One Ring to Rule Them All: Do We Have a Ring in Binary Fuzzing?

Anonymous Author(s)

ABSTRACT

Coverage-guided fuzzing has become mainstream in fuzzing to automatically expose program vulnerabilities. Recently, a group of fuzzers are proposed to adopt a random search mechanism namely *Havoc*, explicitly or implicitly, to augment their edge coverage effectiveness. However, they only tend to adopt *Havoc* by its default setup as an implementation practice while none of them attempt to explore its power by more setups or inspect its rationale for potential improvement. In this paper, to address such issues, we conduct the first empirical study for *Havoc* to enhance the understanding of its characteristics. To be specific, we first find that applying *Havoc* by its default setup to fuzzers can significantly improve their edge coverage performance. Interestingly, we can further find even simply executing *Havoc* itself without appending it to any fuzzer can incur a rather strong edge coverage performance and outperform most of our studied fuzzers in the paper. Moreover, we also attempt to extend the execution time of *Havoc* and find that most fuzzers can not only significantly enhance their edge coverage performance, but also strongly bridge their performance gap. Inspired by the findings, we further propose $Havoc_{DMA}$, which models the *Havoc* mutation strategy as a multi-armed bandit problem to be solved by dynamically adjusting the mutation strategy. The evaluation result presents that $Havoc_{DMA}$ can enhance the edge coverage significantly by 11% on average of all the benchmark projects compared with *Havoc* and even slightly outperform QSYM which adopts three threads for augmenting its computing resource. Furthermore, executing $Havoc_{DMA}$ with three threads in parallel can result in 9.04% more edge coverage averagely over QSYM upon all the benchmark projects.

1 INTRODUCTION

Fuzzing refers to an automated software testing technique that inputs invalid, unexpected, or random data to programs for exposing their exceptions such as crashes, failing assertions, or memory leaks which can be further used for deriving their vulnerabilities/bugs [48]. In particular, many existing fuzzers tend to facilitate their vulnerability/bug exposure via optimizing code coverage of programs [52][42][41][7][24]. Given an initial collection of seeds (e.g., class files), such coverage-guided fuzzers usually develop strategies to iteratively mutate them for generating new seeds which can advance code coverage enhancement.

Notably, a group of recent coverage-guided fuzzers, e.g., AFL [52], AFL++ [16], MOPT [30], QSYM [51], FairFuzz [24], integrate a lightweight random search mechanism namely *Havoc*[1] to their respective fuzzing strategies for augmenting their code coverage exploration. For instance, we observe that while the major fuzzing strategy of FairFuzz can explore 12k+ program edges within around 21 hours, its adopted *Havoc* can further explore 7.8k+ program edges within only around 3 hours. In general, *Havoc* iteratively generates new seeds via randomly mutating the seeds collected after executing its fuzzer. In contrast to many existing fuzzers which adopt only one mutator under each iterative execution, usually *Havoc* randomly selects multiple diverse mutators, e.g., flipping a single bit and inserting/deleting a randomly-chosen continuous chuck of codes, and applies them altogether for generating one seed during each iterative execution. Typically, *Havoc* can be integrated with fuzzers either sequentially, i.e., executing *Havoc* upon the seeds collected after executing their major fuzzing strategies, or in parallel, i.e., executing *Havoc* and their major fuzzing strategies at the same time in different processes/threads upon their seed aggregation.

Although *Havoc* has been widely used by such fuzzers, they tend to include *Havoc* as only implementation improvement without further investigating its rationale, inspecting its performance boundaries, or exploring its potentials. For instance, AFL, AFL++ and FairFuzz simply adopt *Havoc* as an additional mutation stage and QSYM utilizes *Havoc* to generate seeds for its concolic execution to explore code coverage. However, they simply adopt *Havoc* under its default setup, i.e., none of them attempt to investigate whether exploring the potential power of *Havoc* can advance more on their efficacy by exploring its optimal settings, enhancing its integration mode with fuzzers, analyzing its mechanism, etc.

In this paper, we conduct the first comprehensive study of *Havoc* to enhance the understanding of its characteristics. In particular, we first collect 7 recent binary fuzzers and the pure *Havoc* (i.e., applying *Havoc* only without appending it to any fuzzer) as our studied subjects and construct a benchmark by collecting their studied projects in common. Then, we conduct an extensive study to investigate how enabling *Havoc* in the studied subjects can impact their performance (e.g., code coverage and bug exposure). Our evaluation results indicate that for all the studied fuzzers, appending *Havoc* to them under its default setup can significantly enhance their edge coverage upon all the benchmark projects from 44% to 3.7X on average. Meanwhile, we find that even directly applying the pure *Havoc* only can result in a surprisingly strong edge coverage which can significantly outperform most of our studied fuzzers. Moreover, while different fuzzers can incur quite divergent edge coverage results, applying *Havoc* to the studied fuzzers under sufficient execution time can in general not only significantly improve their edge coverage compared with their default *Havoc* integration, but also strongly reduce the performance gap of their edge coverage when applying their original versions. At last, all the studied fuzzers can successfully expose 243 unique crashes by integrating *Havoc* while they can only expose 13 by themselves onlyJiahong:[double only].

Inspired by our findings, we propose an improved technique for *Havoc* namely $Havoc_{DMA}$ which models the *Havoc* mutation

[1]While such mechanism is respectively named by different techniques, we adopt *Havoc* following AFL and FairFuzz.

strategy as a multi-armed bandit problem to be further solved by dynamically adjusting the mutation strategy. The evaluation results indicate that under 24-hour execution, $Havoc_{DMA}$ can outperform the pure *Havoc* significantly by 11% in terms of edge coverage on all the benchmarks on average. $Havoc_{DMA}$ can also slightly outperform QSYM which adopt much more computing resource, i.e., three threads in parallel. Moreover, we design $ParallelHavoc_{DMA}$ by executing $Havoc_{DMA}$ with three threads in parallel. The evaluation result indicates that $ParallelHavoc_{DMA}$ can outperform QSYM by 9.04% on average.

To summarize, this paper makes the following contributions:

- We extensively study the performance impact by applying *Havoc* to a set of studied fuzzers on real-world benchmarks.
- We find that applying *Havoc* can significantly enhance program edge coverage compared with the studied fuzzers and be effective to expose program crashes.
- We propose a lightweight approach $Havoc_{DMA}$ based on our findings which can enhance the edge coverage by 11% compared with the pure *Havoc* under 24-hour execution.

2 BACKGROUND

Havoc was first proposed by AFL [52] and further adopted by many fuzzers [24][7][6][16][30]. While their adoptions of *Havoc* can be slightly different, they typically integrate *Havoc* with their major fuzzing strategies for their iterative executions, i.e., under each iteration, *Havoc* repeatedly mutates each seed provided by (or aggregated to its own seed collection from) executing its integrated fuzzing strategy via applying multiple randomly selected mutators simultaneously. Figure 1 presents the basic workflow of *Havoc*. For each seed in the seed corpus, *Havoc* first determines the count of its mutations based on the real-time seed information, e.g., queuing time of seeds and the existing "interesting" seed number (i.e., the number of the seeds which can explore new edges defined by AFL). Next, each time when mutating such seed, *Havoc* implements mutator stacking, i.e., mutating it by randomly applying multiple mutators (e.g., 15 for AFL, MOPT, etc.) from a set of mutators at the same time. Note that *Havoc* usually enables a maximum size of such mutator stack (e.g., 128 for AFL, MOPT, etc.) and one mutator can thus be selected multiple times when mutating a given seed. If the generated mutant is "interesting" (i.e., exploring new edges), it will be included as a seed for further mutations. *Havoc* repeats such process until hitting the mutation count when it is terminated. Accordingly, its fuzzer can resume the execution of its major fuzzing strategy when needed.

2.1 Mutators and Mutator Stacking

Table 1 presents the details of *Havoc* mutators. Note that in the "mutator" column, the number followed by the mutator name refers to the bit-wise mutation range. For instance, `bitflip 1` refers to flipping one random bit at a random position. To our best knowledge, most fuzzers [52][16][7][6][51][30] enable a total of 15 mutators for *Havoc*. In this paper, we categorize them into two dimensions: *unit mutators* (labeled in red in Table 1) and *chunk mutators* (labeled in blue). In general, *unit mutators* refer to mutating the units of data storage in programs, e.g., bit/byte/word. For example, applying the `bitflip` mutator in Table 1 can flip a bit, i.e., switching between 0 and 1. Meanwhile, *chunk mutators* tend to mutate a seed in terms of its randomly chosen chunk. For instance, the `delete bytes` mutator in Table 1 first randomly selects a chunk of bytes in the seed and then deletes them altogether.

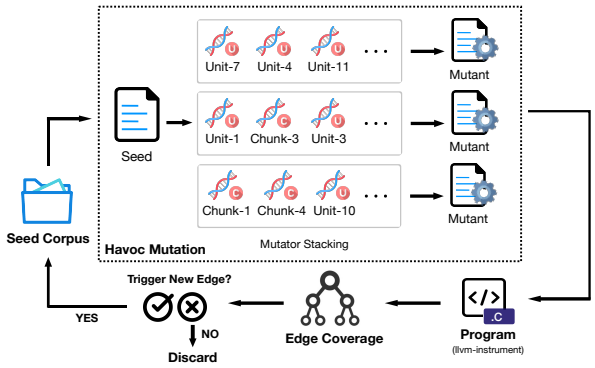

Figure 1: The Framework of *Havoc*

Table 1: Mutation operators defined by *Havoc*

Type	Meaning	Mutator
bitflip	Flip a bit at the random position.	`bitflip 1`

interesting values	Set bytes with hard-coded interesting values.	`interest 8`
		`interest 16`
		`interest 32`

arithmetic increase	Perform addition operations.	`addition 8`
		`addition 16`
		`addition 32`

arithmetic decrease	Perform subtraction operations.	`decrease 8`
		`decrease 16`
		`decrease 32`

random value	Randomly set a byte to a random value.	`random byte`
delete bytes	Randomly delete consecutive bytes.	`delete chunk bytes`
clone/insert bytes	Clone bytes in 75%, otherwise insert a block of constant bytes.	`clone/insert chunk bytes`
overwrite bytes	Randomly overwrite the selected consecutive bytes.	`overwrite chunk bytes`

While many fuzzers [29][42][41][37][19] mainly apply one mutator to a seed each time, *Havoc* enables mutator stacking to stack and apply multiple mutators altogether on a seed to generate one mutant each time. Typically, *Havoc* first defines a *stacking size* for the applied mutators which is usually randomly determined out of the power of two till 128, i.e., 2, 4, 8,...128, for each mutation. Accordingly, *Havoc* can randomly select mutators into the stack where one mutator can be possibly selected multiple times. Eventually, all the stacked mutators are applied to the seed in order to generate a mutant. Note that while most of fu zzers uniformly select mutators for their *Havoc*, MOPT and AFL++ adopt a probability distribution generated by Particle Swarm Optimization [22] for *Havoc* to select mutators.

2.2 Integration

Havoc can be typically integrated with fuzzers in two manners. One is the sequential manner, i.e., appending *Havoc* as a later mutation stage to their major fuzzing strategies. For instance, AFL [52]

launches *Havoc* upon the seeds generated after applying its deterministic mutation strategy to generate more seeds. The other is the parallel manner, i.e., applying *Havoc* and the major mutation strategy of a fuzzer in parallel. For instance, QSYM [51] enables three processes which execute *Havoc*, AFL deterministic mutation strategy, and concolic execution [18] respectively, where the first two processes are independently executed in parallel and their respective generated seeds are continuously aggregated to be used for the concolic execution.

While *Havoc* has been widely adopted by the aforementioned fuzzers, it is simply utlized as an implementation practice while none of the fuzzers has explicitly explored its potential power, e.g., assessing its mechanism and adjusting its setup. Therefore, our paper attempts to explicitly investigate *Havoc*, i.e., extensively assessing its performance impact to fuzzers and its mechanisms, for better leveraging its power and providing practical guidelines for future research.

3 *HAVOC* IMPACT STUDY

3.1 Subjects & Benchmarks

3.1.1 Subjects. In general, we determine to adopt the following types of fuzzers as our study subjects. First, we attempt to include the fuzzers which originally adopt *Havoc* to expose how *Havoc* can impact their performance by default. Next, we also attempt to explore the fuzzers which do not originally adopt *Havoc* but can possibly append *Havoc* under appropriate effort. Accordingly, we can investigate whether and how *Havoc* can be effective in a wider range of regular fuzzers. At last, we also include the pure *Havoc*, i.e., using only one seed to launch *Havoc* for generating new seeds without appending it to any fuzzer, for analyzing how the power of *Havoc* can be unleashed.

Note that while there can be many existing fuzzers which can meet our selection criteria above, we also need to filter them because we wish to select the representative ones to both control our study loads and strengthen our finding efficacy. To this end, we first determine to limit our search scope within the fuzzers published in the top software engineering and security conferences, i.e., ICSE, FSE, ASE, ISSTA, CCS, S&P, USENIX Security, and NDSS, of recent years. Furthermore, we can only evaluate the fuzzers when their source code are fully available and can be successfully executed. At last, it is rather challenging to integrate *Havoc* with certain potential fuzzers due to their engineering-/concept-wise difficulties. Therefore in this paper, we only target AFL variants due to the appropriate workloads for implementing *Havoc* for them.

Eventually, we select 8 representative fuzzers as our studied subjects, including 5 *Havoc*-inclusive fuzzers (AFL, AFL++, MOPT, FairFuzz, QSYM), 2 *Havoc*-exclusive fuzzers (Neuzz [42], MTFuzz [41]) and the pure *Havoc* itself. Note that such subjects can be rather representative in terms of technical styles, i.e., including AFL-based, Concolic-execution-based, and neural program-smoothing-based fuzzers.

3.1.2 Benchmark programs. We construct our benchmark based on the projects commonly adopted by the original papers of our studied fuzzers [24][30][51][42][41]. In particular, we select the 12 frequently used projects out of the papers to form our benchmark for evaluation (present in [39]). Table 2 presents the statistics of our adopted benchmarks. Specifically, we consider our benchmark to be sufficient and representative due to following reasons:

Table 2: Statistics of the studied benchmarks

Programs			LOC
Package	Target	Class	
binutils-2.36	readelf	ELF	72,164
	nm	ELF	55,307
	objdump	ELF	74,532
	size	ELF	54,429
	strip	ELF	65,432
libjpeg-9c	djpeg	JPEG	9,023
tcpdump-4.99.0	tcpdump	PCAP	46,892
libxml2-2.9.12	xmllint	XML	73,320
libtiff-4.2.0	tiff2bw	TIFF	15,024
mupdf-1.18.0	mutool	PDF	123,575
harfbuzz-2.8.0	harfbuzz	TTF	9,847
jhead-3.04	jhead	JPEG	1,885

(1) these 12 benchmark projects cover 7 different kinds of file formats of seed input, e.g., `ELF`, `JPEG`, `TIFF`;
(2) the LoC of these programs which ranges from 1,885 to over 120K can represent a wide range of program size;
(3) they cover diverse functions including development tools (e.g., readelf, objdump), code processing tools (e.g., tiff2bw), graphics processing tools (e.g., djpeg), network analysis tool (e.g., tcpdump), etc.

3.2 Evaluation Setups

Our evaluations are performed on ESC servers with 128-core 2.6 GHz AMD EPYC™ ROME 7H12 CPUs and 256 GiB RAM. The servers run on Linux 4.15.0-147-generic Ubuntu 18.04. The evaluations that involve deep learning model training (i.e., Neuzz and MTFuzz) are executed with four RTX 2080ti cards.

We strictly follow the respective original procedures of the studied fuzzers to execute them. Specifically, we set the overall execution time budget for each fuzzer 24 hours following prior works [23][6][42][7][24][41]. Note that we run each experiment five times for obtaining the average results to reduce the impact of randomness. Notably, all the studied fuzzers are executed with the programs based on AFL instrumentation to collect the runtime coverage information. To this end, we apply the AFL (v2.57) llvm-mode (llvm-8.0) to instrument the program source code during compilation.

We adopt the edge coverage to reflect code coverage where an edge refers to a transition between program blocks, e.g., a conditional jump. We then measure it via the edge number derived by the AFL built-in tool named `afl-showmap`, which has been widely used by many existing fuzzers [24][51][41][42][9].

3.3 Research Questions

We investigate the following research questions for extensively studying *Havoc*.

- **RQ1**: How does the default *Havoc*, i.e., direct application of *Havoc* without modifying its setup or mechanism, perform on different fuzzers? For this RQ, we attempt to investigate

the performance impact by integrating the default *Havoc* with the studied fuzzers.

- **RQ2**: How does *Havoc* perform on different fuzzers under diverse setups? For this RQ, we investigate the performance impact of *Havoc* by enabling *Havoc* in the studied subjects under different setups of the execution time for *Havoc*.

3.4 Result and Analysis

3.4.1 RQ1: performance impact of the default Havoc. We first investigate the impact of the default *Havoc* on the *Havoc*-inclusive fuzzers. As mentioned in Section 2.2, there can be typically two types of the default setting for integrating *Havoc* to fuzzers. For many fuzzers which append *Havoc* as a later fuzzing strategy to their major fuzzing strategies under each iterative execution, *Havoc* is launched upon the termination of their major fuzzing strategies and terminated after hitting the mutation count determined at runtime (illustrated in Section 2) without any specific execution time control by default. As a result, we can infer that the execution time of the default *Havoc* cannot be deterministic. On the other hand, for the fuzzers which execute *Havoc* and their major fuzzing strategies in parallel, the default *Havoc* is usually executed all along under the execution time. Therefore, its execution time can be typically equal to the overall execution time. Figure 2 presents the execution time distribution of all the studied fuzzers under the total execution time 24 hours. We can observe that while AFL, AFL++, and FairFuzz incur quite limited total execution time of their adopted *Havoc* by default, i.e., from 0.79 hour to 3.09 hours, MOPT and QSYM incur quite large execution time for *Havoc*. Note that the default *Havoc* in QSYM is executed for the whole 24 hours as mentioned in Section 2.2.

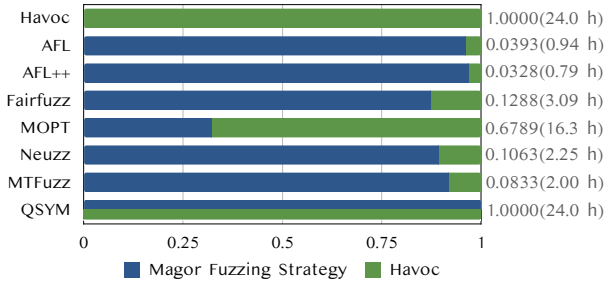

Figure 2: execution time distribution within 24 hours

Table 3 presents the edge coverage results of the five *Havoc*-inclusive fuzzers in terms of their major fuzzing strategies (represented as "Major") and the default *Havoc* (represented as "Havoc") respectively. Note that an edge can be explored multiple times by different fuzzing strategies. Therefore, we only count an edge into the edge coverage result of a fuzzing strategy if such strategy first explores the edge, i.e., the alternative fuzzing strategy (ies) cannot count such edge into its edge coverage result even if they explore it later. Generally, we can observe that for all the studied fuzzers, the default *Havoc* can significantly enhance the edge coverage over their major fuzzing strategies on all the benchmark projects averagely, i.e., 47.2% in AFL, 43.9% in AFL++, 78.2% in FairFuzz, 3.7X in MOPT, 1.5X in QSYM. Moreover, for each benchmark project, we can further observe that the edge coverage gain cannot be negligible. For instance, for AFL, its project-wise edge coverage enhancement of the default *Havoc* over its major fuzzing strategy ranges from 15.2% to 3.5X.

We also attempt to append the default *Havoc* into the originally *Havoc*-exclusive fuzzers, i.e., Neuzz and MTFuzz, and further investigate how the default *Havoc* can impact their edge coverage performance. Specifically, their integration follows the sequential pattern adopted by many existing fuzzers mentioned in Section 2.2, i.e., appending *Havoc* after executing the original mutation strategies of Neuzz and MTFuzz under each iterative execution. Therefore, we can derive that the execution time of the default *Havoc* adopted by them cannot be deterministic. In particular, their execution time distribution can also be presented in Figure 2 where Neuzz incurs 2.25 hours and MTFuzz incurs 2 hours for executing the default *Havoc*.

Table 4 presents the edge coverage results where "Origin" refers to the original versions of Neuzz and MTFuzz and "Integration" refers to Neuzz and MTFuzz integrating *Havoc* with their major fuzzing strategies. We can observe that overall, *Havoc* can significantly enhance the edge coverage by 78.9% on average compared with the major fuzzing strategy of Neuzz and by 66.0% on average compared with the major fuzzing strategy of MTFuzz. Moreover, the integrated fuzzers can enable rather significant performance gain, i.e., 19.3% over the original Neuzz and 26.6% over the original MTFuzz. To summarize, we can derive that for both *Havoc*-inclusive and -exclusive fuzzers, appending the default *Havoc* to them can significantly enhance their major/original fuzzing strategies.

Finding 1:

Applying Havoc by default can significantly improve the edge coverage performance of a fuzzer.

Interestingly, we can find from Table 4 that the pure *Havoc*, i.e., using only one seed to launch *Havoc* and execute it all along without appending it to any fuzzer, can incur a rather strong edge coverage performance, i.e., 31K+ edges on average on all the benchmark projects. More specifically, the pure *Havoc* can significantly outperform most of the studied fuzzers, e.g., 177% over AFL, 257% over AFL++, 45% over Neuzz, while obtaining close performance with MOPT and QSYM. Note that while we can definitely enable multiple ways, e.g., apply more than one seed, to launch the execution of the pure *Havoc*, the fact that using one seed can already achieve such superior performance can be a strong evidence that *Havoc* itself can be a powerful fuzzer.

Finding 2:

Havoc is essentially a powerful fuzzer—executing Havoc under one seed without being appended to any fuzzer for sufficient time can incur a superior edge coverage performance over many existing fuzzers.

We then investigate the correlation between the edge coverage performance and the execution time of *Havoc*. We can observe that while MTFuzz, QSYM, and the pure *Havoc* can incur quite stronger edge coverage over the other fuzzers according to Tables 3 and 4,

Table 3: The performance impact of the default *Havoc* on different fuzzers

Programs	AFL		AFL++		FairFuzz		MOPT		QSYM	
	Havoc	Major	Havoc	Major	Havoc	Major	Havoc	Major	Havoc	Major
readelf	6,496	5,616	3,028	6,816	11,598	24,774	59,510	7,995	37,020	32,577
nm	979	6,209	1,468	4,581	5,724	10,732	17,754	5,405	18,976	11,383
objdump	4,757	8,991	2,043	11,430	8,195	16,009	26,102	11,925	21,163	19,934
size	1,146	7,116	1,339	2,866	7,706	8,841	15,470	3,702	13,128	10,572
strip	2,456	12,854	2,198	9,918	15,676	10,946	35,201	2,117	29,461	17,172
djpeg	862	4,526	792	4,682	2,284	8,121	11,900	3,905	12,600	6,695
tcpdump	1,978	12,994	842	4,452	9,925	8,532	23,910	20,365	22,271	23,533
xmllint	7,253	10,936	4,641	11,345	10,874	17,300	35,862	13,756	34,442	12,096
tiff2bw	778	4,594	978	2,789	2,257	6,577	4,627	4,080	3,759	5,542
mutool	8,957	2,572	8,452	2,576	8,022	6,408	11,542	5,896	12,879	4,948
harfbuzz	6,774	14,939	5,800	10,611	18,624	11,315	53,107	1,071	42,629	16,641
jhead	766	246	358	699	693	421	909	218	2,478	2,038
Average	3,600	7,633	2,662	6,064	8,465	10,831	24,658	6,703	20,900	13,594

Table 4: The impact of *Havoc* in Neuzz and MTFuzz

Programs	Pure Havoc	Neuzz			MTFuzz		
		Origin	Integration		Origin	Integration	
			Havoc	Major		Havoc	Major
readelf	73,478	43,040	18,530	27,699	40,594	13,967	31,220
nm	20,696	16,002	8,617	12,469	20,863	9,841	12,745
objdump	37,401	29,155	14,619	18,661	25,369	12,057	18,784
size	17,634	13,228	6,191	8,040	12,256	8,593	6,686
strip	38,200	29,767	16,117	18,959	28,981	14,098	22,884
djpeg	16,142	15,805	12,861	8,549	7,640	7,432	5,142
tcpdump	43,482	17,216	20,704	5,755	14,067	19,237	9,727
xmllint	49,269	28,213	15,891	21,386	27,682	14,228	24,692
tiff2bw	8,516	9,168	3,174	6,016	7,254	2,088	5,806
mutool	17,014	15,560	5,171	13,196	14,391	3,976	12,961
harfbuzz	50,549	38,726	12,548	30,191	41,691	15,203	33,742
jhead	1,132	1,078	705	407	992	698	400
Average	31,126	21,413	11,261	14,277	20,148	10,118	15,399

they also enable rather longer execution time of their adopted *Havoc* as in Figure 2. More specifically, the ranking of the edge coverage performance can almost strictly align with the ranking of the execution time of *Havoc* among all the studied fuzzers (except for Neuzz and FairFuzz). Therefore, we can infer that for any fuzzer, executing *Havoc* for longer time can potentially result in higher edge coverage.

Finding 3:

Executing Havoc for long time upon a fuzzer can potentially result in strong edge coverage performance.

3.4.2 RQ2: performance impact of Havoc under diverse setups. Inspired by the previous findings, we attempt to further investigate the performance impact of *Havoc* on the fuzzers under diverse execution time setups. Specifically, while implementing the default *Havoc* does not concern its execution time, executing *Havoc* under execution time setups essentially demands the modified implementation of integrating *Havoc* to the fuzzers (i.e., the modified *Havoc*).

Implementation. Note that in this paper, we first modify the implementation for integrating *Havoc* to fuzzers in the sequential manner. To begin with, it is essential to figure how to control the execution time of the major fuzzing strategy and *Havoc* of a fuzzer. Specifically, our insight is to retain the fuzzing states of the major fuzzing strategy and *Havoc* when they are halting. To this end, while realizing such insight by directly integrating the source code of *Havoc* into different fuzzers essentially demands substantial engineering effort, we determine to adopt *socket* programming as an alternative solution which executes the major fuzzing mechanism and its appended *Havoc* in different processes since its built-in blocking mechanism can provide the "wake up" function for both monitoring the execution time of an event given its preset timeout and retaining the fuzzing states while halting. Specifically in the beginning, we execute the major fuzzing strategy of a fuzzer for time duration t to generate new seeds. Subsequently, we transmit the file names of the generated seeds to *Havoc* by *socket*. After completing the whole seed transmission, *Havoc* is executed for time duration t as well while the execution of the original mutation strategy of the fuzzer is paused. Note that instead of dynamically setting a mutation count for controlling its execution as the default *Havoc*, our modified *Havoc* iteratively generates new seeds based on the updated collection of the "interesting" seeds within time duration t. Similarly after executing *Havoc*, we transmit the file names of its generated seeds to the original mutation strategy of the fuzzer via *socket* for further seed generations. Such process is iterated until hitting the total time budget.

Evaluation. We first evaluate *Havoc* by setting the iterative time duration t of the major fuzzing strategy/*Havoc* as 1 hour (i.e., executing them for 1 hour respectively under each iteration). As a result, for each fuzzer, its modified *Havoc* can be executed within a total of 12 hours under our 24-hour budget. Table 5 presents the evaluation results of the fuzzers with and without applying such modified *Havoc* where "O" represents the original fuzzers including their major fuzzing strategies and default *Havoc* and "I" represents the associated fuzzer integrated with the modified *Havoc*. Note that since such setup does not fit for the essential mechanisms of the pure *Havoc* and QSYM which execute *Havoc* for the whole execution time, i.e., 24 hours, we retain their results of the previous evaluations in Table 5 simply for illustration.

We can observe that while MOPT with the modified *Havoc* can incur quite close edge coverage compared with its default *Havoc* integration as in Table 3, the rest fuzzers with the modified *Havoc* can incur quite a significant edge coverage enhancement compared with their default *Havoc* integration, e.g., 1.8X for AFL and 44% for Neuzz. Such result can validate our Finding 3. Specifically, MOPT with the default *Havoc* can already incur quite large execution time, i.e., 16.3 hours, and thus can result in a rather strong edge coverage. On the other hand, the execution time of *Havoc* for the

Table 5: Edge coverage results of fuzzers with modified *Havoc*

Programs	Havoc	QSYM	AFL		AFL++		FairFuzz		MOPT		Neuzz		MTFuzz	
			O	I	O	I	O	I	O	I	O	I	O	I
readelf	73,478	69,597	12,112	73,842	9,844	72,766	36,372	71,689	67,505	73,175	43,040	70,358	40,594	69,824
nm	20,696	30,359	7,188	21,398	6,049	25,259	16,456	21,537	23,159	26,602	16,002	22,258	20,863	24,387
objdump	37,401	41,097	13,748	36,775	13,473	35,004	24,204	35,802	38,027	37,358	29,155	35,739	25,369	36,203
size	17,634	23,700	8,262	17,296	4,205	18,393	16,547	18,118	19,172	18,707	13,228	16,121	12,256	17,395
strip	38,200	46,633	15,310	37,136	12,116	37,419	26,622	37,724	37,318	40,006	29,767	35,147	28,981	37,548
djpeg	16,142	19,295	5,388	18,543	5,474	15,628	10,405	14,660	15,805	18,127	15,805	23,420	7,640	15,962
tcpdump	43,482	45,804	14,972	40,581	5,294	41,178	18,457	40,407	44,275	44,394	17,216	39,687	14,067	42,317
xmllint	49,269	46,538	18,189	45,869	15,986	46,379	28,174	45,004	49,618	47,190	28,213	45,985	27,682	47,365
tiff2bw	8,516	9,301	5,372	8,093	3,767	7,645	8,834	8,204	8,707	8,083	9,168	9,260	7,254	8,671
mutool	17,014	17,827	11,529	17,325	11,028	17,280	14,430	17,065	17,438	17,504	15,560	19,438	14,391	18,554
harfbuzz	50,549	59,270	21,713	56,058	16,411	52,451	29,939	50,619	54,178	59,314	38,726	51,498	41,691	52,964
jhead	1,132	4,516	1,012	1,129	1,057	1,124	1,114	1,123	1,127	1,133	1,078	1,127	992	1,134
Average	31,126	34,495	11,233	31,170	8,725	30,877	19,296	30,163	31,361	32,633	21,413	30,836	20,148	31,027

Table 6: Average edge coverage results under different execution time setups

Setup		AFL_{havoc}	$AFL{+}{+}_{havoc}$	$FairFuzz_{havoc}$	$MOPT_{havoc}$	$Neuzz_{havoc}$	$MTFuzz_{havoc}$
Total	Iteration						
24h	1h	31,170	30,877	30,163	32,633	30,836	31,027
	2h	30,451	31,069	30,853	31,465	31,259	31,265
	4h	30,315	30,541	31,296	32,354	31,462	31,472
	12h	30,567	30,247	30,543	31,764	30,975	30,865

rest fuzzers turns to be much longer and thus results in a significant performance gain for them all. Note that for the fuzzers which originally adopts no *Havoc* (i.e., Neuzz and MTFuzz), their edge coverage performance can also be significantly improved compared with their default *Havoc* integration.

More interestingly, we can find that for most fuzzers, they can incur quite close edge coverage on all the benchmark projects averagely, i.e., around 31K. Moreover, their project-wise performance can be quite close as well, e.g., around 71K in project `readelf` and 38K in project `objdump`. Compared with the edge coverage from their original versions, their performance gaps are significantly reduced. To illustrate, we adopt the STD (Standard Deviation) of the average edge coverage for the studied fuzzers. Specifically, the STD of all the fuzzers appending the modified *Havoc* is 819 compared with 8,879 by appending the default *Havoc* while their average edge coverage is 31,118 compared with 20,278. Such result can indicate that by executing *Havoc* for sufficient time, the edge coverage performance gaps of the fuzzers can be significantly reduced. On the other hand, while the performance of many studied fuzzers are significantly improved by extending the execution time of *Havoc* in a sequential manner, their performance are rather close to the pure *Havoc*. Such facts can potentially indicate that *Havoc* can advance many fuzzers to approach their performance bound.

Finding 4:

Executing Havoc for sufficient time can advance many fuzzers to approach their performance bound, i.e., significantly reducing their performance gaps.

We further attempt to investigate how adapting the integration mode of *Havoc* with fuzzers can impact their edge coverage performance. To this end, we first enable diverse setups of the iterative time duration t of *Havoc* in terms of 2 hours, 4 hours, and 12 hours under the total execution time of 24 hours. Table 6 presents the evaluation results under such setups. We can observe that overall, there is no significant performance difference under all the setups. Specifically, the largest gap of the average edge coverage of a given fuzzer is only 3.76%. Such fact can indicate that the edge coverage performance is somewhat resilient to setting the iterative time duration t, i.e., under sufficient total execution time, adapting the execution time of *Havoc* under each iteration can enable rather limited impact on the edge coverage performance of the associated fuzzer.

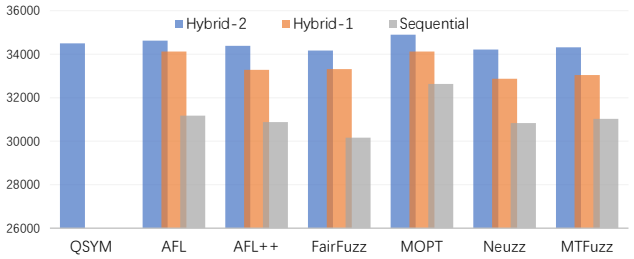

Figure 3: Edge coverage result of different fuzzers with the hybrid integration of *Havoc*

Finding 5:

As long as the total execution time of Havoc can be ensured, how to adapt its iterative execution usually enables limited impact on the edge coverage performance of the associated fuzzer.

While the performance gaps between different fuzzers can be significantly reduced by applying the modified *Havoc*, QSYM can still outperform the other fuzzers by at least 10%. Accordingly,

we infer that executing multiple fuzzing strategies in parallel can be potentially more advanced in facilitating edge exploration. We then attempt to validate such inference by also adopting additional threads for executing *Havoc* in parallel in our studied fuzzers, i.e., while retaining the execution of their modified *Havoc* in the sequential manner for 12 hours, we also execute *Havoc* for the whole 24 hours in parallel in additional threads. In particular, we adopt one and two threads for executing *Havoc* respectively. Figure 3 presents our evaluation results. We observe that when adopting one thread to execute *Havoc* (labelled as "Hybrid-1"), averagely the edge coverage of all the studied fuzzers can be enhanced by 7.4%, which indicates that such hybrid integration of *Havoc* can be more advanced to improve the edge coverage performance. Moreover, we can also observe that compared with adopting one thread for *Havoc*, adopting two threads for *Havoc* (labelled as "Hybrid-2") can further enhance the edge coverage performance by 2.9% on top of all the studied fuzzers. Compared with QSYM which originally enables the optimal performance under three threads for fuzzing, MOPT and AFL can even incur performance gain of 1.2% and 0.4%. On the other hand, since the performance gain by simply increasing additional threads for executing *Havoc* becomes marginal, we can infer that investing more computing resource on executing *Havoc* can advance the fuzzers to approach their performance bound.

Finding 6:

Investing more computing resource in executing Havoc can potentially reduce its execution time for approaching the performance bound.

While the previous findings can reveal that under sufficient execution time of *Havoc*, multiple fuzzers can approach quite close edge coverage performance, we further attempt to investigate how common their explored edges can be. To this end, we determine to adopt the concept of *Jaccard distance* [49] to delineate the similarity of the explored edges from different fuzzers. In particular, *Jaccard distance* is usually used to measure the dissimilarity between two sets by dividing the difference of their union size and intersection size by their union size. Figure 4 presents the evaluation results of *seed dissimilarity* between the pure *Havoc* and the other fuzzers on average, ranging from 0.134 to 0.256. Such result indicates that applying *Havoc* to different fuzzers can potentially explore quite common edges.

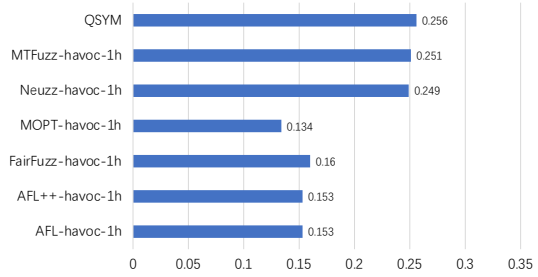

Figure 4: The average *Jaccard distance* of different fuzzers in all studied programs.

Table 7: The unique crashes found by *Havoc*

Programs	Crashes	Unique crashes	
		Havoc	Major
readelf (V2.30)	81	50	0
nm (V2.36)	89	78	1
objdump (V2.30)	1	1	0
size (V2.30)	4	2	1
strip (V2.30)	12	12	0
djpeg (V9c)	4	4	0
jhead (V3.04)	688	96	11
Total	879	243	13

Finding 7:

Applying Havoc to different fuzzers can potentially explore rather common edges.

At last, we investigate the impact of appending *Havoc* on exposing program vulnerabilities. To this end, we attempt to collect the program crashes caused by executing the generated seeds with and without appending *Havoc* to all the studied fuzzers. Note that when we append *Havoc* to fuzzers, we ensure that it can be executed under sufficient time to fully leverage its power.

To begin with, it is essential to identify unique crashes since it is likely that many crashes are caused by only one program vulnerability. In this paper, we follow many prior works [7][30][6][10][23][52][24] to identify the unique crashes only if they enhance edge coverage. Note that in this paper, all of the crashes are explored by all of our previous evaluations and we only record a crash once for the same fuzzer. However, such unique crashes can be possibly explored by different fuzzers. We then divide crashes into two sets, i.e., the ones explored by the *Havoc*s and the ones explored by the major fuzzing strategies. At last, we count the unique crashes for the two sets respectively.

Table 7 presents the results of the unique crashes. Overall, we derive 256 unique crashes from a total of 879 crashes where 243 (95%) are exposed by *Havoc* and 13 are exposed by their original mutation strategies, e.g., the SMT solver in QSYM and the gradient-based mutations in Neuzz. Note that we exposed 69 unique cashes which have been fixed in the latest versions of their associated projects [17][3][4][5]. We also report the rest unknown crashes (i.e., they can be exposed in the latest version) to the corresponding developers [2][31]. Moreover, applying *Havoc* can expose the crashes in 7 of the 12 total benchmark projects and be powerful in exposing unique crashes in projects `nm` (78 out of 79) and `jhead` (96 out of 107). Such facts can indicate that applying *Havoc* can not only successfully advance program vulnerability exposure, but also potentially dominate the effectiveness on certain projects.

Finding 8:

Havoc can play a vital role in exposing program vulnerabilities.

4 ENHANCING *HAVOC*

Our presented powerful performance of *Havoc* is simply caused by modifying its setups, including its execution time and integration modes with fuzzers so far. In this section, we attempt to investigate whether the power of *Havoc* can be further leveraged. To this end, we first investigate the performance impact of the mutator stacking mechanism adopted by *Havoc*, and then propose an intuitive and lightweight technique to improve the edge coverage performance of *Havoc* accordingly.

4.1 Performance Impact from the Mutator Stacking Mechanism

Note that as a simplified mutation strategy, the mutator stacking mechanism contains two technical steps: determining *stacking size* and randomly selecting mutators, to impact the performance of *Havoc*. We then investigate the performance impact caused by each step. In particular, we first attempt to investigate the performance impact from *stacking size*. To this end, instead of randomly determining *stacking size* for mutating seeds at runtime of the original *Havoc*, we implement *Havoc* under a fixed *stacking size* for all its mutations. Figure 5 presents our evaluation results of the edge coverage ratio results in terms of the all the possible fixed *stacking size*, i.e., 2, 4, 8,...128, on top of all the studied benchmark projects. Note that the edge coverage ratio of one project is computed as the the explored edge number in terms of one fixed *stacking size* over the total explored edge number of all the fixed *stacking size*s. We can observe that overall, the *stacking size* which causes the optimal edge coverage performance for each studied project can be quite divergent, e.g., selecting *stacking size* 8, 2, and 32 can optimize the edge coverage in `tcpdump`, `djpeg`, and `mutool` respectively. Such results can suggest that it is essential to adapt the *stacking size* setup for different projects to optimize their respective edge coverage performance.

We then investigate the performance impact from mutators. To this end, instead of uniformly selecting mutators out of a total of 15 mutators, we first uniformly select *chunk mutators* or *unit mutators* and then randomly select their inclusive mutators under the given *stacking size* for mutating one seed. Figure 6 presents the edge coverage ratio results in terms of the selected mutator types on top of all the studied benchmark projects. Note that the edge coverage ratio is computed as the explored edge number in terms of either *chunk mutators* or *unit mutators* over their total explored edge number. We can observe that overall, the distribution of the edge coverage ratio performance can be quite divergent among different projects, e.g., edge coverage ratio of the *unit mutators* ranges from 18.39% (`xmllint`) to 94.53% (`tiff2bw`). Such result can suggest that it is also essential to adapt the selection of the mutator types for different projects to optimize their respective edge coverage performance.

4.2 Approach

Inspired by the evaluation results above, we attempt to propose solutions to enhance *Havoc* on dynamically adjusting the project-wise selections on *stacking size* and mutators. Note that our previous findings reveal that to unleash the power of *Havoc*, it is essential to invest strong computing resources for *Havoc*. Accordingly, our design for such technique adopts the following principles. First, such technique only enables single thread/process, i.e., we attempt to enhance *Havoc* via only applying our designed mechanism instead of realizing the trade-off by leveraging more threads/processes for more computing resource as found already. Second, such mechanism should be lightweight. In particular, when designing a mechanism for adjusting *Havoc* given the deterministic computing resource, ideally we aim for minimizing its execution time while maximizing the execution time for the *Havoc* mechanism itself.

In this paper, we propose a lightweight single-threaded technique *Havoc*$_{DMA}$ (DMA refers to **D**ynamic **M**utation **A**djustment) for the pure *Havoc* to automatically adjust its selections on *stacking size* and mutators at runtime for facilitating its edge exploration. Specifically, we determine to model our task as a multi-armed bandit problem [50] which typically refers to allocating limited resources to alternative choices (i.e., *stacking size* and mutator selections) to

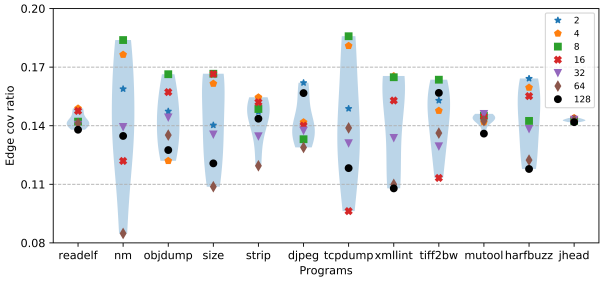

Figure 5: The edge coverage ratio in terms of the fixed stack length

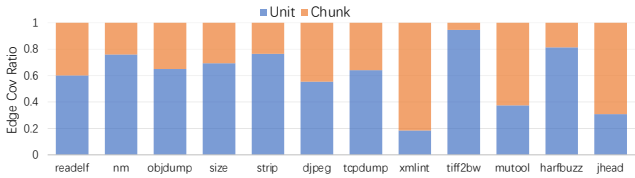

Figure 6: The edge coverage ratio of *unit mutators* and *chunk mutators*

Algorithm 1 The Framework of *Havoc*$_{DMA}$

Input : seed
Output:newseed

```
1:  function MULTI_ARMED_UCB_SELECTION
2:      newseed ← seed
3:      stacksize ← selectStackArm()
4:      mutatortype ← selectMutatorTypeArm(stacksize)
5:      for iteration in stacksize do
6:          mutator ← randomSelectMutatorByType(mutatortype)
7:          newseed ← generateNewSeed(mutator, newseed)
8:      reward ← 0
9:      if isInteresting(newseed) then
10:         reward ← 1
11:     updateStackBandit(reward, stacksize)
12:     updateMutatorTypeBandit(reward, stacksize, mutatortype)
13:     return newseed
```

maximize their expected gain (i.e., edge coverage). More specifically, we design a two-layer multi-armed bandit machine, i.e., a *stacking size*-level bandit machine and a mutator-level bandit machine, which is presented in Figure 7. Note that the *stacking size*-level bandit machine enables 7 arms where each arm is designed corresponding to a *stacking size* choice, i.e., 2, 4, 8,...128. After an arm of *stacking size* is chosen, the mutator-level bandit machine which enables 2 arms representing *chunk mutators* and *unit mutators* would first make a choice out of them and then proceed to select the exact mutators via uniform distribution. Eventually, $Havoc_{DMA}$ generates a mutant via the selected mutators and execute it on the program under test for obtaining environmental feedback for further executions.

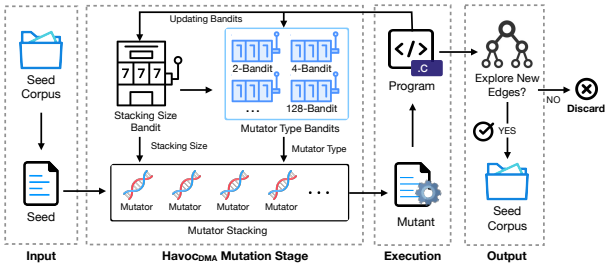

Figure 7: The Framework of $Havoc_{DMA}$

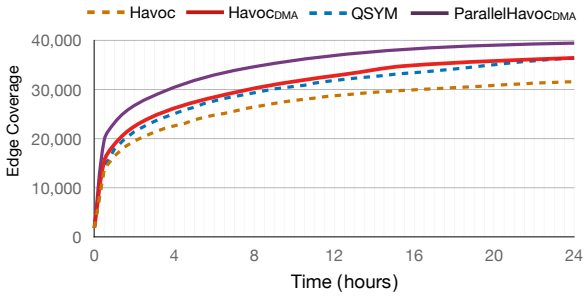

Figure 8: The average edge coverage of $Havoc_{DMA}$ over time

We adopt the widely-used UCB1-Tuned [1] algorithm to solve our proposed multi-armed bandit problem. Equation 1 demonstrates how to select *arm* under such algorithm for a given bandit machine at time t. In particular, $\bar{x}_j$ refers to the average reward for arm_j till time t, n refers to the total execution count for the bandit machine and n_j refers to the execution count for arm_j, σ_j refers to the sample variance of arm_j.

$$arm(t) = \arg\max_j \left(\bar{x}_j + \sqrt{\frac{\ln n}{n_j} min(\frac{1}{4}, \sigma_j + \frac{2 \ln n}{n_j})} \right) \quad (1)$$

Note that we define the reward at time t as whether $Havoc_{DMA}$ has explored new edges or not for all the eight bandit machines. If a seed generated by a chosen *stacking size* and its selected mutators can explore new edges, the reward returned to the *stacking size*-level bandit machine and its corresponding mutator-level bandit machine is 1 respectively and 0 otherwise.

Algorithm 1 presents our overall approach. $Havoc_{DMA}$ first selects *stacking size* for the executing seed and then selects its corresponding mutator type (lines 3 to 4). Next, $Havoc_{DMA}$ generates a mutant by uniformly selecting the mutators of *stacking size* under the chosen type (lines 5 to 7). Eventually, if such mutant explores new edges, we set the reward as 1 for its corresponding *stacking size*-level and mutator-level bandit machines and 0 otherwise to update Equation 1 for further executions (lines 8 to 12).

4.3 Evaluation

To evaluate $Havoc_{DMA}$, we include QSYM for performance comparison since it presents the optimal edge coverage performance in our previous studies. Furthermore, we design a variant of $Havoc_{DMA}$ namely $ParallelHavoc_{DMA}$ where $Havoc_{DMA}$ is executed in three threads in parallel for comparing with QSYM under the identical computing resources. We also include the pure *Havoc* as a baseline. Similar as Section 3.2, we execute each variant for five times for each benchmark project to reduce the impact of randomness.

Figure 8 presents the average evaluation results of edge coverage of the studied approaches on top of all the benchmark projects under 24-hour execution. We can observe that $Havoc_{DMA}$ achieves significantly better performance than pure *Havoc*, i.e., enhancing the average edge coverage among all the benchmark projects by 11% (34,574 vs 31,126 explored edges). Interestingly, although $Havoc_{DMA}$ only adopts one thread for execution, it can slightly outperform QSYM which enables three threads for execution, i.e., by 0.2%. On the other hand, executing $ParallelHavoc_{DMA}$ can result in 9.04% edge coverage gain over QSYM (37,614 vs 34,495 explored edges). Such results altogether can demonstrate the strength of our proposed $Havoc_{DMA}$.

Figure 9 presents the edge coverage trends of our studied approaches upon each benchmark for 24-hour execution. Overall, $Havoc_{DMA}$ outperforms pure *Havoc* in all of the benchmarks significantly. Moreover, $Havoc_{DMA}$ can outperform QSYM at least 10% (60% more in `tiff2w`) in terms of edge coverage in five projects while incurring rather close performance in the rest projects with a single thread. Meanwhile, $ParallelHavoc_{DMA}$ can achieve the optimal edge coverage performance on eight benchmarks.

5 THREATS TO VALIDITY

Threats to internal validity. One threat to internal validity lies in the implementation of the studied fuzzers in the evaluation. To reduce this threat, we reused the existing source code of them for our implementation accordingly. Moreover, all the authors including two faculty members manually reviewed all the codes carefully to ensure its correctness and consistency. Another threat to internal validity may lie in the implementation of our proposed approach $Havoc_{DMA}$ where the specific mutators and the *stacking size* adopted by different fuzzers can be possibly varying. However, $Havoc_{DMA}$ can extensively cope with such issue since it only attempts for dynamically adapting the mutator stacking strategy which does not concern the exact mutator types.

Threats to external validity. The threats to external validity mainly lies in the subjects and benchmarks. To reduce this threat, we select 8 representative state-of-the-art fuzzers which includes multiple types, including AFL-based, Cocncolic-execution-based,

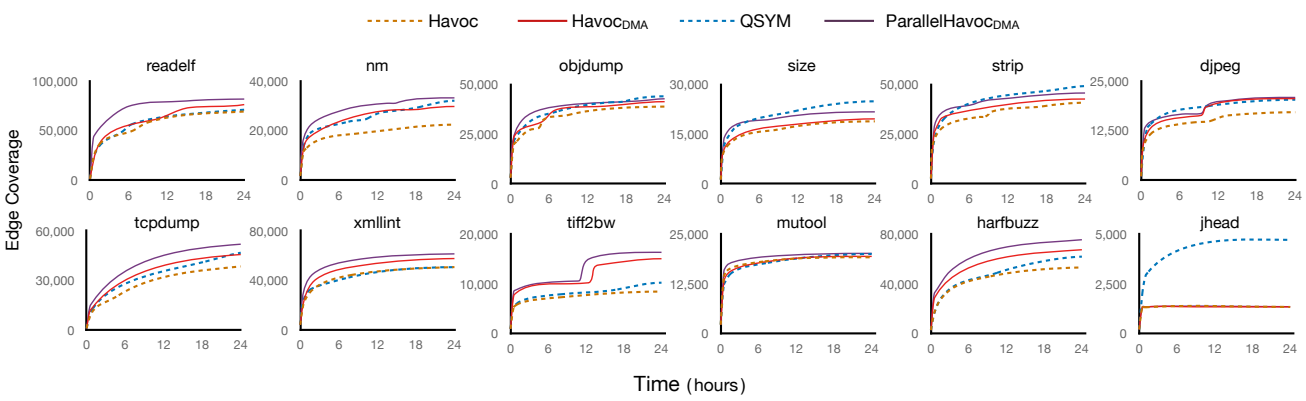

Figure 9: Edge coverage of *Havoc*$_{DMA}$ over time

and neural program-smoothing-based fuzzers. We also adopt 12 benchmarks according to their popularity, i.e., the most frequently used benchmarks by the original papers of our studied fuzzers.

Threats to construct validity. The threats to construct validity mainly lies in the metrics used, i.e., edge coverage to reflect code coverage. To reduce such threat, while there can be various ways to measure edge coverage, We determine to follow many existing fuzzers [24][51][41][42][9] via the AFL built-in tool named `afl-showmap`.

6 RELATED WORK

Search-based Software Engineering. The random search methodologies such as Evolutionary Programming are widely adopted to solve software engineering problems by modeling them into search problems. Chen et al. [12] seeked to distribute test cases evenly within the input space. Pacheco et al. [32] proposed RANDOOP to generate unit tests for Java code using feedback-directed random test generation. Harman et al. [20] introduced how Pareto optimal search can improve search-based refactoring for software engineering. Chen et al. [11] proposed a meta multi-objectivization model to tune the software configuration. Lin et al. [28] specified the mutation points for randomly generating test cases via the program graph constructed by static analysis. Patra et al. [33] proposed SemSeed, a tool that learns the pattern of real-world bugs by utilizing a random search algorithm for creating actual bugs. Rabin et al. [36] presented SIVAND, a simple model-agnostic approach to identify critical input features for models in CI systems. Chakraborty et al. [8] presented a Fair-SMOTE algorithm to remove the bias label and rebalance positive and negative samples in the machine learning tasks for software engineering.

Coverage-guided Fuzzing. AFL [52] is one of the most popular fuzzers and the fundamental implementations for many other fuzzers. Fioraldi et al. [16] integrated different techniques into the basic framework of AFL, e.g., taint tracking. Böhme et al. [7] utilized a Markov chain model to allocate energy to different seeds for further selection. Compared with the fuzzers implemented upon AFL, libFuzzer [29] proposed by the LLVM community focuses on fuzzing the binary library instead of an executable program. Peng et al. [34] proposed T-Fuzz, which removes sanity checks from the target program and then leverages a symbolic execution engine to generate a path to the buggy point if it finds any crash. Honggfuzz [19] utilized multi-process and multi-thread to boost up the whole fuzzing process. Chen et al. [13] proposed a synchronization mechanism for integrating different fuzzers to fuzz a program. Sergej et al. [40] proposed NYX, a coveraged-guided fuzzer to fuzz hypervisors which are also known as virtual machine monitors. Wang et al. [45] proposed SYZVEGAS to fuzz the kernel of operating systems by reinforcement learning. Li et al. [26] introduced Steelix, which integrates light-weight static analysis to coverage-guide fuzzing. Wang et al. [46] proposed Skyfire, which leverages the knowledge in the vast amount of existing samples to generate well-distributed seed inputs for fuzzing programs that process highly-structured inputs. They further proposed a grammar-aware coverage-based greybox fuzzing approach to fuzz programs that process structured inputs named Superion [47]. In this paper, we propose a technique to dynamically adjust mutation selections for *Havoc* and result in strong edge coverage performance.

Studies on Testing. Shen et al. [43] investigated different bugs on different deep learning compiler. Li et al. [25] conducted a study on injected configuration testing and provided guidelines for improving the effectiveness of configuration error injection testing. Ren et al. [38] demonstrated that different experiment setups can affect the performance of smart contract testing tools. Elsner et al. [15] conducted a large-scale empirical study to evaluate RTP and unsafe RTS that exclusively rely on CI and VCS metadata. Peng et al. [35] conducted a large evaluation on test-case prioritization (TCP) and improved the effectiveness of TCP. Herrera et al. [21] systematically investigated and evaluated how seed selection affects the performance of a fuzzer to expose vulnerabilities in real world systems. Klees et al. [23] surveyed the recent research literature and assessed the experimental evaluations to illustrate what experimental setup is needed to deliver reliable results for different fuzzers. Liang et al. [27] presented the main obstacles and corresponding typical solutions for fuzzing. Tonder et al. [44] presented a method to map crashing inputs to unique bugs using program transformation. Choi et al. [14] experimented on thousands of DLL files and API functions in Windows system to reveal the potential security issue in Windows. In this paper, we conduct an extensively study

on *Havoc* to demonstrate that *Havoc* can significantly enhance edge coverage.

7 CONCLUSION

In this paper, we investigate the impact and mechanism of a random fuzzing strategy *Havoc*. We first conduct an extensive study to evaluate the impact of *Havoc* by applying *Havoc* to a set of studied fuzzers on real-world benchmarks. The evaluation results demonstrate that the pure *Havoc* can incur significant edge coverage and vulnerability exposure performance compared with other fuzzers. Moreover, integrating *Havoc* to a fuzzer or extending total execution time for the existing *Havoc* implementations can also enhance the edge coverage significantly. The performance gap among different fuzzers can also be considerably reduced by appending *Havoc*. At last, we also construct a lightweight approach to enhance *Havoc* by dynamically adjusting the mutation strategies.

REFERENCES

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2 (2002), 235–256.

[2] GNU Binutils. 2021. Bug 28269 - [nm] stack-overflow in nm-new 'demangle path'. https://sourceware.org/bugzilla/show_bug.cgi?id=28269.

[3] GNU Binutils. 2021. Bug 28272 - [strip] SEGV in group signature - v2.30. https://sourceware.org/bugzilla/show_bug.cgi?id=28272.

[4] GNU Binutils. 2021. Bug 28273 - [strip] heap-use-after-free in 'group signature'. https://sourceware.org/bugzilla/show_bug.cgi?id=28273.

[5] GNU Binutils. 2021. Bug 28274 - [strip] heap-buffer-overflow. https://sourceware.org/bugzilla/show_bug.cgi?id=28274.

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[8] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in Machine Learning Software: Why? How? What to Do?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 429–440. https://doi.org/10.1145/3468264.3468537

[9] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[10] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[11] Tao Chen and Miqing Li. 2021. Multi-Objectivizing Software Configuration Tuning (for a single performance concern). *arXiv preprint arXiv:2106.01331* (2021).

[12] Tsong Yueh Chen, Hing Leung, and IK Mak. 2004. Adaptive random testing. In *Annual Asian Computing Science Conference*. Springer, 320–329.

[13] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1967–1983.

[14] YoungHan Choi, HyoungChun Kim, and DoHoon Lee. 2008. An Empirical Study for Security of Windows DLL Files Using Automated API Fuzz Testing. In *2008 10th International Conference on Advanced Communication Technology*, Vol. 2. 1473–1475. https://doi.org/10.1109/ICACT.2008.4494042

[15] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 491–504. https://doi.org/10.1145/3460319.3464834

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[17] Probe Fuzzer. 2018. Reachable assertion in find section (src/binutils/readelf.c). https://lists.gnu.org/archive/html/bug-binutils/2018-02/msg00076.html.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[19] Google. 2021. Honggfuzz. https://github.com/google/honggfuzz.

[20] Mark Harman and Laurence Tratt. 2007. Pareto optimal search based refactoring at the design level. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 1106–1113.

[21] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 230–243.

[22] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, Vol. 4. IEEE, 1942–1948.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[24] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[25] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and Opportunities: An in-Depth Empirical Study on Configuration Error Injection Testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490. https://doi.org/10.1145/3460319.3464799

[26] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[27] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 562–566. https://doi.org/10.1109/SANER.2018.8330260

[28] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-Based Seed Object Synthesis for Search-Based Unit Testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1068–1080. https://doi.org/10.1145/3468264.3468619

[29] LLVM. 2021. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1949–1966.

[31] Matthias-Wandel. 2021. Nonfatal Error by heap-buffer-overflow (version 3.04). https://github.com/Matthias-Wandel/jhead/issues/42.

[32] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[33] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 906–918. https://doi.org/10.1145/3468264.3468623

[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056

[35] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 324–336. https://doi.org/10.1145/3395363.3397383

[36] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding Neural Code Intelligence through Program Simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3468264.3468539

[37] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[38] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. Empirical Evaluation of Smart Contract Testing: What is the Best Choice?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 566–579. https://doi.org/10.1145/3460319.3464837

[39] Github Repository. 2021. Havoc-Study. https://github.com/MagicHavoc/Havoc-Study.

[40] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wör-ner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2597–2614. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

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

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

[43] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 968–980.

[44] Rijnard van Tonder, John Kotheimer, and Claire le Goues. 2018. Semantic Crash Bucketing. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 612–622. https://doi.org/10.1145/3238147.3238200

[45] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[46] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594. https://doi.org/10.1109/SP.2017.23

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

[48] Wikipedia. 2021. Exposing Bugs by Fuzzing. https://en.wikipedia.org/wiki/Fuzzing#Exposing_bugs.

[49] Wikipedia. 2021. Jaccard Distance. https://en.wikipedia.org/wiki/Jaccard_index.

[50] Wikipedia. 2021. Multi-armed Bandit Problem. https://en.wikipedia.org/wiki/Multi-armed_bandit.

[51] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761.

[52] Michał Zalewski. 2020. American Fuzz Lop. https://github.com/google/AFL.