FuzzBench.

4.3 Contributions of SLIME's Main Parts (RQ2)

To evaluate the contribution of the property-adaptive energy allocation algorithm as shown in Section 3.5, we construct the following one version of SLIME with a different queue selection algorithm, i.e., SLIME_rand, which randomly selects each property queue with the same probability in the `Exploitation` Stage. We evaluate the vulnerability discovery performance of SLIME_rand on `gdk`, `objdump` and `tiffsplit`. Each trial lasts 120 hours and is repeated 20 times, whose experiment settings are the same as in Section 4.1. Then, we compare the vulnerability discovery of SLIME_rand with the results of MOpt, AFL++ and SLIME in Section 4.2. Based on the above experiment setup, we can measure the contribution of the property-adaptive energy allocation algorithm to the vulnerability discovery from the results of SLIME_rand and SLIME, and measure the contribution of the property queue construction from the results of MOpt and SLIME_rand. Table 8 shows the number of total unique vulnerabilities after deduplication in 20 trials and the average number of unique vulnerabilities in each trial found by each fuzzer, from which we have the following conclusions.

• The property queue construction cannot significantly improve the vulnerability discovery performance without the property-adaptive energy allocation algorithm. For instance, MOpt finds 1 more unique vulnerability than SLIME_rand on `gdk` and `tiffsplit` after deduplication in 20 trials, respectively. However, after implementing the property-adaptive energy allocation algorithm, SLIME find more unique vulnerabilities than others on most programs.

Table 8: The number of unique vulnerabilities found by MOpt, AFL++, SLIME_rand, and SLIME.

		MOpt	AFL++	SLIME_rand	SLIME
# of total unique vulnerabilities in 20 trials	gdk	31	26	30	**32**
	objdump	30	28	30	**39**
	tiffsplit	24	15	23	**32**
	Total	85	69	83	**103**
Average # of unique vulnerabilities in each trial	gdk	17.15	14.00	18.45	**20.03**
	objdump	7.65	5.90	10.10	**12.10**
	tiffsplit	8.40	4.00	7.70	**15.75**
	Total	33.20	23.90	36.25	**47.88**

Table 9: The average edge coverage increment of SLIME_no and SLIME when using an extensive data set, which has found the most edge coverage, as the initial seed set.

Programs	Original edge results	SLIME_no	SLIME	Increase [a]
ffmpeg	32,825.75	+3,813.35	+4,531.15	+18.82%
imginfo	3,500.25	+207.15	+328.75	+58.70%
pdfimages	11,932.70	+36.60	+50.30	+37.43%
tiffsplit	3,308.50	+27.30	+29.60	+8.42%

[a]Increase is calculated by the results of SLIME divided by SLIME_no's.

• Our customized UCB-V algorithm significantly improves the vulnerability discovery performance of SLIME. SLIME discovers significantly more unique vulnerabilities than other fuzzers on all the 3 programs. For instance, SLIME finds 9 and 8 more total unique vulnerabilities than SLIME_rand and MOpt on `tiffsplit`, respectively. The average number of unique vulnerabilities found by SLIME is also significantly larger than other fuzzers in each trial of a program.

To evaluate the contribution of the `Seed Replacement` in SLIME to the fuzzing performance, we construct SLIME_no without the `Seed Replacement`, and evaluate SLIME_no and SLIME with the following experiment setup. We evaluate the edge coverage performance of SLIME_no and SLIME on `ffmpeg`, `imginfo`, `pdfimages`, and `tiffsplit`. In order to increase difficulty in finding new edge coverage and show the usage of the `Seed Replacement`, we use the coverage results in Section 4.2, which have found the most unique edges, as the initial seed set of SLIME_no and SLIME for each target program. Thus, both SLIME_no and SLIME are started with a large seed set and are conducted 20 trials for each program, respectively. Each trial lasts 48 hours. Then, we compare the average edge coverage increment, i.e., the number of newly discovered edge coverage on average compared to the initial seed set, of SLIME_no and SLIME reported by AFL-cmin. The results are shown in Table 9, from which we have the following conclusion. **The `Seed Replacement` can improve the edge coverage performance.** For instance, the average edge coverage increment of SLIME is 0.19× and 0.37× larger than SLIME_no on `ffmpeg` and `pdfimages`, respectively. The results demonstrate the contribution of the `Seed Replacement` to the fuzzing performance.

RQ2: The property queue construction cannot improve the vulnerability discovery performance alone, and our UCB-V algorithm significantly improves SLIME's vulnerability discovery performance. The `Seed Replacement` also contributes to the fuzzing performance.

5 DISCUSSION AND LIMITATION

Energy allocation between different stages. SLIME mainly focuses on adaptively assigning mutation energy to the seed files with different properties in the `Exploitation` Stage. In addition, it would be possible to further improve the energy allocation logic between the `Exploration` Stage and the `Exploitation` Stage. Therefore, how to make better use of different energy allocation strategies in the two stages and seek an energy allocation balance is an interesting future work.

Further utilization of the estimated quality. SLIME quantifies the estimated quality for each seed, which is calculated by its property values on the top 3 efficient properties. The estimated quality is used to decide whether to keep a seed in the original queue as shown in Section 3.4. In future work, the estimated quality could be used in other situations, e.g., the mutation order of seeds and the execution times of one seed in each mutation. How to optimize the usage of the estimated quality could be a promising topic.

6 RELATED WORK

6.1 Mutation-based Fuzzing

Various mutation-based fuzzing solutions have been proposed from different aspects to improve fuzzing performance.

Several studies utilize symbolic execution [15, 25, 26, 43, 59] or taint tracking [18, 22, 36, 40] to improve fuzzing performance. Yun et al. proposed a concolic execution engine named *QSYM*, which runs hybrid fuzzing with AFL to improve the constraint solving performance [57]. Profuzzer includes a lightweight mechanism to discover the relations between input bytes and program behaviors [55]. Redqueen solves magic bytes and checksum automatically with input-to-state correspondence [5]. GreyOne adopts a sound fuzzing-driven taint inference, which takes both taint attributes and constraint conformance into consideration [18]. Recent works employ machine learning algorithms to improve fuzzing effectiveness [14, 21, 39, 41, 61]. Specifically, NEUZZ leverages surrogate neural network models to locate byte positions that have great impacts on program behaviors, and demonstrates the potential of gradient-guided seed generation methods together with the neural smoothing technique [42].

Plenty of energy allocation solutions have been proposed from different aspects. AFL assigns more energy to the seeds, which cover more edge coverage, execute faster, and are discovered later [2]. Several energy allocation fuzzers allocate more energy to explore low-frequency paths, untouched branches and unexplored descendants [8, 19, 56]. Considering more precise coverage, AFL-Sensitive presents context-sensitive and n-gram branch coverage to distinguish potential seeds [49], and Ankou employs an informative distance-based fitness function [35]. Cerebro allocates energy to a seed according to the complexity of its execution code and uncovered close code [30]. AFL++HIER leverages a multi-armed bandit model to allocate energy on different clusters of seeds with multi-level coverage metrics [50].

Unlike other state-of-the-art energy allocation strategies, SLIME adaptively allocates different energy to the seed files with different properties according to their efficiency on each target program, rather than allocating energy following the static logic.

6.2 Generation-based Fuzzing

Generation-based fuzzing mainly focuses on generating test cases with specific input formats [16, 20, 23, 34, 48]. Traditionally, CSmith [54], Radamsa [45], LangFuzz [24], and IFuzzer [44] generate test cases by employing the context-free grammar as the specification. Meanwhile, CLP utilizes manually-specified generation rules to express semantic rules [16]. Skyfire learns a probabilistic context-sensitive grammar to generate test cases for XSLT, XML, JavaScript, and rendering engines [47]. Nautilus combines grammar-based input generation with feedback-directed fuzzing on multiple targets, including ChakraCore, PHP, mruby, and Lua [4]. Lee et al. presented a neural network language model guided fuzzer named *Montage* to find JavaScript engine vulnerabilities [27].

6.3 Other Kinds of Fuzzing Tools

Directed fuzzing takes a set of predetermined target positions, and guides the fuzzing process to trigger the target positions [7, 61]. Specifically, Hawkeye leverages precise control flow information to achieve better directedness [9]. Some works prefer bug coverage. For instance, ParmeSan actively guides the fuzzing process towards triggering the sanitizer checks [37], and SAVIOR solves constraints for UBSan checks to direct the fuzzing process towards actual bugs [13]. Other than the general purpose, more kinds of fuzzers are proposed to find specific types of bugs, such as algorithmic complexity vulnerabilities [38], memory corruption [52], uncontrolled memory consumption [53], use-after-free [46], and so on. Recently, plenty of works focus on Internet of Things (IoT) security and propose their well-designed tools [10, 28, 51, 58, 60].

7 CONCLUSION

In this paper, by conducting a case study, we confirm our observation that the seed files with different properties have different efficiency to discover unique paths and crashes on a program, and the seed files with the same property also have different efficiency on different programs. Based on this observation, we present a program-sensitive energy allocation solution SLIME to adaptively assign the appropriate energy to the seed files with different properties for a given target program. We demonstrate that SLIME significantly outperforms the state-of-the-art fuzzers on vulnerability discovery and coverage performance. Furthermore, the analysis also shows that SLIME's property-adaptive energy allocation algorithm significantly improves the vulnerability discovery performance, and the `Seed Replacement` of SLIME can improve the coverage performance. With great flexibility, SLIME can serve as a key energy allocation strategy to improve the vulnerability discovery and coverage performance of most mutation-based fuzzers.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable comments to improve our paper. This work was partly supported by NSFC under No. U1936215 and 62102360, the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, and the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform).

REFERENCES

[1] 2021. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.
[2] 2021. American Fuzzy Lop. https://github.com/google/AFL.
[3] 2021. NVD, CVE: Common Vulnerabilities and Exposures. https://cve.mitre.org.
[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*. https://doi.org/10.14722/ndss.2019.23412.
[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *NDSS*, Vol. 19. 1–15. https://doi.org/10.14722/ndss.2019.23371.
[6] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvári. 2009. Exploration–exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science* 410, 19 (2009), 1876–1902. https://doi.org/10.1016/j.tcs.2009.01.016.
[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344. https://doi.org/10.1145/3133956.3134020.
[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506. https://doi.org/10.1145/2976749.2978428.
[9] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108. https://doi.org/10.1145/3243734.3243849.
[10] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*. https://doi.org/10.14722/ndss.2018.23159.
[11] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725. https://doi.org/10.1109/sp.2018.00046.
[12] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 499–513. https://doi.org/10.1145/3319535.3363225.
[13] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596. https://doi.org/10.1109/sp40000.2020.00002.
[14] Yuqi Chen, Christopher M Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-guided network fuzzing for testing cyber-physical system defences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 962–973. https://doi.org/10.1109/ase.2019.00093.
[15] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-level constraint solving for hybrid fuzzing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 515–530. https://doi.org/10.1145/3319535.3354249.
[16] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 725–730. https://doi.org/10.1145/2642937.2642963.
[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. https://www.usenix.org/conference/woot20/presentation/fioraldi.
[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. *GREYONE*: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium*. 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan.
[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696. https://doi.org/10.1109/sp.2018.00040.
[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 206–215. https://doi.org/10.1145/1379022.1375607.
[21] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59. https://doi.org/10.1109/ase.2017.8115618.
[22] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *22nd USENIX Security Symposium*. 49–64. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/haller.
[23] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines.

In *NDSS*. https://doi.org/10.14722/ndss.2019.23263.

[24] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium*. 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

[25] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627. https://doi.org/10.1109/sp40000.2020.00063.

[26] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*. https://doi.org/10.14722/ndss.2020.24018.

[27] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium*. 2613–2630. https://dl.acm.org/doi/10.5555/3489212.3489359.

[28] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: Non-intrusive Feedback-driven Fuzzing for Microcontroller Firmware. *arXiv preprint arXiv:2202.03013* (2022). https://arxiv.org/abs/2202.03013.

[29] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *30th USENIX Security Symposium*. https://www.usenix.org/system/files/sec21summer$_li-yuwei.pdf$.

[30] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544. https://doi.org/10.1145/3338906.3338975.

[31] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1547–1559. https://doi.org/10.1145/3377811.3380923.

[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium*. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

[33] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Reheem Beyah. 2022. EMS: History-Driven Mutation for Coverage-based Fuzzing. In *29th Annual Network and Distributed System Security Symposium*. https://dx.doi.org/10.14722/ndss.2022.23162.

[34] Rui Ma, Shuaimin Ren, Ke Ma, Changzhen Hu, and Jingfeng Xue. 2017. Semi-valid fuzz testing case generation for stateful network protocol. *Tsinghua Science and Technology* 22, 5 (2017), 458–468. https://doi.org/10.23919/tst.2017.8030535.

[35] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1024–1036. https://doi.org/10.1145/3377811.3380421.

[36] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. 2015. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. 87–97. https://doi.org/10.1145/2699026.2699098.

[37] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium*. 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund.

[38] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168. https://doi.org/10.1145/3133956.3134073.

[39] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017). https://arxiv.org/abs/1711.04596.

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*. https://doi.org/10.14722/ndss.2017.23404.

[41] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 737–749. https://doi.org/10.1145/3368089.3409723.

[42] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817. https://doi.org/10.1109/sp.2019.00052.

[43] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, Vol. 16. 1–16. https://doi.org/10.14722/ndss.2016.23368.

[44] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29.

[45] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. 2008. Experiences with Model Inference Assisted Fuzzing. *WOOT* 2 (2008), 1–2. https://dl.acm.org/doi/10.5555/1496702.1496704.

[46] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 999–1010. https://doi.org/10.1145/3377811.3380386.

[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594. https://doi.org/10.1109/sp.2017.23.

[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735. https://doi.org/10.1109/icse.2019.00081.

[49] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 1–15. https://www.usenix.org/system/files/raid2019-wang-jinghan.pdf.

[50] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. NDSS. https://doi.org/10.14722/ndss.2021.24486.

[51] Qinying Wang, Shouling Ji, Yuan Tian, Xuhong Zhang, Binbin Zhao, Yuhong Kan, Zhaowei Lin, Changting Lin, Shuiguang Deng, Alex X. Liu, and Reheem Beyah. 2021. MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols. In *30th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying.

[52] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. NDSS. https://doi.org/10.14722/ndss.2020.24422.

[53] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777. https://doi.org/10.1145/3377811.3380396.

[54] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/2345156.1993532.

[55] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 769–786. https://doi.org/10.1109/sp.2019.00057.

[56] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity20/presentation/yue.

[57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[58] Binbin Zhao, Shouling Ji, Wei-Han Lee, Changting Lin, Haiqin Weng, Jingzheng Wu, Pan Zhou, Liming Fang, and Raheem Beyah. 2020. A Large-scale Empirical Study on theVulnerability of Deployed IoT Devices. *IEEE Transactions on Dependable and Secure Computing* (2020). https://ieeexplore.ieee.org/document/9259111.

[59] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *NDSS*. https://doi.org/10.14722/ndss.2019.23504.

[60] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium*. 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng.

[61] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium*. 2255–2269. https://dl.acm.org/doi/10.5555/3489212.3489339.