One Fuzzing Strategy to Rule Them All

Mingyuan Wu†
Southern University of Science and Technology, Shenzhen, China and the University of Hong Kong, Hong Kong, China
11849319@mail.sustech.edu.cn

Ling Jiang, Jiahong Xiang
Southern University of Science and Technology, Shenzhen, China
11711906@mail.sustech.edu.cn
11812613@mail.sustech.edu.cn

Heming Cui
The University of Hong Kong
Hong Kong, China
heming@cs.hku.hk

Lingming Zhang
University of Illinois
Urbana-Champaign, USA
lingming@illinois.edu

Yanwei Huang
Zhejiang University
Hangzhou, China
huangyw@zju.edu.cn

Yuqun Zhang*
Southern University of Science and Technology, Shenzhen, China
zhangyq@sustech.edu.cn

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 exploration. However, they only tend to adopt the default setup of Havoc as an implementation option while none of them attempts to explore its power under diverse setups or inspect its rationale for potential improvement. In this paper, to address such issues, we conduct the first empirical study on Havoc to enhance the understanding of its characteristics. Specifically, we first find that applying the default setup of Havoc to fuzzers can significantly improve their edge coverage performance. Interestingly, we further observe that even simply executing Havoc itself without appending it to any fuzzer can lead to strong edge coverage performance and outperform most of our studied fuzzers. Moreover, we also extend the execution time of Havoc and find that most fuzzers can not only achieve significantly higher edge coverage, but also tend to perform similarly (i.e., their performance gaps get largely bridged). Inspired by the findings, we further propose Havoc\_MB\_A, 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\_MB\_A can significantly increase the edge coverage by 11.1% on average for all the benchmark projects compared with Havoc and even slightly outperform state-of-the-art QSYM which augments its computing resource by adopting three parallel threads. We further execute Havoc\_MB\_A with three parallel threads and result in 9% higher average edge coverage over QSYM upon all the benchmark projects.

ACM Reference Format:

1 INTRODUCTION

Fuzzing (or fuzz testing) refers to an automated software testing methodology that inputs invalid, unexpected, or random data to programs for exposing unexpected program behaviors (such as crashes, failing assertions, or memory leaks), which can be further inspected or analyzed to detect potential vulnerabilities/bugs [43]. In particular, many existing fuzzers tend to facilitate their vulnerability/bug exposure via optimizing code coverage of programs [7, 20, 33, 34, 52]. Given an initial collection of seeds, such coverage-guided fuzzers usually develop strategies to iteratively mutate them for generating new seeds that can trigger higher code coverage. Notably, a number of recent coverage-guided fuzzers (e.g., AFL [52], AFL++ [12], MOPT [25], QSYM [50], and FairFuzz [20]) integrate a lightweight random search mechanism namely Havoc\_A to their respective fuzzing strategies for increasing their code coverage. 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 explore 7.8k+ program edges within only around 3 hours. In contrast to many existing fuzzers which adopt only one mutator under each iterative execution, Havoc randomly selects multiple diverse mutators, e.g., flipping a single bit and inserting/deleting a randomly-chosen continuous chunk of bytes, and applies them altogether for generating one seed during each iteration. Typically, under each iteration, 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.

† Mingyuan Wu is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China.
* Yuqun Zhang is the corresponding author. He is also affiliated with the Research Institute of Trustworthy Autonomous Systems, Shenzhen, China and Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation, China.
Although Havoc has been widely adopted by existing fuzzers, they tend to include Havoc only as an implementation option without further investigating its rationale 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 increase code coverage. That said, they simply adopt Havoc under its default setup, i.e., none of the prior work attempt to study the impact of different Havoc settings, explore different ways to integrate Havoc, or further boost the Havoc strategy itself.

In this paper, we conduct the first comprehensive study of Havoc to unleash its potential. 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 increase their edge coverage upon all the benchmark projects from 43.9% to 3.7X on average. Meanwhile, we also find that even directly applying the pure Havoc only can result in surprisingly strong edge coverage and significantly outperform most of our studied fuzzers. Moreover, while different fuzzers can achieve quite divergent edge coverage results, applying Havoc to the studied fuzzers under sufficient execution time can in general not only significantly increase 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. Lastly, Havoc can also help all the studied fuzzers expose more unique crashes than their corresponding major fuzzing strategies.

Inspired by our findings, we propose an improved version of Havoc namely Havoc$_{MAB}$ [32] which models the Havoc mutation strategy as a multi-armed bandit problem (MAB) [45] to be further solved by dynamically adjusting the mutation strategy. The evaluation results indicate that under 24-hour execution, Havoc$_{MAB}$ can outperform the pure Havoc significantly by 11.1% in terms of edge coverage on all the benchmarks on average. Havoc$_{MAB}$ can also slightly outperform state-of-the-art QSYM which augments its computing resource by adopting three threads in parallel. Moreover, we also design Havoc$_{3MAB}$ by executing Havoc$_{MAB}$ with three threads in parallel. The evaluation result indicates that Havoc$_{3MAB}$ can outperform state-of-the-art QSYM by 9% 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 substantially improve edge coverage and crash detection for all the studied fuzzers.
- We propose a lightweight approach Havoc$_{MAB}$ based on our findings which can boost the pure Havoc by 11.1% under a 24-hour execution, and outperform all the other studied fuzzers.

2 BACKGROUND

Havoc was first proposed in AFL [52] and later further adopted by many other fuzzers [6, 7, 12, 20, 25]. While their adoptions of Havoc can be slightly different, they typically integrate Havoc with their major fuzzing strategies (i.e., the core 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 the major 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 a seed, Havoc implements mutator stacking, i.e., mutating it by randomly applying multiple mutators (e.g., 15 for AFL, MOPT, etc.) in order from a set of mutators. 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. Accordingly, its fuzzer can resume the execution of its major fuzzing strategy when needed.

![Figure 1: The framework of Havoc](image)

Table 1: Mutation operators defined by Havoc

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Mutator</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitflip</td>
<td>Flip a bit at a random position.</td>
<td>bitflip 1</td>
</tr>
<tr>
<td>interesting values</td>
<td>Set bytes with hard-coded interesting values.</td>
<td>interest 8, interest 16, interest 32</td>
</tr>
<tr>
<td>arithmetic increase</td>
<td>Perform addition operations.</td>
<td>addition 8, addition 16, addition 32</td>
</tr>
<tr>
<td>arithmetic decrease</td>
<td>Perform subtraction operations.</td>
<td>decrease 8, decrease 16, decrease 32</td>
</tr>
<tr>
<td>random value</td>
<td>Randomly set a byte to a random value.</td>
<td>random byte</td>
</tr>
<tr>
<td>delete bytes</td>
<td>Randomly delete consecutive bytes.</td>
<td>delete chunk bytes</td>
</tr>
<tr>
<td>clone/insert bytes</td>
<td>Clone bytes in 75%, otherwise insert a block of constant bytes.</td>
<td>clone/insert chunk bytes</td>
</tr>
<tr>
<td>overwrite bytes</td>
<td>Randomly overwrite the selected consecutive bytes.</td>
<td>overwrite chunk bytes</td>
</tr>
</tbody>
</table>
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 refers to flipping one random bit at a random position. To our best knowledge, most fuzzers [6, 7, 12, 25, 50, 52] 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.

While many fuzzers [15, 24, 31, 33, 34] mainly apply one mutator to a seed each time, Havoc enables mutator stacking to stack and apply multiple mutators 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 by 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 fuzzers uniformly select mutators for their Havoc, MOPT and AFL++ adopt a probability distribution generated by Particle Swarm Optimization [18] 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 under each iterative execution. The other is the parallel manner, i.e., applying Havoc and the major fuzzing strategy of a fuzzer in parallel. For instance, QSYM [50] enables three threads which execute Havoc, AFL deterministic mutation strategy, and concolic execution [14] respectively; more specifically, the first two threads 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 utilized as an implementation option 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 integrate Havoc under appropriate effort. Accordingly, we can investigate whether and how Havoc can be effective in a wider range of 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 are many existing fuzzers which can meet our selection criteria above, we also need to filter them for selecting the representative ones. 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 the engineering-/concept-wise challenges. 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 fuzzers with Havoc (AFL [52], AFL++ [12], MOPT [25], FairFuzz [20], QSYM [50]), 2 fuzzers without Havoc (Neuzz [34], MTFuzz [33]) and the pure Havoc itself. Note that such subjects can be rather representative in terms of technical designs, 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 [20, 25, 33, 34, 50]. In particular, we select 12 frequently used projects out of the papers to form our benchmark for evaluation. More specifically, we first select all 6 projects that are adopted by at least 3 papers; then, we further randomly select another 6 projects which are adopted by one or two papers. The selection details are presented in our Github page [32]. Table 2 presents the statistics of our adopted benchmarks. Specifically, we consider our benchmark to be sufficient and representative due to following reasons:

1. These 12 benchmark projects cover 7 different file formats for seed inputs, e.g., ELF, JPEG, and TIFF;
2. The sizes of these programs that range from 1,885 to over 120K LoC can represent a wide range of programs in practice;
3. They cover diverse functions including development tools (e.g., readelf, objdump), code processing tools (e.g., tiff2bw), graphics processing tools (e.g., jpeg), network analysis tools (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 GPUs.

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 [6, 7, 19, 20, 33, 34]. 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 source code during compilation. We also follow the instructions mentioned in previous work [11, 17, 19, 20, 41] to construct initial seed corpus. In particular, we collect initial seeds for libjpeg, libtiff and jhead from AFL official seed corpus [53] and for the rest projects from their own test suites.

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 compilation. We also follow the instructions mentioned in previous sections.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Target</th>
<th>Class</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>re Hotel</td>
<td>ELF</td>
<td>72,164</td>
<td></td>
</tr>
<tr>
<td>nm Hotel</td>
<td>ELF</td>
<td>55,307</td>
<td></td>
</tr>
<tr>
<td>binutils-2.36</td>
<td>objdump</td>
<td>ELF</td>
<td>74,552</td>
</tr>
<tr>
<td>size Hotel</td>
<td>ELF</td>
<td>54,429</td>
<td></td>
</tr>
<tr>
<td>strip Hotel</td>
<td>ELF</td>
<td>65,432</td>
<td></td>
</tr>
<tr>
<td>libjpeg-9c</td>
<td>djpeg</td>
<td>JPEG</td>
<td>9,023</td>
</tr>
<tr>
<td>tcpdump-4.99.0</td>
<td>tcpdump</td>
<td>PCAP</td>
<td>46,892</td>
</tr>
<tr>
<td>libxml2-2.9.12</td>
<td>xmlint</td>
<td>XML</td>
<td>73,320</td>
</tr>
<tr>
<td>libtiff-4.2.0</td>
<td>tiff2bw</td>
<td>TIFF</td>
<td>15,024</td>
</tr>
<tr>
<td>mupdf-1.18.0</td>
<td>muttool</td>
<td>PDF</td>
<td>123,575</td>
</tr>
<tr>
<td>harfbuzz-2.8.0</td>
<td>harfbuzz</td>
<td>TTF</td>
<td>9,847</td>
</tr>
<tr>
<td>jhead-3.04</td>
<td>jhead</td>
<td>JPEG</td>
<td>1,885</td>
</tr>
</tbody>
</table>

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 (note that Neuzz and MTFuzz, marked in red, do not have the Havoc stage by default, and will be discussed later). We can observe that while AFL, AFL++, and FairFuzz allow quite limited total execution time of Havoc by default (i.e., from 0.79 hour to 3.09 hours), MOPT and QSYM allow much longer execution time for Havoc. Note that the default setting of QSYM utilizes three threads including the default Havoc. Thus Havoc is executed in QSYM for the whole 24 hours as mentioned in Section 2.2.

3.3 Research Questions

We investigate the following research questions for extensively studying Havoc.

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

3.4 Result Analysis

3.4.1 **RQ1: Performance Impact of the Default Havoc**. We first investigate the impact of the default Havoc on the fuzzers with Havoc. As mentioned in Section 2.2, there can be typically two default setting types 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

We first study the Havoc impact on the five fuzzers with Havoc. Specifically, we create their variants by deleting Havoc from their original implementations, i.e., only retaining their major fuzzing strategies. Table 3 presents the edge coverage results of the five fuzzers with Havoc in terms of their major fuzzing strategies (represented as “Major”) and the original implementations (represented as “Original”) respectively. Generally, we can observe that the edge coverage of all the studied fuzzers decrease significantly after deleting Havoc from their implementations averagely, i.e., 9.7% in AFL, 27.0% in AFL++, 32.2% in FairFuzz, 79.4% in MOPT, and 62.2% in QSYM. Combining Figure 2, we can further infer that the 79.4% edge coverage decrease for MOPT is caused by reassigning 16.3 hours (67.9% of all time budget) originally spent on Havoc to its major strategy; the 62.2% edge coverage decrease of QSYM is caused by excluding the thread executing Havoc. Even for AFL and AFL++ which executes their Havoc only less than 1 hour, excluding Havoc decreases 9.7% in AFL and 27.0% in AFL++ in terms of edge coverage. All such facts indicate that Havoc can significantly increase the edge coverage over the major fuzzing strategies.

We also attempt to append the default Havoc into the fuzzers without Havoc, 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 fuzzing strategies of Neuzz and MTFuzz under each iterative execution. Therefore, the execution time of the default Havoc adopted by them cannot be deterministic.

In particular, their execution time distributions are presented in
Figure 2 where Neuzz spends 2.25 hours and MTFuzz spends 2 hours on executing the default Havoc.

Table 4 presents the edge coverage results where "Origin" refers to the original versions of Neuzz and MTFuzz, while "Integration" refers to Neuzz and MTFuzz integrated with Havoc. We can observe that overall, for the new integrated version, Havoc can achieve 78.9%/66.0% higher edge coverage than the major fuzzing strategy of Neuzz/MTFuzz on average. Moreover, the integrated fuzzers can achieve 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 all the studied fuzzers (no matter originally integrated with Havoc or not), appending the default Havoc to them can significantly enhance their major/original fuzzing strategies.

Finding 1: Applying Havoc by the default setup can significantly improve the edge coverage performance of the studied fuzzers.

Interestingly, we can find from Table 4 that the pure Havoc, i.e., using only one seed to launch Havoc and executing it all along without appending it to any fuzzer, preforms rather strong in terms of edge coverage, 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., applying 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 is a powerful fuzzer.

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 achieve much stronger edge coverage over the other fuzzers according to Tables 3 and 4, they also have longer execution time for Havoc as shown 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 most fuzzers, executing Havoc for longer time potentially results in higher edge coverage.

Finding 2: Havoc is essentially a powerful fuzzer—executing Havoc under one seed without being appended to any fuzzer for sufficient time can already achieve superior edge coverage over many existing fuzzers.

Finding 3: Executing Havoc for a longer time upon a fuzzer can potentially result in stronger edge coverage performance.
We execute the major fuzzing strategy of a fuzzer for time duration \( t \) to incur quite close edge coverage compared with its default setup. Simply for illustration and comparison, we retain their results of the previous evaluations in Table 5 as the default \( \text{Havoc} \) setup does not fit for the essential mechanisms of the pure \( \text{Havoc} \) fuzzer integrated with the modified \( \text{Havoc} \) result, for each fuzzer, its modified \( \text{Havoc} \) can be executed within a

\[
\text{time duration } t = 819 \text{ compared with that of } 8,879 \text{ when originally adopts no } \text{Havoc} \text{ (i.e., Neuzz and MTFuzz), their edge coverage performance can also be significantly improved compared with their original versions.}
\]

More interestingly, we can find that for most fuzzers, they can incur quite close edge coverage with the modified \( \text{Havoc} \) as 16.3 hours, while their project-wise performance can be quite close as well, e.g., around 71K in project \texttt{readelf} and 38K in project \texttt{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 with our new strategy is much longer with our new hybrid strategy, and thus results in rather strong edge coverage. On the other hand, the execution time of \( \text{Havoc} \) for the other fuzzers turns to be much longer with our new hybrid strategy, and thus results in a significant performance gain. Note that for the fuzzers which originally adopts no \( \text{Havoc} \) (i.e., Neuzz and MTFuzz), their edge coverage performance can also be significantly improved compared with their original versions.

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

<table>
<thead>
<tr>
<th>Programs</th>
<th>readelf</th>
<th>QSYM</th>
<th>AFL</th>
<th>AFL++</th>
<th>FairFuzz</th>
<th>MOPT</th>
<th>Neuzz</th>
<th>MTFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>37,401</td>
<td>41,097</td>
<td>13,748</td>
<td>36,775</td>
<td>13,473</td>
<td>35,904</td>
<td>24,204</td>
<td>35,802</td>
</tr>
<tr>
<td>New</td>
<td>49,269</td>
<td>46,538</td>
<td>18,189</td>
<td>45,869</td>
<td>15,986</td>
<td>46,379</td>
<td>28,174</td>
<td>45,004</td>
</tr>
<tr>
<td>Average</td>
<td>43,482</td>
<td>45,804</td>
<td>14,972</td>
<td>40,581</td>
<td>5,294</td>
<td>41,178</td>
<td>18,457</td>
<td>40,407</td>
</tr>
</tbody>
</table>

Table 5: Edge coverage results of fuzzers with modified \( \text{Havoc} \)

<table>
<thead>
<tr>
<th>Programs</th>
<th>readelf</th>
<th>QSYM</th>
<th>AFL</th>
<th>AFL++</th>
<th>FairFuzz</th>
<th>MOPT</th>
<th>Neuzz</th>
<th>MTFuzz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig</td>
<td>37,401</td>
<td>41,097</td>
<td>13,748</td>
<td>36,775</td>
<td>13,473</td>
<td>35,904</td>
<td>24,204</td>
<td>35,802</td>
</tr>
<tr>
<td>New</td>
<td>49,269</td>
<td>46,538</td>
<td>18,189</td>
<td>45,869</td>
<td>15,986</td>
<td>46,379</td>
<td>28,174</td>
<td>45,004</td>
</tr>
<tr>
<td>Average</td>
<td>43,482</td>
<td>45,804</td>
<td>14,972</td>
<td>40,581</td>
<td>5,294</td>
<td>41,178</td>
<td>18,457</td>
<td>40,407</td>
</tr>
</tbody>
</table>

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. Note that such solution can be quite consistent with a single-process fuzzer in terms of CPU resource consumption. 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 \( \text{Havoc} \) by \texttt{socket}. After completing the whole seed transmission, \( \text{Havoc} \) is executed for time duration \( t \) as well while the execution of the original fuzzing strategy is paused. Note that instead of dynamically setting a mutation count for controlling its execution as the default \( \text{Havoc} \), our modified \( \text{Havoc} \) iteratively generates new seeds based on the updated collection of the “interesting” seeds within time duration \( t \). Similarly after executing \( \text{Havoc} \), we transmit the file names of its generated seeds to the original fuzzing strategy of the fuzzer via \texttt{socket} for further seed generations. Such process is iterated until hitting the total time budget.

**Evaluation.** We first evaluate \( \text{Havoc} \) by setting the iterative time duration \( t \) of the major fuzzing strategy/\( \text{Havoc} \) as 1 hour (i.e., executing them for 1 hour respectively under each iteration). As a result, for each fuzzer, its modified \( \text{Havoc} \) 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 \( \text{Havoc} \) where “Orig” represents the original fuzzers with their default implementation and “New” represents the associated fuzzer integrated with the modified \( \text{Havoc} \). Note that since such setup does not fit for the essential mechanisms of the pure \( \text{Havoc} \) and QSYM which execute \( \text{Havoc} \) for the whole execution, i.e., 24 hours, we retain their results of the previous evaluations in Table 5 simply for illustration and comparison.

We can observe that while MOPT with the modified \( \text{Havoc} \) can incur quite close edge coverage compared with its default \( \text{Havoc} \) integration as in Table 3, the rest fuzzers with the modified \( \text{Havoc} \) can achieve much higher edge coverage compared with their original versions, e.g., 1.8X for AFL. Such result can further validate our finding 3. Specifically, the original MOPT can already incur quite long execution time for \( \text{Havoc} \) by default, i.e., 16.3 hours, and thus can result in rather strong edge coverage. On the other hand, the execution time of \( \text{Havoc} \) for the other fuzzers turns to be much longer with our new hybrid strategy, and thus results in a significant performance gain. Note that for the fuzzers which originally adopts no \( \text{Havoc} \) (i.e., Neuzz and MTFuzz), their edge coverage performance can also be significantly improved compared with their original versions.
We further attempt to investigate how changing 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 time duration $t$, i.e., under sufficient total execution time, adapting the execution time of Havoc under each iteration results in rather limited impact on the edge coverage of the associated fuzzer.

While the previous findings 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 [44] 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 5 presents the evaluation results of seed dissimilarity between the pure Havoc and the other fuzzers (with the modified Havoc) on average, ranging from 0.134 to 0.256. Such result indicates that applying Havoc to different fuzzers can potentially explore quite common edges. Note that QSYM has the biggest Jaccard Distance although it executes Havoc for 24 hours. The main reasons can be that 1) QSYM invests more computing resource, i.e., leveraging three threads running in parallel, and 2) QSYM leverages concolic execution [14] that may explore different paths compared with fuzzing. Furthermore, MTFuzz and Neuzz also have large Jaccard Distance mainly because they further use neural networks to guide the fuzzing process.

Finding 5: As long as the total execution time of Havoc is fixed, how to adapt its iterative execution can have limited impact on the edge coverage performance of the associated fuzzer.

Finding 6: Investing more computing resource in executing Havoc can potentially reduce its execution time for approaching the performance bound, but may not be cost-effective.
we append Havoc without appending Havoc. In this paper, we follow prior work [6, 7, 9, 19, 20] to identify unique crashes only if they increase edge coverage. Note that in this paper, all of the crashes are explored by uniform selection out of a total of 15 mutators, we first uniformly select unit mutators, and then randomly select their inclusive mutators under the given stochastic execution or neural networks can better complement Havoc.

To begin with, it is essential to identify unique crashes since it is likely that many crashes are caused by the same program vulnerability. In this paper, we follow prior work [6, 7, 9, 19, 20, 25, 52] to identify the unique crashes only if they increase edge coverage. Note that in this paper, all of the crashes are explored by all of our previous evaluations. While a crash can only be reported once among all the fuzzing strategies (including Havoc) within a fuzzer, it can be possibly explored by different fuzzers. We then divide crashes into two sets, i.e., the ones explored by the involved Havoc mechanisms 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 fuzzing strategies, e.g., the constraint-solving-based mutations in QSYM and the gradient-driven mutations in Neuzz. Note that we exposed 69 unique crashes which have been fixed in the latest versions of their associated projects [3–5, 13]. We also report the rest unknown crashes (i.e., they can be exposed in the latest version) to the corresponding developers [2, 27]. The detailed bug report can be found in our GitHub page [32]. 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 indicate that applying Havoc can not only successfully advance program vulnerability exposure, but also potentially dominate the vulnerability exposure on certain projects.

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 the same program vulnerability. In this paper, we follow prior work [6, 7, 9, 19, 20, 25, 52] to identify the unique crashes only if they increase edge coverage. Note that in this paper, all of the crashes are explored by all of our previous evaluations. While a crash can only be reported once among all the fuzzing strategies (including Havoc) within a fuzzer, it can be possibly explored by different fuzzers. We then divide crashes into two sets, i.e., the ones explored by the involved Havoc mechanisms 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 fuzzing strategies, e.g., the constraint-solving-based mutations in QSYM and the gradient-driven mutations in Neuzz. Note that we exposed 69 unique crashes which have been fixed in the latest versions of their associated projects [3–5, 13]. We also report the rest unknown crashes (i.e., they can be exposed in the latest version) to the corresponding developers [2, 27]. The detailed bug report can be found in our GitHub page [32]. 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 indicate that applying Havoc can not only successfully advance program vulnerability exposure, but also potentially dominate the vulnerability exposure on certain projects.

4 ENHANCING HAVOC

So far, our presented powerful performance of Havoc is simply caused by modifying its setups, including its execution time and integration modes with fuzzers. In this section, we attempt to investigate whether the power of Havoc can be further boosted. 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 its performance accordingly.

4.1 Performance Impact of the Mutator Stacking Mechanism

Note that as a simplified mutation strategy, the mutator stacking mechanism contains two 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 of stacking size. To this end, instead of randomly determining stacking size for mutating seeds at runtime of Havoc originally, we implement Havoc under a fixed stacking size for all its mutations. Figure 6 presents our evaluation results of the edge coverage ratio results in terms of 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 all the possible fixed stacking size over the total explored edge number of all the fixed stacking sizes. 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, djpeo, and mut001 respectively. Such results suggest that it is essential to adapt the stacking size setup for different projects to optimize their respective edge coverage. 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 7 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 by either chunk...
We can observe that overall, the distribution of the edge coverage wise selections on stacking size. The Framework of Algorithm 1 (Havoc) to 94.53% (MAB refers to Multi-Armed Bandit), 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 [45] which typically refers to allocating limited resources to alternative choices (i.e., stacking size and mutator selections for this problem) to maximize their expected gain (i.e., edge coverage for this problem). 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 8. 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, HavocMAB generates a mutant via the selected mutators and executes it on the program under test for obtaining environmental feedback for further executions.

<table>
<thead>
<tr>
<th>Algorithm 1 The Framework of HavocMAB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: seed</td>
</tr>
<tr>
<td><strong>Output</strong>: newseed</td>
</tr>
<tr>
<td>1: function MULTI_ARMEED_UCB_SELECTION</td>
</tr>
<tr>
<td>2: newseed = seed</td>
</tr>
<tr>
<td>3: stacksize ← selectStackArm()</td>
</tr>
<tr>
<td>4: mutatorType ← selectMutatorTypeArm(stacksize)</td>
</tr>
<tr>
<td>5: for iteration in stacksize do</td>
</tr>
<tr>
<td>6: mutator ← randomSelectMutatorByType(mutatorType)</td>
</tr>
<tr>
<td>7: newseed ← generateNewSeed(mutator, newseed)</td>
</tr>
<tr>
<td>8: reward ← 0</td>
</tr>
<tr>
<td>9: if isInteresting(newseed) then</td>
</tr>
<tr>
<td>10: reward = 1</td>
</tr>
<tr>
<td>11: updateStackBandit(reward, stacksize)</td>
</tr>
<tr>
<td>12: updateMutatorTypeBandit(reward, stacksize, mutatorType)</td>
</tr>
<tr>
<td>13: return newseed</td>
</tr>
</tbody>
</table>

| 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., the edge coverage ratio of the unit mutators ranges from 18.39% (xml11rint) to 94.53% (tiff2bw). Such results suggest that it is also essential to adapt the selection of the mutator types for different projects to optimize their respective edge coverage performance. |

![Figure 6: Edge coverage for different fixed stack sizes](image)

![Figure 7: Edge coverage for unit mutators and chunk mutators](image)

### 4.2 Approach

Inspired by the evaluation results above, we attempt to propose solutions to enhance Havoc via dynamically adjusting the project-wise selections on stacking size and mutators. Also, 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 adopts the following principles. First, we only enable single thread/process, i.e., enhancing Havoc via only applying our specifically designed technique instead of leveraging more threads for more computing resource as found already. Second, our technique should be lightweight. In particular, when designing a technique for adjusting Havoc given the deterministic computing resource, ideally we aim for minimizing its overhead while maximizing the execution time for the Havoc mechanism itself.

In this paper, we propose a lightweight single-threaded technique HavocMAB (MAB refers to Multi-Armed Bandit), 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 [45] which typically refers to allocating limited resources to alternative choices (i.e., stacking size and mutator selections for this problem) to maximize their expected gain (i.e., edge coverage for this problem). 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 8. 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, HavocMAB generates a mutant via the selected mutators and executes it on the program under test for obtaining environmental feedback for further executions.

![Figure 8: The Framework of HavocMAB](image)

![Figure 9: The average edge coverage of HavocMAB over time](image)

We adopt the widely-used UCB1-Tuned [1] algorithm to solve our proposed multi-armed bandit problem. Equation 1 demonstrates...
To evaluate significantly better performance than pure Havoc, i.e., increasing the average edge coverage among all the benchmark projects by 11.1% (34,574 vs 31,126 explored edges). Moreover, we apply the Mann-Whitney U test [26] to illustrate the significance of Havoc\textsubscript{MAB}. The fact that the $p$-value of Havoc\textsubscript{MAB} comparing with Havoc in terms of the average edge coverage is 0.00507 indicates that Havoc\textsubscript{MAB} outperforms Havoc significantly ($p < 0.05$). Interestingly, although Havoc\textsubscript{MAB} only adopts one thread for execution, it can slightly outperform QSYM (which leverages three threads for execution) by 0.2% on average among all 5 runs with the STD of 108.55. It can also outperform QSYM for 4 out of 5 runs. On the other hand, executing Havoc\textsubscript{3MAB} can result in 9% edge coverage gain over QSYM (37,614 vs 34,495 explored edges) with a $p$-value of 0.01219. Such results altogether can demonstrate the strength of our proposed Havoc\textsubscript{MAB}.

Figure 10 presents the edge coverage trends of our studied approaches upon each benchmark for 24-hour execution. Overall, Havoc\textsubscript{MAB} outperforms pure Havoc in most of the benchmarks significantly. Moreover, Havoc\textsubscript{MAB} can outperform QSYM by at least 10% (60% more in tiff2bw) in terms of edge coverage on five projects while incurring rather close performance on the rest projects with a single thread except jhead. Meanwhile, Havoc\textsubscript{3MAB} can achieve the optimal edge coverage performance on eight benchmarks. Note that QSYM outperforms all other fuzzers in jhead (averagely 4,516 vs 1,063). This demonstrates that grey-box fuzzing strategies alone are ineffective for jhead while the effectiveness can be largely improved by concolic execution leveraged in QSYM. Based on this observation and Finding 7, we highly recommend future research to investigate more powerful techniques for combining Havoc, concolic execution, and learning-based fuzzing.

## 5 THREATS TO VALIDITY

**Threats to internal validity.** One threat to internal validity lies in the implementation of the studied fuzzers in our evaluation. To reduce this threat, we reused their original source code for our implementation and experimentation directly. Moreover, the first 4 authors manually reviewed all the code carefully to ensure its correctness and consistency.

**Threats to external validity.** The threats to external validity mainly lie in the subjects and benchmarks. To reduce the threats, we select 8 representative state-of-the-art fuzzers, including AFL-based, concolic-execution-based, and neural program-smoothing-based fuzzers. We also adopt 12 benchmark projects according to

\[
arm(t) = \arg \max_j \left( \hat{x}_j + \frac{\ln n_j}{n_j} \min \left( \frac{1}{4}, \sigma_j + \frac{2 \ln n_j}{n_j} \right) \right)
\]
their popularity, i.e., the most frequently used benchmarks by the original papers of our studied fuzzers. Another threat to external validity may lie in the randomness of the evaluation results. To reduce this threat, all the evaluation results are averaged upon five runs to reduce the impact of randomness.

**Threats to construct validity.** The threat to construct validity mainly lies in the main metric used in this paper, i.e., edge coverage, to reflect code coverage. To reduce this threat, while there can be various ways to measure edge coverage, we choose to follow many existing fuzzers [8, 20, 33, 34, 50] and leverage the AFL built-in tool named afl-showmap for collecting edge coverage. Furthermore, we have also evaluated fuzzing effectiveness in terms of the number of unique crashes.

6 RELATED WORK

**Fuzzing.** AFL [52] is one of the most popular fuzzers and has inspired many other recent fuzzers for different application domains. Fioraldi et al. [12] integrated multiple techniques, e.g., taint tracking, into the basic framework of AFL. Liang et al. [23] also introduced a path-aware taint analysis fuzzer to facilitate the efficiency of fuzzing. Böhme et al. [7] utilized a Markov chain model to allocate energy for seed selection. Peng et al. [30] 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 [15] boosted the efficacy of fuzzing under multiple processes and threads while Chen et al. [10] proposed a synchronization mechanism for integrating different fuzzers. Wang et al. [38] proposed SYZVEGAS to fuzz the kernel of operating systems by dynamically adjusting fuzzing strategies via reinforcement learning. Li et al. [21] introduced Steelix, which integrates lightweight static analysis to coverage-guide fuzzing. Wang et al. [39] 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 have also proposed a grammar-aware coverage-based greybox fuzzing approach, named Superion [40], to fuzz programs that process structured inputs. In more recent years, researchers have also proposed various techniques for fuzzing different types of software systems [29, 36, 42, 55]. Wu et al. [48, 49] proposed to detect CUDA synchronization bugs via fuzzing and repair them automatically. Zhang et al. [54] proposed DeepRoad to generate images to fuzz image-based driving systems. Zhou et al. [56] generated realistic and continuous images to fuzz such systems. In this paper, we propose a technique to dynamically adjust mutation selections for Havoc and result in strong edge coverage performance.

**Studies on Fuzzing/Testing.** Shen et al. [35] investigated different bugs on different deep learning compilers. Metzman et al. [28] introduced a platform for developers and researchers to evaluate different fuzzers. Although they studied Havoc associated with fuzzers, they did not evaluate it independently. Klees et al. [19] surveyed the recent research literature and assessed the experimental evaluations to illustrate the essential experimental setup for reliable experiments for fuzzing. We actually follow the instruction of this work to construct our initial seed corpus. Furthermore, Herrera et al. [16] systematically investigated and evaluated how seed selection affects the performance of a fuzzer to expose vulnerabilities in real-world systems. Many researchers studied the rationales behind fuzzing approaches. Wu et al. [47] empirically evaluated the neural program-smoothing-based fuzzers and improved them by proposing lightweight learning-based mutation strategies. Liang et al. [22] presented the main obstacles and corresponding typical solutions for fuzzing. Tonder et al. [37] presented a technique to map crashing inputs to unique bugs using program transformation. In this paper, we conduct the first extensive study on Havoc to demonstrate that Havoc is a powerful fuzzer, and have also shown that it is possible to further advance Havoc.

7 CONCLUSION

In this paper, we investigate the impact and design 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 already achieve superior edge coverage and vulnerability detection compared with other fuzzers. Moreover, integrating Havoc to a fuzzer or extending total execution time for Havoc can also increase the edge coverage significantly. The performance gap among different fuzzers can also be considerably reduced by appending Havoc. At last, we also design a lightweight approach to further boost Havoc by dynamically adjusting its mutation strategy.

8 ACKNOWLEDGEMENT

This work is partially supported by the National Natural Science Foundation of China (Grant No. 61902169), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), and Shenzhen Peacock Plan (Grant No. KQTD201611251455531). This work is also partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group.

REFERENCES

IEEE Symposium on Security and Privacy (SP). IEEE, 1580–1596.


