EMS: History-Driven Mutation for Coverage-based Fuzzing

Chenyang Lyu[†], Shouling Ji[†], Xuhong Zhang[†,§,✉], Hong Liang[†], Binbin Zhao[*], Kangjie Lu[¶], and Raheem Beyah[*]

[†]Zhejiang University, [§]Zhejiang University NGICS Platform, [*]Georgia Institute of Technology, [¶]University of Minnesota
E-mails: {puppet, sji, zhangxuhong, hongliang}@zju.edu.cn, binbin.zhao@gatech.edu, kjlu@umn.edu, rbeyah@ece.gatech.edu

***Abstract*—Mutation-based fuzzing is one of the most popular approaches to discover vulnerabilities in a program. To alleviate the inefficiency of mutation-based fuzzing incurred by high randomness in the mutation process, multiple solutions are developed in recent years, especially coverage-based fuzzing. They mainly employ adaptive mutation strategies or integrate constraint-solving techniques to make a good exploration of the test cases which trigger unique paths and crashes. However, they lack a fine-grained reusing of fuzzing history to construct these interesting test cases, i.e., they largely fail to properly utilize fuzzing history across different fuzzing trials. In fact, we discover that test cases in fuzzing history contain rich knowledge of the key mutation strategies that lead to the discovery of unique paths and crashes. Specifically, partial path constraint solutions implicitly carried in these mutation strategies can be reused to accelerate the discovery of new paths and crashes that share similar partial path constraints.**

Therefore, we first propose a lightweight and efficient Probabilistic Byte Orientation Model (`PBOM`) that properly captures the byte-level mutation strategies from intra- and inter-trial history and thus can effectively trigger unique paths and crashes. We then present a novel history-driven mutation framework named EMS that employs `PBOM` as one of the mutation operators to probabilistically provide desired mutation byte values according to the input ones. We evaluate EMS against state-of-the-art fuzzers including AFL, QSYM, MOPT, MOPT-dict, EcoFuzz, and AFL++ on 9 real world programs. The results show that EMS discovers up to 4.91× more unique vulnerabilities than the baseline, and finds more line coverage than other fuzzers on most programs. We report all of the discovered new vulnerabilities to vendors and will open source the prototype of EMS on GitHub.

I. Introduction

As one of the most prevalent software-testing approaches, mutation-based fuzzing generates test cases with simple random-based mutation strategies and tests a target program frequently to explore vulnerabilities. The high randomness in the mutation process results in a limited exploration of the test cases that trigger unique paths or crashes, leading to low efficiency of finding vulnerabilities. In recent years, multiple solutions have been proposed to improve mutation-based fuzzing, one of which is coverage-based fuzzing, e.g., AFL [3]. Since the precondition of triggering vulnerabilities is to execute the corresponding code paths in the target program, coverage-based fuzzing tries to explore as many unique execution paths of a program as possible. Therefore, many efforts have been taken to improve coverage-based fuzzers. Among them, one prevalent direction is employing adaptive mutation strategies to improve the seed selection and generation process [13], [44], [54], [71], e.g., AFLFast [13] and EcoFuzz [71] focus on adaptively estimating the potential of each test case to trigger unique branching behaviors and thus allocate more time to mutate the promising ones. Another popular direction is integrating constraint-solving techniques [17], [18], [63], [70], [72], [74] to solve the complex constraints, which lead to different branching behaviors and cover more difficult paths that are rarely triggered via traditional mutation.

While existing fuzzers take a good exploration of the test cases that trigger unique paths and crashes, they lack a fine-grained reusing of fuzzing history to construct these test cases. We find that fuzzing history contains rich knowledge of the key mutation strategies that lead to the discovery of unique paths and crashes. More specifically, the mutations from the seed test cases to the mutated ones which trigger unique execution paths might contain common strategies. These strategies lead to the trigger of specific branching behaviors, which might also exist in many other execution paths. Thus, for the unique paths containing these branching behaviors, we might be able to reach them faster in the following fuzzing process, if we can learn these strategies and re-utilize the partial path constraint solutions implicitly carried in them. Furthermore, we might avoid or significantly reduce the repetitive cost of explicitly solving complex path constraints. Our insights rely on the assumption that the same/similar program logics, which are often wrapped into functions and libraries, are repeatedly used within a single program or across different programs. The reuse practice is very common in software development, and the thriving open source communities further promote it.

To further support our insights, we conduct two case studies on several programs. In particular, we study the *cmp* assembly instruction and the associate immediate operands, since they directly control the branching behaviors of a program. Highlights of our findings are as follows. 1) In a program, the same immediate operands are compared with the values of registers multiple times by the instruction *cmp*; and 2) different programs contain the same immediate operands employed by the instruction *cmp*, which account for a non-

Xuhong Zhang is the corresponding author.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23162
www.ndss-symposium.org

negligible number of all the immediate operands used in the *cmp* instruction. In addition, we show the fact that the proportion of the shared basic blocks is non-negligible in the execution paths of different programs from the same vendor. More details of the case studies are introduced in Section III-A. These case studies demonstrate only two types of reuse of program logics to help understand the value of reusing efficient mutation strategies. In practice, there should be more diverse reuse patterns resulting from the reuse practice in software development. Thus, more efficient mutation strategies can be inferred. Therefore, both the intra-trial fuzzing history (history in the current fuzzing process) and inter-trial fuzzing history (history from the previous fuzzing processes which can be from the same or a different program) can be valuable assets to guide effective fuzzing.

To utilize fuzzing history in a fuzzing process, the key challenge is how to capture the *mutation strategies* that trigger unique paths and crashes from the intra- and inter-trial history. In other words, given the input byte values from a seed test case, the learned mutation strategy model should be able to output the corresponding mutated values and the mutation types that have caused the test case to trigger a unique path or crash. Since we consider both intra- and inter-trial fuzzing history, the mutation strategy model needs to support incremental updates, as the intra-trial history is generated in real time in the fuzzing process and is more relevant to the tested program. On the other hand, the model is required to have an appropriate computational cost and high execution speed to ensure the efficiency of the mutation-based fuzzers. As we all know, the execution speed significantly influences the unique path and crash discovery of fuzzers.

To achieve our goal, we propose *Probabilistic Byte Orientation Model* (PBOM) to learn and reuse the efficient mutation strategies from the intra- and inter-trial fuzzing history with fast execution speed. To be specific, we first categorize all the mutation operators into three types: `overwrite`, `delete` and `insert`. For each operator used to mutate one test case, we temporarily record the input byte values, the byte length, the mutation type, and the corresponding mutated byte values. If the mutated test case triggers a new unique path or crash, we store the recorded data into the training set. Then, we utilize the stored data in the training set to construct *inter-PBOM*. Essentially, PBOM uses a hash map to map input byte values to its corresponding efficient mutation strategies. Each mutation strategy contains the mutation type, the mutated byte values, the frequency of this mutation, and the selection probability of this mutation. Note that the selection probability of a mutation is calculated based on its frequency in history.

Furthermore, in order to utilize intra-trial history, we continuously collect the efficient mutation strategies during the fuzzing process, and periodically update the *intra-PBOM*.

Based on the *intra-* and *inter-PBOM*s, we present a novel history-driven mutation framework, named EMS, to utilize the fuzzing history. Specifically, EMS employs the two PBOMs as two additional mutation operators to probabilistically provide the desired mutation byte values and mutation types according to the input byte values and lengths. Both the traditional mutation operators and PBOMs are utilized to find interesting test cases that trigger unique paths and crashes.

EMS is a generic framework that can be applied to most mutation-based fuzzers. In this paper, we apply it to the state-of-the-art fuzzer MOPT and implement the prototype of EMS. We compare the fuzzing performance of EMS with the state-of-the-art fuzzers on 9 real world programs. In total, EMS discovers 130 unique vulnerabilities. Compared to other fuzzers, EMS finds 4.91× more unique vulnerabilities reported by AddressSanitizer (ASan) [1] than the baseline AFL [3], and finds 2.33×, 1.17×, 0.78×, 0.67×, and 0.31× more vulnerabilities than QSYM [72], MOPT [44], MOPT-dict, EcoFuzz [71], and AFL++ [22], respectively.

In summary, we make the following contributions.

- We discover that both intra- and inter-trial fuzzing history contain rich knowledge of the key mutation strategies that lead to the discovery of unique paths and crashes. These mutation strategies implicitly carry partial path constraint solutions and can be used to accelerate the discovery of new paths and crashes sharing similar partial path constraints.

- We propose a lightweight and efficient PBOM to capture the mutation strategies that trigger unique paths and crashes in the intra- and inter-trial history. We further present a novel history-driven mutation framework EMS that employs PBOM as one of the mutation operators to probabilistically provide the desired mutation bytes according to the input ones.

- We implement EMS based on the state-of-the-art fuzzer MOPT and construct the prototype of EMS. Then, we evaluate EMS against AFL, QSYM, MOPT, MOPT-dict, Ecofuzz, and AFL++ on 9 real world programs. The results show that EMS discovers more unique vulnerabilities and line coverage than other fuzzers on most programs. We utilize the standardized benchmark FuzzBench to show the significant coverage performance of EMS. When using different initial seed sets, EMS also finds the most vulnerabilities reported by different sanitizers. Furthermore, we conduct an analysis of the *PBOM operator*'s contribution and the efficient mutation strategies learned from inter-trial history to demonstrate the validity of PBOM. We also conduct an evaluation on the different programs from the same vendor, and show the discovery improvement of EMS with the different *inter-PBOM*s. The close execution speed compared to MOPT also demonstrates the low overhead of EMS.

- We report all of the discovered vulnerabilities to the vendors to improve the programs' security. Also, we will open source EMS at *https://github.com/puppet-meteor/EMS* to facilitate the research in the fuzzing area.

II. Background

A. Mutation-based Fuzzing

The core idea of mutation-based fuzzing is to mutate the prepared test cases and frequently test the target program to see whether the mutated test cases can trigger abnormal behaviors. The general workflow of mutation-based fuzzers is as follows. A fuzzer 1) requires an initial seed set and constructs a queue of seed test cases; 2) selects seed test cases from the queue and randomly mutates the seeds with several kinds of mutation operators; 3) employs the mutated test cases to test the target program, and adds the interesting test cases that trigger new

execution paths or abnormal behaviors into the seed queue; and 4) goes back to step 2) to continue the fuzzing process.

The logic of most mutation-based fuzzers to mutate the test cases is straightforward. For instance, AFL, one of the most well-known mutation-based fuzzers, implements three stages to mutate the test cases: the deterministic stage, the havoc stage, and the splicing stage. In the deterministic stage, AFL utilizes bit- or byte-level mutation operators, e.g., bitflip, byteflip, and byte insertion, to mutate each bit or byte of a seed test case; In the havoc stage, AFL randomly selects the operators multiple times and employs all of them to mutate at the random locations on the seed test case; In the splicing stage, AFL first splices the parts of two seed test cases together to generate one new case, and then enters the havoc stage to employ further mutation operators.

Without analyzing how to solve path constraints, traditional mutation-based fuzzers explore new execution paths blindly by utilizing the randomly mutated test cases to test a program. Because of the straightforward logic, the execution speed of mutation-based fuzzers is fast, leading to effective vulnerability exploration. However, the straightforward logic cannot solve complex path constraints, which limits the fuzzing efficiency. Therefore, plenty of works focus on improving the path coverage and develop coverage-based fuzzing on top of the mutation-based fuzzing.

B. Coverage-based Fuzzing

To address the aforementioned limitation of mutation-based fuzzing, researchers propose to leverage coverage information as feedback to better guide the fuzzing process. Since it is efficient and effective to accurately track the execution paths, and a fuzzer cannot find a vulnerability in a not covered execution path, improving the coverage of execution paths is reasonable to enhance the fuzzing performance.

Several works employ adaptive strategies to improve coverage-based fuzzing. For instance, AFLFast [13] and EcoFuzz [71] focus on adaptively adjusting the execution frequency of each seed test case on different programs. They employ the Markov chain model and adversarial multi-armed bandit model to evaluate the potential of each test case to trigger unique branching behaviors, respectively. Then, they allocate more time to mutate the promising test cases, and vice versa [13], [71]. MOPT proposes that the optimal selection probability distribution of mutation operators is different on different target programs. It provides an iterative scheduling strategy to adaptively adjust the selection probability of each mutation operator according to its efficiency of discovering unique paths and crashes [44].

Another direction of the prevalent coverage-based fuzzing is to integrate mutation-based fuzzing with constraint-solving techniques, e.g., concolic execution. To solve the path constraints, such techniques are supposed to first compile the programs utilizing the powerful instrumentation to trace and collect the path constraints. Then, the constraint-solving techniques need to perform expensive procedures, including formulating path constraints, tracing the data fields that influence the target constraints, and calculating the numerical interval of the data fields that can trigger different states of the constraints. Thus, both constraint collecting and solving can be expensive. Using the constraint-solving techniques to solve a path constraint usually requires significant computational cost and time, which might reduce the performance of fuzzing [74]. To overcome the challenges, several works improve the fuzzing performance by selectively assigning difficult paths to concolic execution [63], [74]. Yun et al. developed a fast and lightweight concolic execution engine named *QSYM* [72] to improve the performance. Impressively, Angora employs the gradient descent algorithm, and several data tracking and analysis techniques to solve path constraints faster in place of concolic execution [17].

Recently, multiple researches try to discover the valuable byte locations in seed test cases with machine learning techniques. For instance, *Augmented-AFL* utilizes the prior experience during one trial to train a neural network model, and then employs the model to predict the good locations in a test case that are going to trigger unique crashes or paths after mutation [53]. She et al. utilized the byte data of test cases as the input and the branching behaviors of a target program as the output to train a neural network model, and then employed the gradient-guided algorithm on the model to locate the bytes in a test case that influence the branching behaviors most [60]. Due to their design philosophy, these tools first need to run AFL on each target program to obtain the training dataset, then the model can be trained.

In summary, these state-of-the-art coverage-based fuzzers mainly focus on improving fuzzing performance via a better exploration of the test case space. However, *an evident limitation is that existing fuzzers lack an adequate utilization of the history information in the intra trial.* Moreover, *they do not utilize the results of the past experiments, i.e., the inter-trial history.* We observe that the intra- and inter-trial history can be quite valuable in guiding effective fuzzing, which is the main focus of this paper.

III. DESIGN OF EMS

In this section, we introduce the motivation of this paper, show the framework of EMS, and then present the design of the proposed Probabilistic Byte Orientation Model (PBOM).

A. Why Intra- and Inter-Trial History Matters

To some extent, the intra-trial history is considered by existing fuzzers containing adaptive strategies [13], [44], [54], [71]. However, they mainly focus on the high level heuristics obtained from the intra-history to guide the seed selection and generation process, which lack a fine-grained reusing of the employed mutation strategies that effectively trigger unique paths or crashes. The motivation is that different execution paths of one program may contain the same function call. In addition, common values, such as -1, 0 and 1, always appear in the program constraints. Thus, different execution paths might have the same particular values in the path constraints, e.g., magic bytes and checksums, that control the branching behavior.

On the other hand, the inter-trial fuzzing history is also a valuable asset to guide the fuzzing. First, the inter-trial fuzzing history from the same program has a similar contribution to the intra-trial history. Then, it can direct the fuzzing to resolve path constraints that have already been resolved for the same

TABLE I: Statistics of the number of immediate operands and their usages by the *cmp* instruction.

		Singular[a]	Repetitive[b]	Total
pdfimages	Number of immediate operands	15	21	36
	Number of usages of immediate operands	15	46	61
objdump	Number of immediate operands	25	34	59
	Number of usages of immediate operands	25	195	220
nasm	Number of immediate operands	6	5	11
	Number of usages of immediate operands	6	35	41

[a]If an immediate operand is used only once, it is singular.
[b]If an immediate operand is used more than once, it is repetitive.

program, e.g., an initial seed set with better path coverage can improve the fuzzing performance. In addition, the inter-trial fuzzing history from a different program could also be useful due to the following observations. Many software platforms provide a unified development framework and underlying libraries in order to improve the quality and efficiency of program development [21], [32], [34], [46], [56], [65], [73]. For instance, nearly 10% of IoT firmware images employ the same open-source library `BusyBox` [4]. Similarly, there may be the same path constraints in different programs because of the shared libraries.

*1) **Immediate Operand Analysis**:* To verify our hypothesis, we conduct a case study to analyze the assembly codes in three different programs `pdfimages`, `objdump` and `nasm`. In particular, we investigate the immediate operands and their usages in the *cmp* assembly instructions, since they directly control branching behaviors of a program and are closely related to path constraints. Moreover, there are many universal immediate operands in different programs such as "$\$0x00000000$" and "$\$0x00000001$", which will be beneficial to our analysis and weaken the impact of the rare immediate operands. To improve the persuasion of the analysis, **we do not include the universal immediate operands, which are defined as the interesting values of AFL and are used frequently to mutate input files, in most of our analysis**. The results are shown in Table I and Fig. 1, from which we have the following conclusions.

• **The same immediate operand influences the control flow and data flow multiple times in a program.** For instance, `pdfimages` utilizes `cmap` tables to map the character codes to the glyph index values used in the font. This introduces the same magic bytes in different execution paths. As a result, we find that the immediate operand "$\$0x636d6170$" is compared with the values of different registers by the assembly instruction *cmp* at different code positions of `pdfimages`. These repetitive immediate operands influence the control and data flow of a program.

• **The repetitive immediate operands account for the vast majority in each program.** To figure out how many immediate operands are used multiple times, we count how many times each unique immediate operand is used in the assembly instruction *cmp* and then classify these immediate

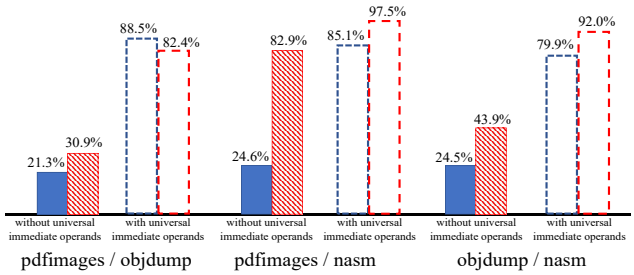

Fig. 1: The percentage of usages of the same immediate operands, i.e., the number of usages of the same immediate operands employed in both programs divided by the number of usages of all the immediate operands in each program.

operands into two categories, based on whether they are used more than once. If an immediate operand is used more than once, we consider it as a repetitive immediate operand. Otherwise, it is a singular immediate operand. The results are shown in Table I, from which we have the following observations. 1) The number of repetitive immediate operands is larger than the singular immediate operands on most programs. For instance, in `objdump`, the number of repetitive immediate operands is 1.36× of the singular immediate operands. 2) The total number of usages of the repetitive immediate operands is significantly larger than that of the singular immediate operands. The aforementioned observations indicate that the influence of repetitive immediate operands is more significant than singular immediate operands on the branching behaviors of a program.

• **The proportion of the usages of the same immediate operands employed in two programs cannot be ignored.** In order to find out the correlation of the immediate operands' usages in different programs, we count the total number of usages of the same immediate operands used in *cmp* in a pair of programs. The percentage of the same immediate operands' usages, i.e., the number of usages of the same immediate operands employed in both programs divided by the number of usages of all the immediate operands in each program, is shown in Fig. 1, from which we have the following observation. The same immediate operands cannot be ignored in different programs. For instance, the usages of the same immediate operands account for 24.6% and 82.9% in `pdfimages` and `nasm`, respectively. The proportion of the usages of the same immediate operand achieves more than 79.9% in all the programs, if the universal immediate operands are included.

In addition to the easily accessible and reproducible observations as described above, more implicit efficient strategies can be reused because of the implementation of the same program logic. For instance, the same efficient mutation strategy can trigger a similar branching behavior in the shared codes when fuzzing different programs from the same vendor.

*2) **Shared Code Analysis**:* To figure out the number of the shared codes used in different programs from the same vendor, we count the number of the shared basic blocks triggered in the execution paths of different programs in this case study. To achieve this, we utilize MOPT to fuzz `pdfimages`, `pdftotext`, `pdfinfo`, `objdump`, `addr2line`, and `objcopy` from `xpdf-4.02` and `binutils-2.28`, whose settings are shown in Table X.

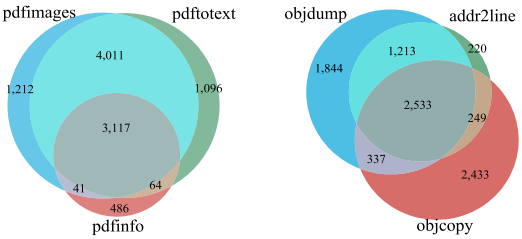

Fig. 2: The number of shared basic blocks and unique basic blocks triggered in three programs from the same vendor.

The experiment settings are the same in Section V-A, and each evaluation lasts for 5 hours. Then, we record the triggered basic blocks in the execution paths of each program, and count the number of the shared triggered blocks in different programs from the same vendor. The number of the shared basic blocks triggered in different programs and the number of the unique basic blocks triggered in each program from the same vendor are shown in Fig. 2, from which we have the following conclusion. **The proportion of the shared basic blocks is non-negligible in different programs from the same vendor.** For instance, the number of the shared basic blocks is 7,128 (4,011 + 3,117) triggered in `pdfimages` and `pdftotext`, whose proportion is 85.05% and 86.00% in each of them, respectively. These shared basic blocks contain the same constraints to be solved for a fuzzer in the inter trials. However, the state-of-the-art fuzzers ignore the valuable mutation strategies that have solved these same constraints in the inter trials, and thus cannot leverage the inter-trial fuzzing history in the fuzzing scenarios like parallel fuzzing and continuous fuzzing.

Motivation. Based on the analysis above, we discover that 1) most of the immediate operands employed by *cmp* are repetitive in one program; and 2) different programs have the same immediate operands, which are the majority of all the operands. Therefore, it is possible that the immediate operands in the current path constraints are the same ones in other constraints, which might be solved in intra- and inter-trials. From another aspect, different programs developed by the same vendor invoke the same codes and contain the shared basic blocks in their execution paths, introducing more kinds of the same path constraints. Thus, many efficient mutation strategies can be concluded from the trials that fuzz on the same and different programs. However, state-of-the-art fuzzers do not consider utilizing repetitive historical information like this. Therefore, it is necessary to construct a new fuzzing solution to properly utilize the intra- and inter-trial fuzzing history.

B. *Framework of* EMS

The goal of EMS is to learn the *mutation strategies* from intra- and inter-trial fuzzing history, which lead to the discovery of unique branching behaviors. In other words, given the input byte values from a seed test case, EMS aims to provide efficient strategies to mutate the input byte values, which finally makes the test case more likely to trigger unique paths and crashes. Additionally, the aforementioned challenges, such as the computational cost and the execution speed, also need to be addressed.

In the following subsections, we mainly introduce the framework of EMS and the design of `PBOM`. The relevant symbols are defined as follows.

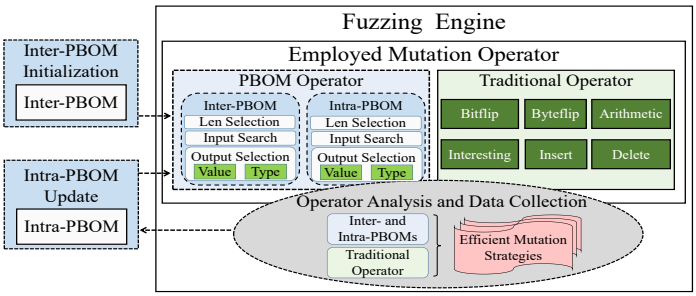

Fig. 3: The framework of EMS.

- in is the input byte values read from a test case, where L is the byte length of in;
- $(out, type)$ is a mutation operator corresponding to a unique in, where out is the output byte values with the same L, and $type$ is the mutation type;
- $\mathbb{MO}$ is a set of mutation operators corresponding to in, i.e., $(out, type) \in \mathbb{MO}$;
- $(out, type, F, P)$ is a probabilistic output mutation strategy corresponding to in, where F is the frequency of a unique $(out, type)$ under in, and P is the selection probability of this output strategy under in;
- $\mathbb{T}$ is the set of $(out, type, F, P)$ corresponding to in, i.e., $(out, type, F, P) \in \mathbb{T}$;
- $\mathbb{IN}$ is the set of input byte values, i.e., $in \in \mathbb{IN}$;
- `favorite_list` is a set of the recorded locations, at which the test cases are mutated to trigger unique paths and crashes in the intra trial;
- *Inter-`PBOM`* is the `PBOM` trained by the inter-trial fuzzing history, and *intra-`PBOM`* is the `PBOM` trained by the intra-trial fuzzing history.

The entire framework of EMS is shown in Fig. 3. To achieve the goal, EMS constructs *inter-* and *intra-`PBOM`*s to learn and utilize inter- and intra-trial history, respectively. At the beginning of fuzzing, EMS invokes the *Inter-`PBOM` Initialization* to construct *inter-`PBOM`*. During the fuzzing process, it employs the *`PBOM` Operator* to mutate test cases with *inter-* and *intra-`PBOM`*s. Further, EMS utilizes the *Operator Analysis and Data Collection* to continuously collect the intra-trial history, and periodically invokes the *Intra-`PBOM` Update* to update *intra-`PBOM`* with the new collected intra-trial history. The details are as follows.

Inter-`PBOM` Initialization. In order to utilize the inter-trial history, EMS employs the *Inter-`PBOM` Initialization* to construct *inter-`PBOM`* at the beginning of fuzzing, which connects input byte values with the efficient output mutation strategy and assigns appropriate selection probability learned from the inter-trial fuzzing history. To achieve this, EMS 1) first extracts input byte values in and all the corresponding mutation strategies $\mathbb{MO}$ from inter-trial history; 2) counts the frequency F of each unique $(out, type) \in \mathbb{MO}$ under in; 3) calculates the selection probability P of $(out, type)$ according to F under in; and 4) constructs *inter-`PBOM`* by utilizing in as the input and the corresponding mutation strategy $(out, type, F, P)$ as the output.

`PBOM` Operator. EMS implements both *inter-* and *intra-`PBOM`*s to utilize efficient mutation strategies learned from the inter- and intra-trial history, which are employed by the

PBOM Operator as shown in Fig. 3. Taking *inter-PBOM* as an instance, the process of the *PBOM Operator* is as follows: 1) **len selection**: EMS probabilistically selects the length L of input byte values to be mutated, according to the proportion of each length in $\mathbb{IN}$. Then, EMS reads the contiguous input byte values in of the selected length L from a random location of one test case, or from the stored preferred location; 2) **input search**: EMS utilizes in as the input of *inter-PBOM*, and searches the corresponding output mutation strategies; and 3) **output selection**: EMS probabilistically selects one output mutation strategy $(out, type, F, P)$ according to P. Finally, EMS mutates once at the selected location according to the output byte values out and mutation type $type$, i.e., overwriting in with the selected out, deleting in, or inserting out in front of in.

Operator Analysis and Data Collection. In each mutation process of a test case, EMS records the following data of each operator (including fuzzer's traditional mutation operators and the *PBOM operator* using *inter-* and *intra-PBOM*s): 1) the original byte values as in; 2) the mutation type as $type$; 3) the mutated byte values as out; and 4) the mutated location. Then, if the mutated test case triggers a new unique path or crash, EMS considers the recorded in and $(out, type)$ of all the used operators in this mutation process as efficient mutation strategies, and employs them to update *intra-PBOM* lately.

Intra-PBOM Update. In the *Intra-PBOM Update*, EMS analyzes efficient mutation strategies of the intra-trial history collected from the previous component. For the first time to invoke the *Intra-PBOM Update*, EMS follows the same process of the *Inter-PBOM Initialization* to construct *intra-PBOM*. Then, EMS periodically invokes the *Intra-PBOM Update* to update *intra-PBOM* with the newly collected mutation strategies. Therefore, *intra-PBOM* is trained by the mutation strategies that have played roles during intra-trial fuzzing process.

C. Probabilistic Byte Orientation Model

In this subsection, we describe the data structure and probability algorithm of PBOM in detail.

As shown in Fig. 4, in order to prevent the execution speed of the fuzzer from decreasing, we construct *inter-* and *intra-PBOM*s with two hash maps, respectively. The process to construct a hash map for a PBOM is as follows. First, we utilize the unique hash of input byte values in as the index of a hash map. In each index, EMS maintains a list of index nodes for unique in with the same hash. Thus, to locate a node for specific in, EMS calculates the index of in and searches the blue node with the corresponding in as shown in Fig. 4. To add a new node for a new unique in, EMS calculates the index of in and adds a new index node at the end of the list; Second, we construct a linked list for each unique in to store the corresponding output mutation strategies $\mathbb{T}$. Each mutation node in the linked list of in stores 1) a unique mutation operator containing the output byte values out and mutation type $type$, and 2) the frequency F and the selection probability P of the mutation operator $(out, type)$ under this in. To add a new mutation node for a new unique $(out, type)$, EMS locates the index node for the corresponding in and adds a new mutation node at the end of in's linked list.

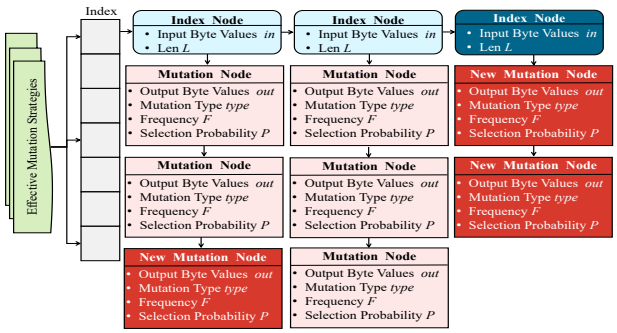

Fig. 4: The data structure of PBOM.

To construct *inter-PBOM*, first, EMS collects inter-trial history by employing normal fuzzers, e.g., AFL and MOPT, to fuzz programs. Then, EMS constructs the data structure as shown in Fig. 4 and updates each node's selection probability P in the linked list of in. In the following paragraphs we introduce the selection probability algorithm of *inter-PBOM*.

Since the efficient mutation strategies are triggered by the traditional mutation operators of normal fuzzers when collecting inter-trial history, many of them are generated by simple operators, such as flipping a bit, or increasing 1 on value of a byte. Moreover, the mutation strategies can be collected from multiple different programs and can be collected for a long time. In summary, the number of collected mutation strategies can be large, and most strategies are triggered by the simple operators because of their heavy use.

Therefore, the higher the frequency F of $(out, type)$ is, the more easily mutation-based fuzzers can generate $(out, type)$ from in with traditional mutation operators in the inter-trial history. On the contrary, the low-frequency $(out, type)$ can be constructed by rare mutation operators, e.g., inserting the specific byte values into a seed test case. It is less useful if *inter-PBOM* always reproduces the simple operators. Thus, *inter-PBOM* assigns more selection probability P to $(out, type)$ that appears less frequently. Based on the frequency F of each $(out, type, F, P) \in \mathbb{T}$, the following formulas calculate the probability distribution $\mathbb{P}$, where p is the weight of $(out, type)$ to calculate P under in:

$$
\begin{aligned}
p_i &= 1 - \frac{F_i}{F_1 + F_2 + ... + F_{n-1} + F_n} \\
&= 1 - \frac{count((out_i, type))}{\sum_{(out_k, type) \in \mathbb{MO}} count((out_k, type))} \\
P_i &= \frac{p_i}{p_1 + p_2 + ... + p_{n-1} + p_n} \\
&= \frac{\sum_{(out_k, type) \in \mathbb{MO}} count((out_k, type)) - count((out_i, type))}{(n-1) \times \sum_{(out_k, type) \in \mathbb{MO}} count((out_k, type))}
\end{aligned}
\tag{1}
$$

According to Formula 1, *inter-PBOM* assigns higher selection probability P to $(out, type)$ with fewer frequencies F. Then, it constructs the selection probability distribution $\mathbb{P}$ for $\mathbb{MO}$ that selects the rare $(out, type)$ more often to `overwrite`, `delete` or `insert` a seed test case.

On the other hand, all the history data collected for *intra-PBOM* are the efficient mutation strategies that are valid on the current program. Then, one output operator $(out, type)$ with the larger frequency F means that it has triggered more unique

paths and crashes so far. Therefore, in order to employ efficient mutation strategies more times on the current program, *intra-PBOM* allocates larger selection probability to $(out, type)$ with the larger frequency F, whose formula is as follows.

$$P_i = \frac{F_i}{F_1 + F_2 + ... + F_{n-1} + F_n} = \frac{count((out_i, type))}{\sum_{(out_k, type) \in \mathbb{MO}} count((out_k, type))} \quad (2)$$

By constructing a linked list containing $\mathbb{T}$ for each unique in with the aforementioned formulas, each of *inter-* and *intra-PBOM*s utilizes efficient mutation strategies collected from inter- and intra-trial history, respectively.

IV. Implementation of EMS

In this section, we introduce the detailed implementation of EMS. As aforementioned, since AFL-based fuzzers will enter the havoc stage after generating a new test case in the splicing stage, it is needless to add the *PBOM operator* in the splicing stage. Therefore, EMS, which is constructed based on MOpt, implements the *PBOM operator* in the deterministic and havoc stages to utilize efficient mutation strategies. As shown in Fig. 5, the workflow of EMS is as follows.

1) At the beginning of fuzzing: EMS employs the *Inter-PBOM Initialization* to construct *inter-PBOM* and update the selection probability distribution for each unique in, which is the only update of *inter-PBOM* in the entire fuzzing process.

2) In the fuzzing process: As shown in Fig. 5, EMS invokes the *PBOM operator* in two stages. It continually invokes the *Operator Analysis and Data Collection* in the generation of each mutated test case. The *Intra-PBOM Update* will be invoked periodically in the workflow.

PBOM Operator in the Deterministic Stage. The first place to invoke the *PBOM operator* is in the deterministic stage. Since *intra-PBOM* is empty before the first update, EMS only employs *inter-PBOM* in the deterministic stage. Similar to other operators in the deterministic stage, EMS reuses each efficient mutation strategy once at a time to mutate a test case on the same input byte values. In other words, EMS utilizes each $(out, type)$ under the same in to mutate a test case once at a time. The procedures are as follows.

1) EMS randomly selects input byte values in with the length L on the original test case, and searches in and L in the index nodes of the hash map;
2) If in and L have been matched, EMS employs one $(out, type, F, P)$ stored in the linked list once at a time to mutate the original test case. To be specific, EMS first reads the output operator $(out, type)$ in the mutation node; Second, EMS mutates once according to $(out, type)$ to overwrite the original byte values in with out, delete in, or insert out in front of in; Then, it tests a target program with the mutated test case in order to trigger a new path or crash; Finally, EMS restores the test case to the original one, and starts the next mutation if there is an unused $(out, type)$ in the linked list.

PBOM Operator in the Havoc Stage. In the havoc stage, AFL-based fuzzers randomly select one mutation operator to mutate byte values at a random location, which will be executed multiple times to construct one mutated test case. Similarly, in the havoc stage of EMS, *inter-* and *intra-PBOM*s are considered as two different mutation operators, whose selection probability is decided by the particle swarm optimization algorithm, respectively. If EMS selects one of PBOMs to mutate the test case, it will perform the following procedures.

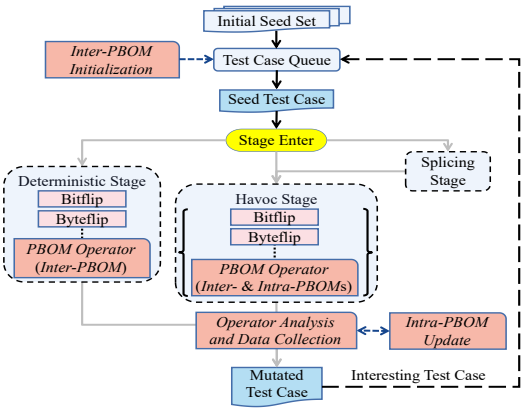

Fig. 5: The workflow of EMS.

1) In order to decide the length of input byte values in, EMS analyzes the proportion of each length in $\mathbb{IN}$, and probabilistically selects one length L according to the proportion;
2) To select a mutation location of a test case, with a small probability EMS randomly selects one from `favorite_list`, i.e., the stored location that has been mutated and has triggered the unique path or crash in the intra trial. Otherwise, EMS randomly selects one from all the possible locations of a test case. Then, EMS reads the input byte values in with the selected length L, and searches in the hash map to find the matched index node;
3) Once EMS matches in and L in the index node of the hash map, it probabilistically selects one output strategy $(out, type, F, P)$ stored in the linked list of the index node according to the selection probability P. Then, EMS mutates at the selected location of the test case according to out and $type$.

Two invocations of the *PBOM operator* at different stages ensure EMS can employ the learned mutation strategies in different ways. On the other hand, the following two invocations enable the update of *intra-PBOM*.

Operator Analysis and Data Collection. For each mutation operator that is used, including the traditional mutation operators and the *PBOM operator* using PBOMs, EMS temporarily records the used mutation strategies, which contains in, L, out, $type$, and the mutation location. Note that only the data with the length L of 1, 2 and 4 will be recorded, which cover most mutation results caused by the majority of potential mutation operators in AFL-based fuzzers. If the mutated test case triggers a new unique path or crash, EMS stores the temporarily recorded strategies in the training set for the update of *intra-PBOM*, and adds the mutation location to `favorite_list`.

Intra-PBOM Update. When the execution number of EMS reaches the preset value during the fuzzing process, EMS

periodically updates F and P in the mutation nodes and adds new nodes in the linked lists of *intra-PBOM* with the training set. For each recorded mutation strategy containing in, L, out, and $type$, the update process is as follows.

1) EMS calculates the hash map's index according to in. Then, EMS searches index nodes in the index to match in and L as described in Section III-C. If it is not matched, EMS adds a new node at the end of this index to store in and L;
2) EMS searches in the linked list of the matched index node to match out and $type$. If it is not matched, EMS adds a new mutation node to store out, $type$ and the frequency F with the initial value 0. Then, F increases 1;
3) After EMS traverses all the training data and constructs a new hash map of *intra-PBOM*, EMS traverses all the index nodes of the entire hash map, and updates the selection probability P of the mutation nodes under each index node according to Formula 2. Then, EMS empties the training set and continues the fuzzing process.

V. Evaluation

In this section, we evaluate the fuzzing performance of EMS following the guidelines in [38], [41].

A. Experiment Setup

Compared fuzzers. We compare EMS with the state-of-the-art open-source mutation-based fuzzers, including AFL [3], QSYM [72], MOpt [44], MOpt-dict, EcoFuzz [71], and AFL++ [22]. We pick these fuzzers for the following reasons. First, AFL is one of the most famous fuzzers in academia and industry, which can be a baseline for other fuzzers. Second, QSYM implements a fast concolic execution engine that solves path constraints and improves its fuzzing performance. Then, MOpt and EcoFuzz are two state-of-the-art fuzzers that utilize adaptive strategies to improve the seed selection and generation process. In this paper we implement EMS based on MOpt. Recently, Fioraldi et al. incorporated multiple state-of-the-art fuzzing researches and presented AFL++ [22], which is one of the best fuzzers evaluated by the well-known open source benchmark FuzzBench [5]. We enable most configurations of AFL++ in our evaluation, including the CmpLog instrumentation, the RedQueen mutator and the MOpt mutator. By reading the immediate operands of each target program as a dictionary and leveraging them as an additional mutation operator, we then construct MOpt-dict based on MOpt. We compare EMS against MOpt-dict to show the effectiveness of PBOMs. Note that the fuzzing performance of EMS and other fuzzers can also be improved with a dictionary. Each MOpt-based fuzzer employs '-L 5' as the configuration in the evaluation. Since these fuzzers represent different solutions in the fuzzing field, the performance of EMS can be fairly assessed compared with other solutions.

Real world target programs. Following the guidance of the state-of-the-art research UniFuzz [41], we evaluate the aforementioned fuzzers on 9 open-source linux programs as shown in Table II, which are selected from UniFuzz [41] or are widely used in the state-of-the-art fuzzing research [17], [23], [24], [44], [70], [71]. We utilize `pdfimages` and `pdftotext` to evaluate the performance of *inter-PBOM* on the programs from the same vendor. From another aspect, other target programs come from different vendors with different functionalities. Thus, we comprehensively evaluate each fuzzer from the perspectives of vendors and code logic, which make the analysis more all-sided. Furthermore, the programs are prevalent and widely-used open-source programs. Hence, evaluating the security of them is meaningful for the vendors and users.

TABLE II: Target programs evaluated in the experiments.

Target	Source	Input format	Test instruction
pdfimages	xpdf-4.02	pdf	@@ /dev/null
pdftotext	xpdf-4.02	pdf	@@ /dev/null
objdump	binutils-2.28	binary	-S @@
infotocap	ncurses-6.2	txt	@@ -o /dev/null
cflow	cflow-1.6	C files	@@
nasm	nasm-2.14.03rc2	asm	-f bin @@ -o /dev/null
w3m	w3m-0.5.3	txt	@@
mujs	mujs-1.0.2	javascript	@@
mp3gain	mp3gain-1.5.2-r2	mp3	@@

Initial seed sets. Following the same seed collection and selection procedure as in UniFuzz [41] and in the previous works [44], [51], [52], [55], for each program, we 1) use 100 input files provided by the open-source dataset in UniFuzz; or 2) randomly download 100 input files from the Internet according to its required input file format as shown in Table II. Thus, each fuzzer employs the same 100 input files as an initial seed set to fuzz the same program.

Experiment settings. Since mutation-based fuzzers employ a random process to mutate test cases, the fuzzing performance fluctuates to a certain degree. To mitigate the impact of performance fluctuation on the evaluation, we implement the following two experiment settings: First, each evaluation lasts for 168 hours. Relatively long experiment time can reduce the influence of fuzzing randomness. Second, each evaluation is repeated 16 times and statistically analyzed, which can improve the reliability of the conclusions.

Each fuzzing evaluation runs on a docker container configured with 1 CPU core of 2.40GHz E5-2680 V4 and the OS of 64-bit Ubuntu 16.04 LTS. We run fuzzing experiments on 10 servers, each of which has two E5-2680 V4 CPUs and 256GB memory. In total, we spend several months in order to get reliable and reproducible results.

Evaluation metrics. As aforementioned, we evaluate 7 fuzzers with different implementation logics. Their metrics to count the abnormal behaviors and unique execution paths may have differences. To eliminate the differences, we employ the following two metrics to evaluate the fuzzers' performance.

The first metric is the number of unique vulnerabilities reported by ASan. The reason is as follows. Many crashes explored by fuzzers may not trigger unique vulnerabilities. Thus, finding more crashes does not mean that a fuzzer discovers more unique vulnerabilities. On the contrary, the number of unique vulnerabilities is a more direct and important metric to measure the performance of fuzzers.

For the crashes that trigger real vulnerabilities, ASan reports the stack traces of target programs, which are widely used for deduplication in the Common Vulnerabilities and

TABLE III: The number of unique vulnerabilities after deduplication in 16 trials.

	AFL	QSYM	MOPT	MOPT-dict	EcoFuzz	AFL++	EMS
pdfimages	2	3	4	5	7	13	**15**
pdftotext	2	6	9	9	9	6	**13**
objdump	5	11	3	6	18	22	**30**
infotocap	0	0	6	6	3	**7**	**7**
cflow	1	4	6	7	6	7	**9**
nasm	0	0	11	15	13	**20**	18
w3m	0	1	0	1	0	0	**11**
mujs	4	3	4	6	6	6	**7**
mp3gain	8	11	17	18	16	18	**20**
total	22	39	60	73	78	99	**130**

Exposures (CVE) dataset [6] and debugging for vendors. By filtering stack traces to obtain the unique function call sequences, we can collect the explored unique vulnerabilities on target programs. In this paper, we extract the top three function calls in the stack traces to de-duplicate vulnerabilities following the guidelines in [38], [41]. We analyze the number of unique vulnerabilities triggered in each trial, and present the unique vulnerabilities after deduplication in all the trials as the total unique vulnerabilities.

The second metric is the line coverage analyzed by afl-cov [2], which is widely used in recent researches [24], [41], [60], [71]. As we all know, different fuzzers may have different metrics to maintain seed test cases in the queue. For instance, QSYM needs to run hybrid fuzzing with AFL, and they will sync test cases with each other. The number of test cases in the queue cannot show the program execution coverage of a fuzzer. Thus, we utilize afl-cov [2] to analyze the line coverage of the source codes triggered by each fuzzer on a target program. Employing line coverage to evaluate fuzzing performance can eliminate the diversity between different fuzzers, which provides a more accurate and uniform metric.

B. EMS *Buildup*

In this subsection, we introduce the settings for *inter-* and *intra-*`PBOM`s of EMS.

In order to train the initial *inter-*`PBOM`, we 1) spend 5 hours employing MOPT to fuzz `pdfimages` once, which discovers 5,270 unique paths and 143 unique crashes; and 2) obtain 137,337 efficient mutation strategies that trigger the unique paths and crashes for 43,758 unique input bytes, in which the number of mutation strategies for input bytes with the length of 1,2,4 is 70,007, 27,744 and 39,586, respectively. It takes 9 seconds to construct initial *inter-*`PBOM` with these mutation strategies.

To fuzz a target program in the benchmark, EMS first loads *inter-*`PBOM` at the beginning of fuzzing to utilize the inter-trial history learned from `pdfimages`. Then, EMS fine tunes *intra-*`PBOM` with the efficient mutation strategies that have triggered the unique paths and crashes so far to further utilize the intra-trial fuzzing history learned from each target.

C. Unique Vulnerability and Line Coverage Discovery

Unique vulnerability. In this subsection, we analyze the discovered vulnerabilities of each fuzzer with ASan. The results

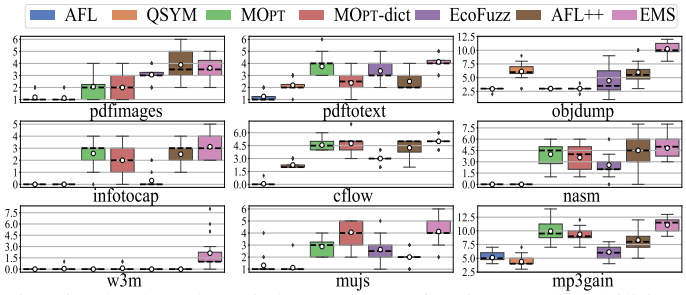

Fig. 6: The boxplot of the number of unique vulnerabilities reported by ASan in 16 trials, where '○' and '– –' represent the mean and median, respectively. Y-axis: the number of unique vulnerabilities discovered in each trial.

TABLE IV: The published CVE IDs found by each fuzzer.

	CVE ID	AFL	QSYM	MOPT	MOPT-dict	EcoFuzz	AFL++	EMS
pdfimages	CVE-2019-17064	●	●	●	●	●	●	●
	CVE-2019-9588							●
pdftotext	CVE-2019-16088	●	●	●	●	●	●	●
	CVE-2019-9588						●	
objdump	CVE-2017-8396	●			●	●	●	●
	CVE-2017-8398		●					●
	CVE-2017-14930		●					
	CVE-2017-16831		●					
	CVE-2018-7568					●	●	●
	CVE-2018-1000876							●
	CVE-2019-9072		●					
	CVE-2019-17450		●					
cflow	CVE-2019-16165	●	●	●	●	●	●	●
	CVE-2019-16166						●	
	CVE-2020-23856			●	●	●	●	●
nasm	CVE-2018-19755			●	●	●	●	●
	CVE-2018-20535			●	●		●	●
	CVE-2018-20538			●	●			●
	CVE-2019-20334						●	●
mujs	CVE-2017-5628				●	●		●
	CVE-2018-6191	●	●	●	●	●	●	●
mp3gain	CVE-2017-14406	●	●	●	●	●	●	●
	CVE-2017-14407	●	●	●	●	●	●	●
	CVE-2017-14409						●	
	CVE-2017-14410				●		●	●
	CVE-2019-18359		●					●
	total	7	12	10	13	11	16	**19**

of unique vulnerabilities are shown in Table III and Fig. 6, from which we have the following conclusions.

• As shown in Table III, EMS finds the most vulnerabilities on most target programs. Specifically, it finds 31 and 52 more unique vulnerabilities than the second best and third best fuzzer, respectively. EMS finds 4.91× more unique vulnerabilities than the baseline AFL. Only AFL++ finds 2 more unique vulnerabilities than EMS on `nasm`. Interestingly, EMS finds more unique vulnerabilities than other fuzzers on `pdftotext`, which partly demonstrates the contribution of the *inter-*`PBOM` trained on a program from the same vendor.

• We can learn from Fig. 6 that EMS achieves the best vulnerability discovery performance in most trials on several target programs. For instance, the mean of EMS is higher than other fuzzers on `pdftotext` and `infotocap`. The median of EMS is higher than others on `objdump` and `mp3gain`. The results demonstrate that EMS has a higher probability of finding more unique vulnerabilities in a single trial.

• Interestingly, although we construct EMS based on MOPT, EMS discovers 70 more unique vulnerabilities that cannot be found by MOPT in total. EMS finds more unique vulnerabilities than MOPT and MOPT-dict on all the target programs. This demonstrates the practical utility of EMS in mutation-based fuzzing. The mutation strategies learned by EMS can improve the vulnerability discovery performance of mutation-based fuzzers.

In order to verify the validity of the discovered vulner-

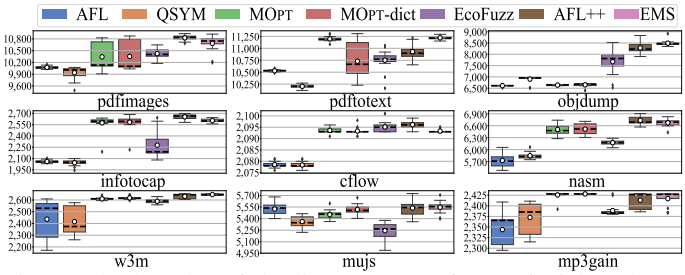

Fig. 7: The boxplot of the line coverage from 16 trials, where '○' and '– –' represent the mean and median, respectively. Y-axis: the line coverage discovered in each trial.

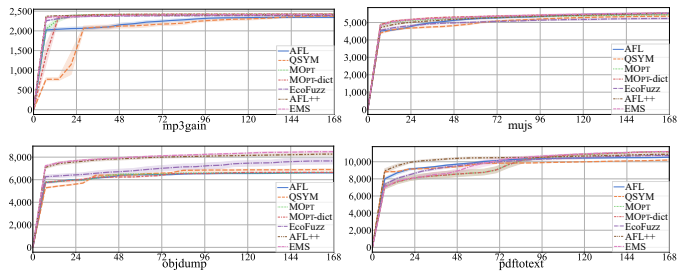

Fig. 8: The line coverage growth on 4 programs discovered by each fuzzer over 168 hours. Each coverage interval with a different color shows the mean and 95% confidence interval for a fuzzer. Y-axis: the number of covered code lines.

abilities for each fuzzer, we utilize the published Common Vulnerabilities and Exposures (CVE), which are confirmed by the vendors and seriously threaten the services and users before being patched, as the ground truth to measure the performance of each fuzzer on serious vulnerability discovery. To achieve this, following the guidance in UniFuzz [41], we 1) manually collect the stack traces of the published CVE IDs and their Proof-of-Concept (PoC) exploits from the CVE dataset [6] for each program; 2) leverage the PoC exploits to reproduce the stack traces of the CVE IDs on our instrumented target program; 3) compare the vulnerability type and the top three stack traces between the CVE IDs and the vulnerabilities found by each fuzzer; and 4) record the published CVE IDs found by each fuzzer. Since no fuzzer triggers any published CVE ID on `infotocap` and `w3m`, we show the triggered CVE IDs for the remaining target programs. The results can be found in Table IV, from which we have the following observation. **EMS achieves the best CVE discovery performance compared to other fuzzers.** Specifically, EMS finds 19 unique CVE IDs in total on 7 programs, and finds the most CVE IDs on `pdfimages`, `nasm`, `mujs`, and `mp3gain`, respectively. These CVEs seriously threaten the security of the program. For instance, CVE-2018-1000876 triggered by EMS publishes an integer overflow vulnerability in `objdump` that can result in a heap overflow. Thus, the results demonstrate the validity and efficiency of EMS on serious vulnerability discovery. As for the other vulnerabilities that do not match the stack traces of the published CVE IDs, we have reported them to the vendors in order to patch the unexpected unique crashes.

Line coverage. The results of line coverage discovered by each fuzzer are shown in Fig. 7, from which we have the following conclusions. As shown in Fig. 7, the line coverage achieved by EMS is in the front rank on most programs. For instance, the median of EMS is significantly higher than other fuzzers on `objdump` and `w3m`. As for the mean line coverage,

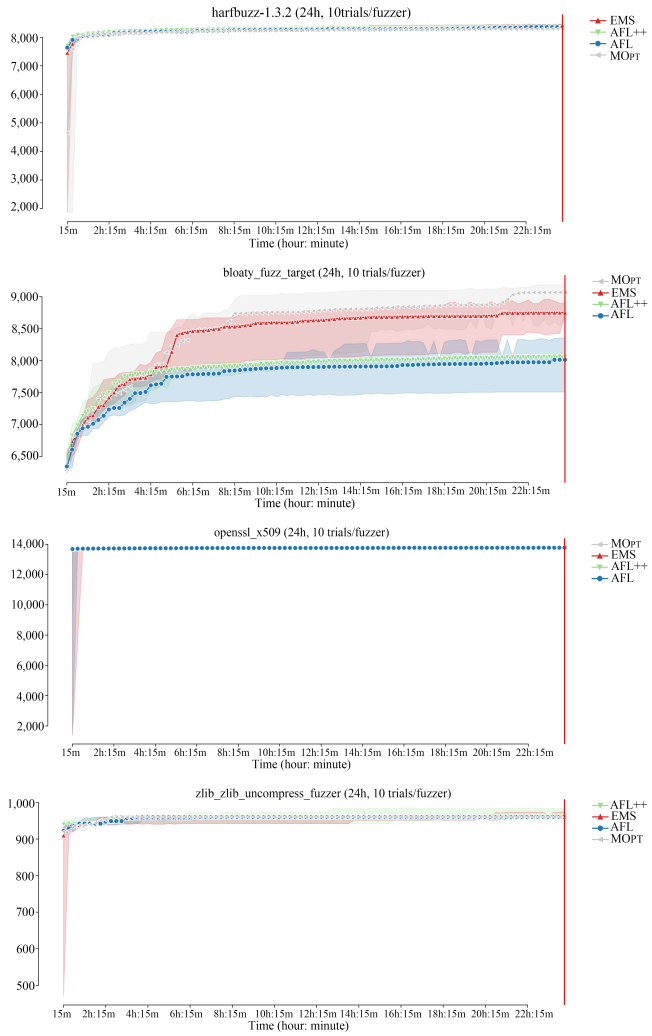

Fig. 9: The discovered region coverage growth over 24 hours evaluated on FuzzBench. Each coverage interval with a different color shows the mean and confidence interval for a fuzzer. Y-axis: the number of discovered region coverage.

EMS achieves the top two on 6 out of 9 programs. Therefore, the results in Fig. 7 demonstrate the significant line coverage performance of EMS on most target programs.

Furthermore, we record the line coverage growth of each fuzzer over 168 hours on `pdftotext`, `objdump`, `mujs`, and `mp3gain`, which is shown in Fig. 8. Then, we have the observation that **the line coverage of EMS grows faster than others on these 4 programs over the 168 hours**. For instance, the mean of EMS is significantly higher than others on `mujs` and `objdump`. Although the line coverage of AFL++ grows rapidly in the first 72 hours when fuzzing `pdftotext`, it is eventually surpassed by EMS. Thus, a relatively long time duration to evaluate the coverage performance of a fuzzer on these real world programs is recommended.

D. Evaluation on FuzzBench

To further evaluate fuzzers' performance with the acknowledged benchmark, we utilize *FuzzBench*, one of the most famous standardized benchmarks, to evaluate AFL, MOPT, AFL++, and EMS on `harfbuzz`, `bloaty`, `openssl`, and `zlib`. Each evaluation lasts for 24 hours and is repeated 10

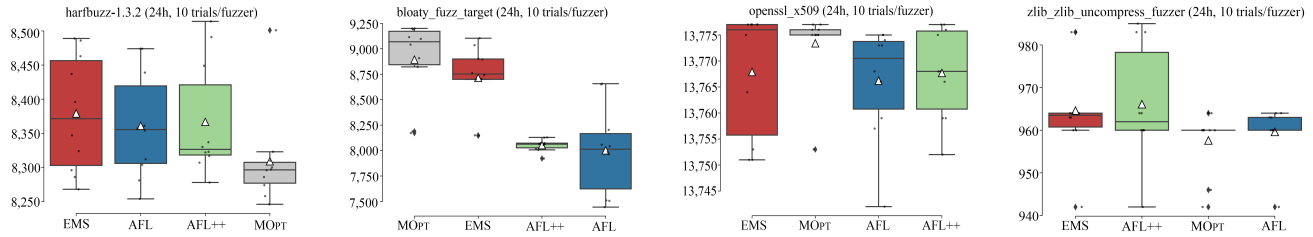

Fig. 10: The boxplot of region coverage generated from 10 trials on FuzzBench, where '△' and '—' represent the mean and median, respectively. The fuzzer with the highest median coverage is on the left. Y-axis: the region coverage found in each trial.

TABLE V: The total number of unique vulnerabilities of each fuzzer reported by ASan, LSan, MSan, and UBSan after deduplication in 16 trials with three different initial seed sets.

		100 seeds							10 seeds							empty seed						
		AFL	QSYM	MOpt	MOpt-dict	EcoFuzz	AFL++	EMS	AFL	QSYM	MOpt	MOpt-dict	EcoFuzz	AFL++	EMS	AFL	QSYM	MOpt	MOpt-dict	EcoFuzz	AFL++	EMS
ASan	pdfimages	2	3	4	5	7	13	**15**	2	2	7	10	11	7	**16**	0	0	0	0	0	0	0
	infotocap	0	0	6	6	3	**7**	**7**	1	0	5	5	4	**6**	**6**	3	1	5	5	4	5	**6**
	w3m	0	0	0	1	2	0	**8**	0	1	4	1	2	0	**10**	0	2	3	2	1	0	**6**
	total	2	3	10	12	12	20	**30**	3	3	16	16	17	13	**32**	3	3	8	7	5	5	**12**
LSan	pdfimages	1	2	6	4	8	**10**	**10**	1	1	**9**	8	6	5	7	0	0	0	0	0	0	0
	infotocap	0	0	2	**3**	1	**3**	2	1	0	2	2	2	**3**	2	1	0	2	2	1	2	2
	w3m	0	0	0	5	1	0	**11**	0	1	4	5	3	0	**8**	0	1	1	3	1	0	**6**
	total	1	2	8	12	10	13	**23**	2	2	15	15	11	8	**17**	1	1	3	5	2	2	**8**
MSan	pdfimages	3	4	6	5	9	**13**	11	3	3	9	**10**	6	9	9	0	0	0	0	0	0	0
	infotocap	0	0	3	4	1	4	**5**	1	0	1	2	1	**5**	3	1	0	2	**3**	1	2	**3**
	w3m	0	1	1	1	1	1	1	0	1	1	1	1	0	1	0	1	1	1	1	0	1
	total	3	5	10	10	11	**18**	17	4	4	11	13	8	**14**	13	1	1	3	**4**	2	2	**4**
UBSan	pdfimages	4	6	8	9	16	**21**	18	4	5	15	16	16	13	**18**	0	0	0	0	0	0	0
	infotocap	0	0	2	2	1	2	2	1	0	2	2	2	2	2	1	0	2	2	1	2	2
	w3m	0	0	0	3	1	0	**11**	0	0	4	5	3	0	**8**	0	0	1	2	1	0	**5**
	total	4	6	10	14	18	23	**31**	5	5	21	23	21	15	**28**	1	0	3	4	2	2	**7**

times to reduce the randomness. The results are shown in Fig. 9 and Fig. 10, from which we have the following conclusions.

• The results in Fig. 9 show the rapid growth of EMS's coverage. For instance, the confidence interval of EMS is above that of other fuzzers at the beginning of the testing on `harfbuzz`. The mean of EMS is higher than AFL++ and AFL on `bloaty` during the fuzzing process. Therefore, with a higher probability EMS can find more coverage faster than other fuzzers.

• EMS achieves better region coverage performance compared to other fuzzers on FuzzBench as shown in Fig. 10. For instance, the mean and median of EMS are higher than other fuzzers on `harfbuzz`. The median of EMS is higher than others on `openssl` and `zlib`. The results demonstrate the significant coverage performance of EMS on a standardized benchmark.

E. Performance Analysis under Different Scenarios

In this subsection, we evaluate the fuzzing performance of different fuzzers on `pdfimages`, `infotocap` and `w3m` under different scenarios. To be specific, we follow the guidance of [38] to further construct new fuzzing experiments with different initial seed sets and different sanitizers.

On one hand, we construct the following two initial seed sets: 1) 10 well-formed seed files for each program, and 2) an empty seed file containing a letter 'a'. To obtain the well-formed seed files, we first collect enough input files with the correct format for each program. Then, we employ afl-cmin [3] to remove the input files with duplicate edge coverage. Finally, we randomly select 10 from the remaining files as the initial seed set of the corresponding program.

On the other hand, we not only use ASan to find unique vulnerabilities as aforementioned, but also employ the following sanitizers to detect different types of vulnerabilities found by each fuzzer.

- LeakSanitizer (LSan): LSan is a run-time memory leak detector to detect specific vulnerability types such as stack overflow, memory leaks and segmentation violations. Without ASan's instrumentation, LSan may find different vulnerabilities compared to ASan's results.
- MemorySanitizer (MSan): MSan is used to detect uninitialized memory reads, e.g., stack- or heap-allocated memory is read before it is written. Note that MSan and ASAN cannot work simultaneously due to their respective instrumentations and mechanisms.
- UndefinedBehaviorSanitizer (UBSan): UBSan is a detector to catch various kinds of undefined behavior vulnerabilities during program execution.

Therefore, combining with the seed file and sanitizer setting in Section V-A, we have three sets of seed files and four kinds of sanitizers. In total, there are 12 different experimental scenarios with different combinations of seed file set and sanitizer. Since we evaluate each fuzzer on four programs, there are 36 results in total for each fuzzer.

Each new experiment still lasts for 168 hours and is repeated 16 times under the same experiment settings in Section V-A. The number of unique vulnerabilities after deduplication in 16 trials under 12 different scenarios is shown in Table V.

1) Performance Analysis with Different Initial Seed Sets: We have the following conclusion according to the vulnerability results with the different initial seed sets in Table V. **EMS finds the most vulnerabilities with different initial seed sets.** For instance, when employing 10 well-formed seed files as the initial seed set, EMS finds the most vulnerabilities on 3 targets reported by ASan, LSan and UBSan. EMS also finds significantly more vulnerabilities than other fuzzers in

TABLE VI: The vulnerability discovery and edge coverage contributed by the *PBOM operator*. P and T represent the mutations provided by the *PBOM operator* and traditional mutation operators shown in Fig. 3, respectively.

pdfimages						
Trial	Unique vulnerabilities found by T	Unique vulnerabilities found by P + T	Contribution	Edge coverage triggered by T	Edge coverage triggered by P + T	Contribution
1	2	3	1	1,825	2,303	+26.2%
2	1	2	1	1,766	2,281	+29.2%
3	2	2	0	1,747	2,234	+27.9%
4	3	3	0	1,659	2,170	+30.8%
5	3	5	2	1,836	2,344	+27.7%
6	2	2	0	1,776	2,289	+28.9%

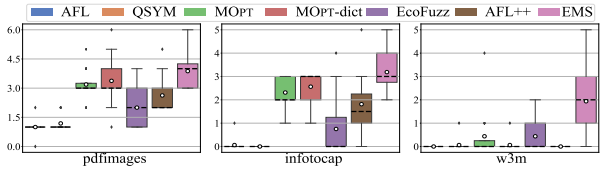

Fig. 11: The boxplot of the number of unique vulnerabilities reported by ASan in 16 trials, when using 10 well-formed seed files as the initial seed set. '○' and '– –' represent the mean and median, respectively. Y-axis: the number of unique vulnerabilities discovered in each trial.

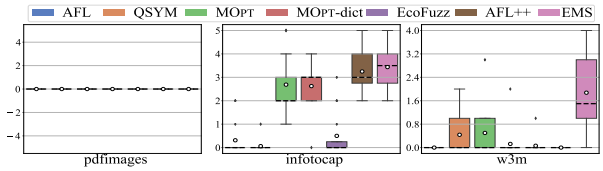

Fig. 12: The boxplot generated by the number of unique vulnerabilities reported by ASan in 16 trials, when using an empty seed as the initial seed set. '○' and '– –' represent the mean and median, respectively. Y-axis: the number of unique vulnerabilities discovered in each trial.

most results when using an empty seed as the initial seed set.

Furthermore, we construct the boxplot to show the distribution of number of unique vulnerabilities reported by ASan in each trial. As shown in Fig. 11, the results demonstrate that EMS outperforms other fuzzers on average when using 10 well-formed files as the initial seed set to fuzz `pdfimages`, `infotocap` and `w3m`. When using empty seed as the initial seed set, the mean and median of EMS are higher than others on `infotocap` and `w3m`, which is shown in Fig. 12.

The aforementioned results demonstrate that **EMS has efficient vulnerability discovery performance with different initial seed sets**.

2) Performance Analysis with Different Sanitizers: The results in Table V also show the following conclusion about the vulnerability discovery reported by different sanitizers. **EMS finds more vulnerabilities reported by different sanitizers than other fuzzers in total.** For instance, EMS finds the most vulnerabilities reported by UBSan on 3 targets with the different initial seed sets. EMS also finds significantly more unique vulnerabilities reported by ASan and LSan than other fuzzers. Only AFL++ finds one more vulnerability reported by MSan than EMS on 3 targets when using 100 seeds and 10 seeds as the initial seed set, respectively.

Overall, EMS achieves the best performance of vulnerability discovery in a total of 12 experimental scenarios, which involve different initial seed sets and different sanitizers. Except for the 11 results that most fuzzers perform the same, EMS finds the most vulnerabilities on 18 of the remaining 25 results. This demonstrates that **EMS outperforms other fuzzers in discovering vulnerabilities under different experimental scenarios.**

VI. Further Analysis

A. *PBOM* Contribution Analysis

Directly comparing the performance of EMS and MOPT can only demonstrate the effectiveness of the *PBOM operator* at a high level. Thus, to perform a fine-grained analysis on the efficacy of the *PBOM operator*, we implement a customized version of EMS, namely EMS-test, to measure the effectiveness of the *PBOM operator* in place. Specifically, EMS-test skips the deterministic stage, and mutates test cases with the traditional and `PBOM` operators in the havoc and splicing stages. When constructing a test case from a seed test case in the queue, EMS-test records the mutations from the traditional mutation operators but ignores the mutations from the *PBOM operator*. Suppose the mutated test case triggers a unique path or crash. In that case, EMS-test additionally constructs a **shadow version** of the mutated test case by replaying the recorded mutations at the same locations of the same seed test case. Thus, the shadow version is obtained without the contribution of the *PBOM operator*. Both the original mutated test case and its shadow version are stored locally to find the unique vulnerabilities reported by ASan and evaluate edge coverage reported by afl-cmin [3]. By comparing the resulting vulnerabilities and edge coverage of the original test cases and the shadow ones, we can show the contribution of the *PBOM operator*.

We employ EMS-test to fuzz `pdfimages`. Each evaluation lasts for 5 hours and is repeated 6 times to reduce the randomness. The experiment settings are the same as in Section V-A. We present the discovered unique vulnerabilities reported by ASan and the edge coverage reported by afl-cmin of each evaluation in Table VI, and have the following conclusions.

• When the mutations of the *PBOM operator* are removed, there are three possible situations about vulnerability discovery as follows: 1) the shadow test case cannot trigger any vulnerability; 2) the shadow test case triggers the same vulnerability found by other shadow ones; and 3) the shadow test case triggers a unique vulnerability that has never been found before. To figure out the contribution of the *PBOM operator*, we utilize ASan to find the unique vulnerabilities triggered by

TABLE VII: The number of the same efficient mutation strategies and the number of different ones collected from two different programs under the different fuzzing durations.

Program A	Duration	N_{n1} (pct.)	N_{n2} (pct.)	N_y (pct.)	N_t	N_{n1} (pct.)	N_{n2} (pct.)	N_y (pct.)	N_t	Program B
pdfimages	5 hours	2,755 (33.0%)	565 (6.8%)	5,020 (60.2%)	8,340	5,971 (37.6%)	4,876 (30.7%)	5,020 (31.6%)	15,867	nasm
	1 day	3,331 (29.4%)	821 (7.3%)	7,168 (63.3%)	11,320	8,021 (36.9%)	6,553 (30.1%)	7,168 (33.0%)	21,742	
	2 days	2,824 (25.7%)	754 (6.9%)	7,400 (67.4%)	10,978	9,388 (38.1%)	7,861 (31.9%)	7,400 (30.0%)	24,649	
	7 days	2,906 (28.5%)	525 (5.1%)	6,775 (66.4%)	10,206	9,098 (39.2%)	7,361 (31.7%)	6,775 (29.2%)	23,234	
objdump	5 hours	3,977 (53.2%)	2,007 (26.9%)	1,487 (19.9%)	7,471	2,446 (50.3%)	925 (19.0%)	1,487 (30.6%)	4,858	infotocap
	1 day	3,941 (50.8%)	1,795 (23.2%)	2,015 (26.0%)	7,751	3,530 (50.9%)	1,394 (20.1%)	2,015 (29.0%)	6,939	
	2 days	3,645 (51.0%)	1,294 (18.1%)	2,210 (30.9%)	7,149	4,878 (53.6%)	2,010 (22.1%)	2,210 (24.3%)	9,098	
	7 days	5,732 (54.5%)	2,049 (19.5%)	2,733 (26.0%)	10,514	5,176 (53.7%)	1,722 (17.9%)	2,733 (28.4%)	9,631	
cflow	5 hours	1,637 (44.8%)	844 (23.1%)	1,174 (32.1%)	3,655	3,566 (37.8%)	4,678 (49.7%)	1,174 (12.5%)	9,418	w3m
	1 day	1,576 (37.9%)	902 (21.7%)	1,676 (40.4%)	4,154	4,337 (33.6%)	6,894 (53.4%)	1,676 (13.0%)	1,2907	
	2 days	1,802 (44.5%)	649 (16.0%)	1,598 (39.5%)	4,049	3,701 (29.5%)	7,226 (57.7%)	1,598 (12.8%)	12,525	
	7 days	1,661 (44.0%)	733 (19.4%)	1,385 (36.6%)	3,779	3,550 (31.4%)	6,367 (56.3%)	1,385 (12.3%)	11,302	

the original test cases and the shadow ones, and analyze the contributions of vulnerability discovery caused by the mutations of the `PBOM` *operator*. The results are shown in Table VI. Interestingly, the original mutated test cases find 1, 1 and 2 more unique vulnerabilities than the shadow ones in 3 trials, respectively. After removing P (the mutations provided by the `PBOM` *operator*) in these four (1 + 1 + 2) original test cases, we get their shadow versions which cannot trigger any of the four vulnerabilities on `pdfimages`. Therefore, the contribution to trigger the four unique vulnerabilities indeed comes from P (the mutations from the `PBOM` *operator*). This demonstrates that **the *`PBOM` operator* can improve the performance of vulnerability discovery**.

• As for the edge coverage, we utilize afl-cmin [3] to collect unique edge coverage of original test cases and shadow ones on `pdfimages`, respectively. The results are shown in Table VI. The edge coverage increases significantly in all 6 trials with the mutations from the `PBOM` *operator*. For instance, the original test cases find 511 more unique edges than the shadow ones in the 4th trial. Therefore, **the *`PBOM` operator* can significantly improve the edge coverage.**

To eliminate the concern that the aforementioned results might be due to that the traditional operators only contribute a small portion of the total number of mutations, we analyze the number of mutations from the `PBOM` *operator* and traditional operators in each fuzzing round, and obtain the following results. 1) Because of the random operator selection in the havoc stage, not all the test cases contain the mutations from the `PBOM` *operator*. Overall 6 trials, there are around 74.83% and 49.22% interesting test cases containing the mutations from the `PBOM` *operator* in the unique crash discovery and unique path discovery, respectively. Thus, many shadow versions of the mutated test cases remain the same. 2) For each mutated test case, the average number of mutations from the `PBOM` *operator* is 2.98, which is significantly smaller than that from the traditional operators (46.11). Therefore, the shadow test cases are obtained by keeping the most mutations after removing the mutations from the `PBOM` *operator*.

In summary, **the *`PBOM` operator* provides the key mutations to find more unique vulnerabilities and edges faster**. This demonstrates the effectiveness of the collected mutation strategies in *inter-* and *intra-*`PBOM`s.

TABLE VIII: The similarities and differences of the efficient mutation strategies between 7 days of execution and other durations of execution on the same program.

Program	Similarities and differences					7 days
	Duration	N_{n1} (pct.)	N_{n2} (pct.)	N_y (pct.)	N_t	N_t
pdfimages	5 hours	1,907 (22.9%)	315 (3.8%)	6,118 (73.4%)	8,340	10,206
	1 day	3,535 (31.2%)	530 (4.7%)	7,255 (64.1%)	11,320	10,206
	2 days	3,092 (28.2%)	636 (5.8%)	7,250 (66.0%)	10,978	10,206
objdump	5 hours	3,034 (40.6%)	531 (7.1%)	3,906 (52.3%)	7,471	10,514
	1 day	2,717 (35.1%)	749 (9.7%)	4,285 (55.3%)	7,751	10,514
	2 days	2,293 (32.1%)	397 (5.6%)	4,459 (62.4%)	7,149	10,514
infotocap	5 hours	1,835 (37.8%)	403 (8.3%)	2,620 (53.9%)	4,858	9,631
	1 day	2,129 (30.7%)	1,060 (15.3%)	3,750 (54.0%)	6,939	9,631
	2 days	3,475 (38.2%)	1,112 (12.2%)	4,511 (49.6%)	9,098	9,631
cflow	5 hours	873 (23.9%)	405 (11.1%)	2,377 (65.0%)	3,655	3,779
	1 day	1,051 (25.3%)	576 (13.9%)	2,527 (60.8%)	4,154	3,779
	2 days	1,178 (29.1%)	289 (7.1%)	2,582 (63.8%)	4,049	3,779
nasm	5 hours	3,711 (23.4%)	839 (5.3%)	11,317 (71.3%)	15,867	23,234
	1 day	6,720 (30.9%)	1,517 (7.0%)	13,505 (62.1%)	21,742	23,234
	2 days	8,491 (34.4%)	1,455 (5.9%)	14,703 (59.6%)	24,649	23,234
w3m	5 hours	2,922 (31.0%)	631 (6.7%)	5,865 (62.3%)	9,418	11,302
	1 day	5,122 (39.7%)	1,021 (7.9%)	6,764 (52.4%)	12,907	11,302
	2 days	4,569 (36.5%)	1,403 (11.2%)	6,553 (52.3%)	12,525	11,302

B. Efficient Mutation Strategy Analysis

We analyze the similarities and differences between the efficient mutation strategies learned on different programs to verify the effect of inter-trial history. Specifically, we spend different time, including 5 hours, 1 day, 2 days and 7 days, conducting fuzzing experiments on six programs (i.e., `pdfimages`, `nasm`, `objdump`, `infotocap`, `cflow`, and `w3m`) to collect their efficient mutation strategies, respectively. Then, we remove the mutation strategies whose frequencies are less than 3, since these strategies are less useful for a program as they rarely trigger unique paths and crashes. Finally, to show the effect of inter-trial history on different programs, we count the number of common mutation strategies found in two different programs and the number of unique strategies in each program. To show the observations more clearly, we define the following parameters.

- N_t: The total number of efficient mutation strategies collected from the current experiment.
- N_{n1}: The number of mutation strategies whose input byte values and length (in, L) appear in both experiments, while their output byte values and mutation types $(out, type)$ only appear in the respective experiment.
- N_{n2}: The number of mutation strategies whose (in, L) are unique in the current experiment and do not appear in the other experiment.
- N_y: The number of mutation strategies in which both

TABLE IX: The average number of executions and executions per second of MOPT and EMS over 8 trials.

Program	MOPT		EMS		
	Average # of executions	Executions per second	Average # of executions	Executions per second	Decrease
cflow	8,593,562.63	198.93	8,246,698.25	190.90	-4.04%
nasm	8,749,619.00	202.54	8,537,833.00	197.64	-2.42%
sassc	6,009,008.00	139.10	5,826,812.00	134.88	-3.03%
w3m	1,033,043.00	23.91	1,032,240.00	23.89	-0.08%
total	24,385,232.63	564.47	23,643,583.25	547.31	-3.04%

(in, L) and the corresponding $(out, type)$ appear in the two experiments.

- (pct.): The percentage of N_{n1}, N_{n2} or N_y in N_t.

The results are shown in Table VII, from which we have the following observations.

• The proportion of the same efficient mutation strategies found from two different programs cannot be negligible. As shown in Table VII, there are more than 60% of efficient mutation strategies of `pdfimages` also triggering unique paths and crashes on `nasm` in all four experiments. Around 35% of `cflow`'s efficient mutation strategies also overlap with the mutation strategies on `w3m`. **This demonstrates the validity of applying *inter-PBOM* learned from one program to other programs.**

• N_{n1} and N_{n2} show the interesting findings as follows. On one hand, the large proportion of N_{n1} and N_y illustrates that EMS triggers a large number of unique paths and crashes on two different programs by mutating the same input byte values, even though the input formats of the two programs and the initial input files used to fuzz them are different. The aforementioned result indicates that when utilizing inter-trial history for fuzzing, prioritizing the mutations on the byte values appeared in the history could also be useful. On the other hand, N_{n2} is the number of unique mutation strategies that only work on a program itself. As shown in Table VII, the N_{n2} of `pdfimages`, `infotocap` and `cflow` is around 6.5%, 19.8% and 20.1%, respectively. These efficient mutation strategies can be useful when fuzzing the same program.

Therefore, we further analyze the similarities and differences of the collected efficient mutation strategies between 7 days of execution and other time of execution on the same program. The results are shown in Table VIII. The observation is as follows. **The same mutation strategies account for the majority between the results of 7 days and other results on the same program.** For instance, N_y is larger than 64.1%, 60.8% and 52.3% on `pdfimages`, `cflow` and `w3m`, respectively. The large proportion of N_{n1} and N_y shown in Table VIII once again indicates that it may improve fuzzing performance by prioritizing the mutations on the byte values appeared in the inter-trial fuzzing history. Furthermore, the small proportion of N_{n2} demonstrates that most mutation strategies in the inter-trial history can be useful on the same program. **These results demonstrate the validity of applying *inter-PBOM* on the same program.**

C. Algorithm Overhead Analysis

In order to evaluate the algorithm overhead of EMS more accurately, we need to control the impact of each test case's mutation time and the impact of the execution speed of each test case on the overall execution speed. Thus, we construct a specific version of MOPT and EMS as follows: *Both* MOPT *and* EMS *only enter the havoc stage, randomly select the mutation operators for a fixed number of times, and mutate the same test case across the entire fuzzing process.* Then, we construct the evaluation of each fuzzer on `cflow`, `nasm`, `sassc` and `w3m`. Each evaluation lasts for 12 hours and is repeated 8 times to reduce the randomness. Finally, we collect the total execution times of each fuzzer to calculate the average execution times over 8 trials. The results are shown in Table IX, from which we learn the following conclusion.

TABLE X: The target programs from the same vendor evaluated in the experiments.

Source	Target	Input format	Test instruction
xpdf-4.02	pdfimages	pdf	@@ /dev/null
	pdftotext	pdf	@@ /dev/null
	pdfinfo	pdf	@@
binutils-2.28	objdump	binary	-S @@
	addr2line	binary	s -e @@
	objcopy	binary	--debugging -p -D @@ /dev/null

The average execution time and executions per second show the low overhead of EMS. For instance, the number of EMS's executions on `cflow` reduces by 4.04% on average over 8 trials. The total executions per second of EMS on 4 targets reduces by 3.04% compared to MOPT, which is caused by the mutation process, the data collection process and the fine-tuning process of `PBOM`s. However, according to the results in Section V-C, although the execution speed of EMS is a bit slower than MOPT, it discovers more unique vulnerabilities and line coverage than MOPT on most programs. This demonstrates that the overhead of EMS is acceptable.

D. Performance of EMS with Different Inter-PBOMs on Different Programs From the Same Vendor

In order to analyze the contribution of the inter-trial history on different programs from the same vendor, we evaluate the fuzzing performance of EMS on target programs from `xpdf-4.02` and `binutils-2.28`, which are the two source codes containing multiple binary programs in our benchmark. The setting of each target program is shown in Table X. For the programs from `xpdf-4.02`, we utilize the inter-trial history collected from `pdfimages` to construct the *inter-PBOM*. On the other hand, we utilize the inter-trial history collected from `objdump` to construct the *inter-PBOM* for the programs from `binutils-2.28`. Furthermore, to evaluate the fuzzing performance of different *inter-PBOM*s, we construct EMS without *inter-PBOM*, or with the *inter-PBOM* trained based on 5 hours, 24 hours and 48 hours of training data, which are represented by EMS_empty, EMS_5h, EMS_24h, and EMS_48h, respectively. Each fuzzer is evaluated 5 times on a target program, each of which lasts for

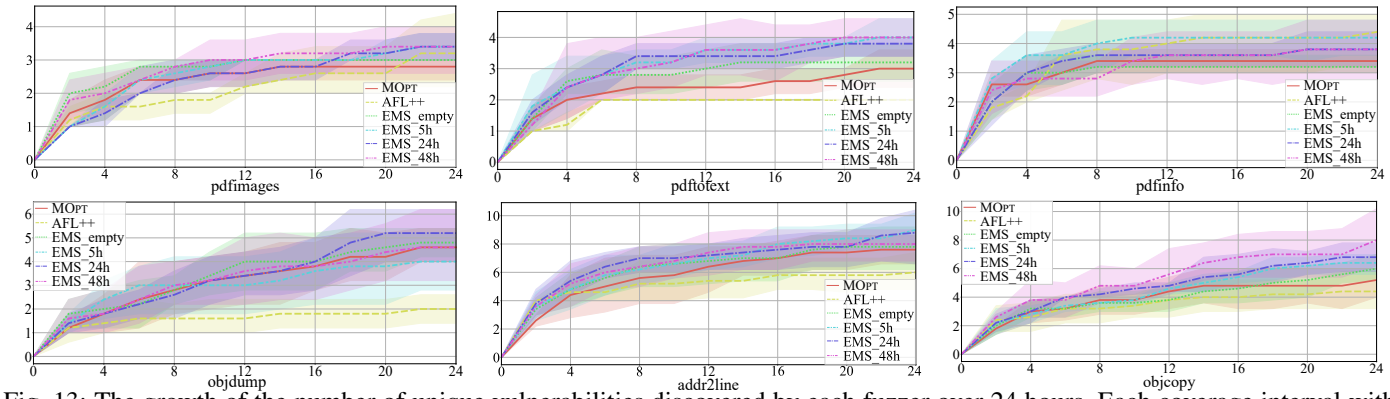

Fig. 13: The growth of the number of unique vulnerabilities discovered by each fuzzer over 24 hours. Each coverage interval with a different color shows the mean and 95% confidence interval for a unique fuzzer. Y-axis: the number of the unique vulnerabilities reported by ASan.

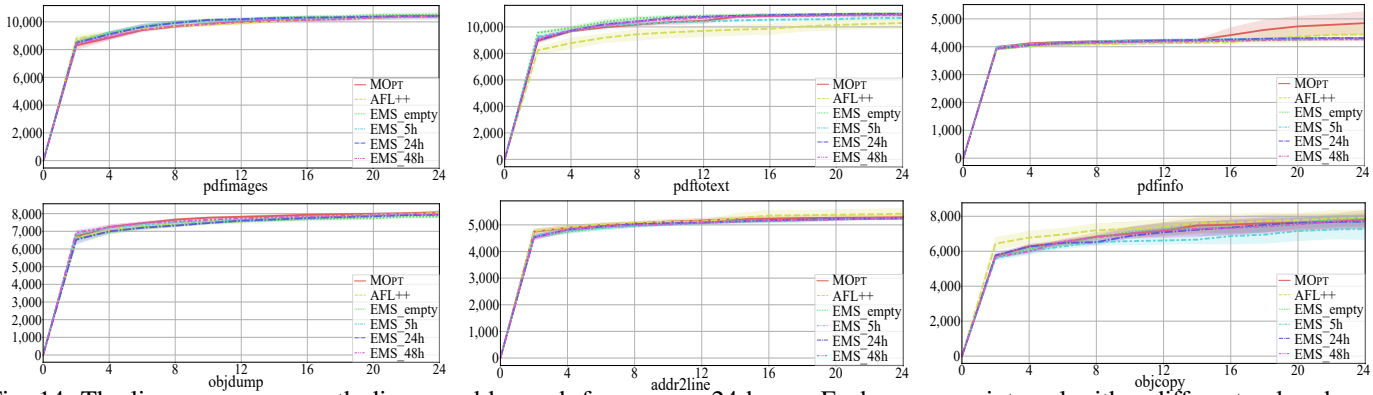

Fig. 14: The line coverage growth discovered by each fuzzer over 24 hours. Each coverage interval with a different color shows the mean and 95% confidence interval for a unique fuzzer. Y-axis: the number of covered code lines.

24 hours. Each MOPT-based fuzzer utilizes '-L 0' as the configuration in this evaluation. The experiment settings are the same as in Section V-A. The results are shown in Fig. 13 and Fig. 14, from which we have the following conclusions.

• EMS with different *inter-*`PBOM`s can find more unique vulnerabilities quickly over 24 hours. For instance, the mean of EMS_48h is higher than other fuzzers on `objcopy` as shown in Fig. 13. The confidence interval of EMS_24h is above others on `objdump`, which represents the higher probability for EMS_24h to discover more unique vulnerabilities. The results demonstrate the contribution of the *inter-*`PBOM` trained from different programs developed by the same vendor.

• Each fuzzer performs closely on the line coverage growth as shown in Fig. 14. For instance, EMS_5h performs well on `pdfimages` while it performs poorly on `objcopy`. AFL++ performs well on `addr2line` while it performs poorly on `pdftotext`. We infer the reason as follows. A 24-hour fuzzing duration is not long enough to differentiate the line coverage performance of the evaluated fuzzers on these real world programs. As shown in Fig. 14, the line coverage of each fuzzer is still growing on most programs after the 24-hour duration. The mutators of each fuzzer still work and can find unique code lines. However, the line coverage results shown in Fig. 8 demonstrate the significant line coverage performance of EMS on `objdump` and `pdftotext` in the 168-hour evaluation. Thus, relatively long time duration to evaluate the coverage performance of a fuzzer on these real world programs is recommended. Note that the different line coverage performance of MOPT-based fuzzers in Appendix VI-D and Section V-C is caused by the different configurations of '-L'. Following the guidance in [7], [44], we use '-L 5' in the 168-hour evaluation and use '-L 0' in the 24-hour evaluation.

• Interestingly, EMS achieves better vulnerability discovery with the *inter-*`PBOM`s collected from the different durations compared to EMS_empty. For instance, the means of the number of unique vulnerabilities discovered by EMS_5h, EMS_24h and EMS_48h are higher than that of EMS_empty on most programs as shown in Fig. 13. The results demonstrate the contribution of the *inter-*`PBOM` trained by the program from the same vendor, which is consistent with the analysis in Section III-A2. From another aspect, EMS_empty finds more unique vulnerabilities than MOPT and AFL++ on most programs as shown in Fig. 13. This simultaneously shows the significant vulnerability discovery performance of EMS without the *inter-*`PBOM`, and demonstrates the contribution of the *intra-*`PBOM`.

VII. Discussion and Limitation

In this section, we discuss several topics related to our design and implementation.

A. Mutation Operator Development

In our implementation of EMS, we utilize the efficient mutation strategies that trigger unique paths and crashes to update the *inter-* and *intra-*`PBOM`s as the mutation operators,

which significantly improve the vulnerability discovery and line coverage. In addition, the efficient mutation strategies can be classified to construct different fine-grained mutation operators. For instance, we can classify the efficient strategies into several types, e.g., the ones that trigger specific vulnerabilities and the ones that trigger rare execution paths, and then construct the corresponding operators. More fine-grained mutation operators can be developed to enhance the utilization of different types of fuzzing history in future work.

B. Mutation Location Selection

In the design of EMS, the fuzzing history is used primarily for extracting efficient mutation strategies to mutate seed test cases. On the other hand, the mutation locations can also be guided by the fuzzing history. In our implementation, we utilize the intra-trial history to probabilistically select the recorded locations where the mutations lead to interesting test cases, i.e., the `favorite_list` described in Section IV. More fine-grained mutation location information may be concluded according to the intra- and inter-trial fuzzing history, e.g., analyzing the impact of mutation locations on some specific branching behaviors according to the past fuzzing results.

C. Application of Machine Learning Algorithms

For a history-driven solution, it is intuitive to consider employing machine learning algorithms to learn from fuzzing history. Indeed, we have tried to implement EMS with the sequence-to-sequence (seq2seq) model. However, although it performs the best on several target programs, the machine learning-based EMS requires more CPU and GPU computational cost, leading to slower execution speed. Thus, the overall fuzzing performance is weakened. Therefore, in this paper we develop EMS based on the lightweight probabilistic model `PBOM`. In summary, machine learning algorithms are not suitable to be employed as the mutation operators, as they will be invoked frequently and thus will slow down the execution speed. On the contrary, for a target program with high latency of executing a test case, e.g., fuzzing a service, machine learning algorithms may have more potential because of their high fitting ability.

VIII. Related Work

A. Mutation-based Fuzzing

AFL is one of the most prevalent mutation-based fuzzers and establishes the framework for fuzzing [3]. In recent years, plenty of works have focused on improving the fuzzing performance. For instance, AFLFast [13] develops a better energy allocation strategy and improves the coverage by prioritizing low-frequency paths. Further, Yue et al. employed a variant of the adversarial multi-armed bandit model to optimize the energy allocation strategy, and presented an adaptive energy-saving fuzzer named *EcoFuzz* [71].

One solution to improve the fuzzing performance is to integrate fuzzing with other technologies, such as taint tracing, concolic execution and gradient descent algorithms. Several works employ dynamic taint analysis, program analysis and symbolic analysis techniques to improve the fuzzing performance [15], [28], [33], [66]. Recently, Yun et al. presented a fast concolic execution engine named *QSYM* and provided a better hybrid fuzzing for AFL [72]. Angora employs the gradient descent algorithm to solve path constraints without concolic execution [17]. *Matryoshka* further solves the deeply nested conditional statements [18]. Zhao et al. presented a novel dispatch strategy to better utilize the concolic execution during the hybrid fuzzing [74]. *FuzzGen* utilizes a whole system analysis to improve the fuzzing performance on the complex libraries [35]. Gan et al. presented a lightweight fuzzing-driven taint technique to guide fuzzing, and provided a data flow sensitive fuzzing solution named *GreyOne* [23].

Coverage-based fuzzing is another solution to improve the fuzzing performance [40], [54]. *T-Fuzz* utilizes a dynamic tracing based technique to remove the checks in a program and improves the code coverage [50]. *CollAFL* mitigates coverage collisions and provides more accurate edge coverage [24]. Recently, Lyu et al. analyzed the fuzzing performance of different mutation operators, and presented MOpt to adaptively optimize the selection probability distribution of operators [44]. Aschermann et al. employed input-to-state correspondence and presented *REDQUEEN* to solve the roadblocks in the targets, such as magic bytes and checksum tests [9]. *ProFuzzer* automatically detects the critical input fields of test cases, and adaptively adjusts the mutation strategy on these fields to trigger vulnerabilities [70].

Different from the papers above, EMS considers reusing the efficient mutation strategies collected from the inter trials, and improves the fuzzing performance with the help of *inter-`PBOM`*. EMS with *inter-`PBOM`* performs significantly better than other fuzzers on our multiple benchmarks, and can be helpful in the fuzzing scenarios like parallel fuzzing and continuous fuzzing. Furthermore, one of the goals in this paper is to trigger similar branching behavior in different execution paths, which can be caused by the same immediate operands used in different instructions and the shared basic blocks in the paths. To achieve this, EMS reuses efficient mutation strategies, which include interesting mutation values and different mutation types, at different locations of seed files. As a result, EMS can serve as a new mutator in most mutation-based fuzzers.

B. Generation-based Fuzzing

Most of the generation-based fuzzers utilize data-driven techniques to generate the test cases with specific input formats [20], [25], [31], [45]. *Skyfire* utilizes probabilistic context-sensitive grammar to learn syntax and semantic rules, and generates the test cases with the formats of XML and XSL [64]. Han et al. presented a novel generation algorithm named *semantics-aware assembly*, which can generate the test cases with semantical and syntactical correctness [30]. *Nautilus* combines the use of grammars with code coverage feedback, which generates the test cases with a higher probability to have semantical and syntactical correctness [8]. Lee et al. presented a neural network language model-guided fuzzer named *Montage* to find JavaScript engine vulnerabilities [39].

C. Machine Learning-based Fuzzing

In recent years, multiple works have focused on improving fuzzing performance with machine learning techniques [10],

[14], [26], [43], [48], [59], [61]. *Augmented-AFL* utilizes neural networks to predict the locations in a test case that can trigger unique paths and crashes with a higher probability [53]. *NEUZZ* employs a neural network model to learn the real-word program's branching behaviors, and then utilizes the program smoothing technique to locate the bytes in a test case that influence the branching behaviors [60].

D. Other Fuzzing Strategies

Many works with diverse motivations are proposed recently, including the ones that provide better initial seed sets and provide more comprehensive metrics for evaluating fuzzing performance [38], [55], [67]; the ones that present effective kernel fuzzers [19], [29], [36], [37], [49], [62], [68]; the ones that provide directed fuzzing at specific locations [12], [16], [75]; the ones that protect binaries from fuzzing attacks and improve hypervisor fuzzers [27], [57], [58]; and the ones that focus on algorithmic complexity vulnerabilities and improve the fuzzing execution speed [11], [42], [47], [52], [69].

IX. Conclusion

In this paper, we discover that both intra- and inter-trial fuzzing history contain rich knowledge of key mutation strategies that lead to the discovery of unique paths or crashes. These mutation strategies implicitly carry partial path constraint solutions and can be used to accelerate the discovery of new paths or crashes sharing similar partial path constraints. Motivated by this insight, we propose `PBOM`, a lightweight and efficient model, to capture the mutation strategies that trigger unique paths and crashes from the intra- and inter-trial history. We present a novel history-driven mutation framework EMS that employs `PBOM` as one of the mutation operators to probabilistically provide the desired mutation byte values and mutation types according to the input ones. We evaluate EMS against AFL, QSYM, MOpt, MOpt-dict, EcoFuzz, and AFL++ on 9 real world programs. The results show that EMS discovers more unique vulnerabilities and has higher line coverage than other fuzzers on most of the programs. EMS also achieves superior coverage performance on the standardized benchmark FuzzBench, and performs the best on discovering different types of vulnerabilities with different initial seed sets. In addition, we conduct further analysis to demonstrate the validity and low overhead of EMS. The performance of EMS with different *inter-`PBOM`*s demonstrates the contribution of the inter-trial fuzzing history on different programs from the same vendor. Overall, EMS can serve as a new direction to improve the coverage and vulnerability discovery of 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, 62102360, and U1836202, 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] "AddressSanitizer," http://clang.llvm.org/docs/AddressSanitizer.html.

[2] "afl-cov," https://github.com/mrash/afl-cov, 2020.

[3] "American Fuzzy Lop," https://github.com/google/AFL, 2020.

[4] "BusyBox," https://busybox.net/, 2020.

[5] "Fuzzbench - fuzzer benchmarking as a service." https://github.com/google/fuzzbench, 2021.

[6] "NVD, CVE: Common Vulnerabilities and Exposures," https://cve.mitre.org, 2021.

[7] "The open source link of mopt," https://github.com/puppet-meteor/MOpt-AFL, 2021.

[8] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars." in *NDSS*, 2019.

[9] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[10] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 95–110, 2017.

[11] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," in *NDSS*, 2020.

[12] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[13] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016.

[14] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops*. IEEE, 2018, pp. 116–122.

[15] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *S&P*. IEEE, 2015.

[16] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *CCS*, 2018, pp. 2095–2108.

[17] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *S&P*. IEEE, 2018.

[18] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *CCS*, 2019, pp. 499–513.

[19] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: interface aware fuzzing for kernel drivers," in *CCS*, 2017.

[20] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014.

[21] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *S&P*. IEEE, 2019, pp. 472–489.

[22] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies*, 2020.

[23] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing," in *USENIX Security*, 2020.

[24] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *S&P*. IEEE, 2018.

[25] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, 2008.

[26] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[27] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "Antifuzz: Impeding fuzzing audits of binary executables," in *USENIX Security*, 2019, pp. 1931–1947.

[28] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations." in *USENIX Security*, 2013.

[29] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *CCS*, 2017.

[30] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *NDSS*, 2019.

[31] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *USENIX Security*, 2012.

[32] H. Huang, A. M. Youssef, and M. Debbabi, "Binsequence: fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 155–166.

[33] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *S&P*. IEEE, 2020, pp. 1613–1627.

[34] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 387–391.

[35] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *USENIX Security*, 2020.

[36] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *S&P*. IEEE, 2019, pp. 754–768.

[37] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *NDSS*, 2020.

[38] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *CCS*, 2018.

[39] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language model-guided javascript engine fuzzer," in *USENIX Security*, 2020.

[40] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[41] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *USENIX Security*, 2021.

[42] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 809–814.

[43] C. Lyu, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

[44] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "Mopt: Optimized mutation scheduling for fuzzers," in *USENIX Security*, 2019, pp. 1949–1966.

[45] R. Ma, S. Ren, K. Ma, C. Hu, and J. Xue, "Semi-valid fuzz testing case generation for stateful network protocol," *Tsinghua Science and Technology*, vol. 22, no. 5, pp. 458–468, 2017.

[46] A. Mockus, "Large-scale code reuse in open source software," in *First International Workshop on Emerging Trends in FLOSS Research and Development*. IEEE, 2007, pp. 7–7.

[47] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *S&P*. IEEE, 2019, pp. 787–802.

[48] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *arXiv preprint arXiv:1711.02807*, 2017.

[49] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *USENIX Security*, 2018, pp. 729–743.

[50] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *S&P*. IEEE, 2018.

[51] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *S&P*. IEEE, 2017.

[52] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *CCS*, 2017.

[53] M. Rajpal, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," *arXiv preprint arXiv:1711.04596*, 2017.

[54] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, 2017.

[55] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing." in *USENIX Security*, 2014.

[56] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[57] S. Schumilo, C. Aschermann, A. Abbasi, S. Wör-ner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *USENIX Security*, 2021.

[58] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Hyper-cube: High-dimensional hypervisor fuzzing," in *NDSS*, 2020.

[59] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "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*, 2020, pp. 737–749.

[60] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *S&P*. IEEE, 2019, pp. 803–817.

[61] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *S&P*. IEEE, 2017, pp. 521–538.

[62] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *NDSS*, 2019.

[63] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, 2016.

[64] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *S&P*. IEEE, 2017.

[65] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, and A. X. Liu, "Mpinspector: a systematic and automatic approach for evaluating the security of iot messaging protocols," in *USENIX Security*, 2021, pp. 4205–4222.

[66] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *S&P*. IEEE, 2010.

[67] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS*, 2013.

[68] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *S&P*. IEEE, 2018, pp. 661–678.

[69] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *CCS*, 2017.

[70] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *S&P*. IEEE, 2019, pp. 769–786.

[71] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *USENIX Security*, 2020.

[72] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX Security*, 2018.

[73] B. Zhao, S. Ji, W.-H. Lee, C. Lin, H. Weng, J. Wu, P. Zhou, L. Fang, and R. Beyah, "A large-scale empirical study on thevulnerability of deployed iot devices," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[74] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." in *NDSS*, 2019.

[75] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *USENIX Security*, 2020.