MOCK: Optimizing Kernel Fuzzing Mutation with Context-aware Dependency

Jiacheng Xu†, Xuhong Zhang†, ✉, Shouling Ji†, Yuan Tian‡,
Binbin Zhao*, Qinying Wang†, Peng Cheng†, ✉, and Jiming Chen†
†Zhejiang University, ‡University of California, Los Angeles, *Georgia Institute of Technology
{stitch, zhangxuhong, sji}@zju.edu.cn, yuant@ucla.edu, binbin.zhao@gatech.edu, {wangqinying, lunarheart, cjm}@zju.edu.cn,

***Abstract*—Kernels are at the heart of modern operating systems, whereas their development comes with vulnerabilities. Coverage-guided fuzzing has proven to be a promising software testing technique. When applying fuzzing to kernels, the salient aspect of it is that the input is a sequence of system calls (syscalls). As kernels are complex and stateful, specific sequences of syscalls are required to build up necessary states to trigger code deep in the kernels. However, the syscall sequences generated by existing fuzzers fall short in maintaining states to sufficiently cover deep code in the kernels where vulnerabilities favor residing.**

In this paper, we present a practical and effective kernel fuzzing framework, called MOCK, which is capable of learning the contextual dependencies in syscall sequences and then generating context-aware syscall sequences. To conform to the statefulness when fuzzing kernel, MOCK adaptively mutates syscall sequences in line with the calling context. MOCK integrates the context-aware dependency with (1) a customized language model-guided dependency learning algorithm, (2) a context-aware syscall sequence mutation algorithm, and (3) an adaptive task scheduling strategy to balance exploration and exploitation. Our evaluation shows that MOCK performs effectively in achieving branch coverage (up to 32% coverage growth), producing high-quality input (50% more interrelated sequences), and discovering bugs (15% more unique crashes) than the state-of-the-art kernel fuzzers. Various setups including initial seeds and a pre-trained model further boost MOCK's performance. Additionally, MOCK also discovers 15 unique bugs in the most recent Linux kernels, including two CVEs.

I. INTRODUCTION

In the modern world of computing, the operating system kernel plays a crucial role in the overall security of a computer system. It is responsible for managing and coordinating various hardware and software resources of a computer and providing a secure and reliable platform for applications. Due to a large code base and complicated architectures, kernels have been plagued by vulnerabilities. For example, 288 vulnerabilities related to Linux kernel, with an average CVSS [3] score of 6.5 out of ten, were reported in 2022, and an average of 196 vulnerabilities, with an average CVSS score of 6.3, have been discovered per year over the past five years [46]. Vulnerabilities in the kernel can allow attackers to access sensitive information, disrupt normal functioning, or compromise the entire system [4], [21].

✉Xuhong Zhang and Peng Cheng are the corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23131
www.ndss-symposium.org

Security researchers are dedicated to discovering vulnerabilities in the kernel by utilizing both static and dynamic approaches [23], [17], [56], [7], [30], [47]. Fuzzing [54], [28], [12] has been an effective and preferable testing technique for vulnerability discovery, and has proven its capability for software assurance by finding thousands of bugs in real-world complex systems [43]. It repeatedly feeds generated inputs into programs and discovers complex vulnerabilities with the help of programs' feedback and various sanitizers. In the field of kernel fuzzing, researchers leverage the syscalls interfaces as entry points, which are naturally provided by the kernel to handle communication between its various components and userspace applications. The test cases consist of syscall sequences which include syscall functions and parameters. In this manner, Google's Syzkaller [14], the state-of-the-art kernel fuzzer, has reported more than 3,900 bugs in the Linux upstream kernel over the past five years [16].

Despite the significant progress, kernel fuzzing has always faced challenges due to the stateful communication with the kernel. This is because there are explicit or implicit relations, referred to as dependencies, between syscalls. The execution of the preceding syscalls serves as the context for the subsequent syscalls, which impacts the subsequent syscalls' executing paths. It is, therefore, essential to construct syscalls in specific sequences to build up the required states of an execution path. Hence, the order in which syscalls are organized is crucial for a fuzzer to effectively find bugs deep in the kernel. Improper syscall selection for a given syscall sequence context will finally result in an ineffective test case, which will likely be rejected by the kernel or behave equivalently to the preceding context. A context-aware syscall selection approach can prominently enhance the input quality, reduce the search space and improve the efficiency of kernel fuzzers.

Existing kernel fuzzing solutions have worked to address the challenge of statefulness but still have many limitations in synthesizing stateful syscall sequences. The existing approaches [37], [56], [50], [48], [14] are designed to synthesize new test cases on a point-to-point basis, i.e., the selection of the next syscall is solely based on the previous one. For example, Syzkaller's choice table [14] and HEALER' relation table [48] ignore the subtle semantics or states encoded in the syscall sequences. Such a context-free design may work well for the low-hanging fruits with simple semantics but hinders fuzzers from generating complex, in-depth syscall sequences and exploring deep space in the kernel. Therefore, a context-aware design for generating syscall sequences is desired. Towards this, the first problem to address is how to prepare the corpus from which to learn the context-aware dependencies. Directly using the existing syscall trace datasets [2] may bring much noise, as the syscall sequences in the trace contain many loosely related syscalls, making it hard to extract the genuine context for a dependency. Limited by the availability of a golden corpus, the learned dependencies may be incomplete and even contain errors. Therefore, the second problem we need to address is the balance between the exploitation of the learned dependencies and the exploration of the existing strategies.

To address the aforementioned problems, we propose a novel approach to model the context-aware syscall dependencies as conditional probabilities and implement a prototype named MOCK. Against the first problem, we aggregate the syscall sequences that trigger new coverage in the fuzzing campaign and minimize them to extract the genuine contexts consisting of closely related syscalls. The minimized sequences embody the influence dependencies of syscalls, serving as a high-quality training set. We design a neural network language model to capture the dependencies that comprise the calling contexts efficiently. As the corpus accumulates and collects adequate training samples, MOCK enables the model training to learn the context-aware dependencies from the training set regularly and utilizes the learned dependencies to synthesize new test cases in a context-aware manner. In order to settle the second problem, we model the syscall sequence mutation as a Multi-Armed Bandit problem to balance exploitation and exploration. The various mutation operations are seen as bandits, and the coverage feedback acts as rewards. The context-aware dependencies are dedicated to examine the deeper part of the kernel while the context-free, as well as random operations, help increase the diversity of mutation.

We measure MOCK's performance on the recent versions of the Linux kernel. Our evaluation shows that MOCK's context-aware dependencies perform more effectively in achieving branch coverage with an average of 7% coverage growth than the state-of-the-art kernel fuzzers HEALER, Syzkaller and SyzVegas. It is also capable of producing 50% more interrelated high-quality input. In addition, we evaluate MOCK with various setups including initial seeds and a pre-trained model, which proves to help reduce warmup time and accelerate fuzzing campaign. Compared to HEALER, Syzkaller, and SyzVegas, MOCK also finds 15% more unique vulnerabilities, especially the ones whose triggerings require a long, interrelated syscall sequence. Moreover, MOCK has newly found 15 unique vulnerabilities in the most recent Linux kernels, four of which are confirmed by Linux developers.

We summarize our main contributions as follows:

- We identify the syscall relations as the context-aware dependencies in conformance to the kernel's statefulness, which is largely neglected by existing kernel fuzzers. The novel context-aware dependency schema for syscalls contributes to synthesizing stateful syscall sequences and enhancing the quality of test cases.
- We develop the first system incorporating context-aware syscall dependency for kernel fuzzing. Specifically, MOCK optimizes kernel fuzzing by (1) designing a neural network language model to learn the contextual syscall dependency from runtime syscall sequences dynamically, (2) mutating syscall sequences according to the contextual dependency, and (3) adaptively scheduling the context-aware mutation task for diversity assurance. To facilitate future kernel security research, we will open-source MOCK at *https://github.com/m0ck1ng/mock*.
- We evaluate MOCK in comparison with HEALER [48], Syzkaller [14] and SyzVegas [50] on the Linux kernels. The results indicate that MOCK outperforms its counterparts impressively in terms of branch coverage, interrelated sequences synthesis, and discovery of vulnerabilities. Additionally, our further analysis reveals that our designs are valuable contributions that can be seamlessly integrated into the existing fuzzer with minimal overhead.
- We have discovered 15 unique bugs in the recent Linux kernels and reported all the discovered bugs, of which four are confirmed by Linux developers during the responsible disclosure process. Up to now, we have received two CVEs that lead to high severity, allowing attackers to escalate privilege.

II. BACKGROUND

A. Kernel Fuzzing

Syscalls are a set of interfaces for interacting with the kernel and the behavior of each syscall heavily depends on the kernel states created by the previous syscalls. Hence, programs are required to invoke syscalls following dependencies to function correctly. Since the sequence of syscalls is highly structured, it is non-trivial to achieve good code coverage with random combinations of syscalls.

Google's Syzkaller is now the state-of-the-art fuzzer for kernel fuzzing [14]. To generate diverse and interesting test cases that cover various corner cases, Syzkaller develops a set of declarative templates, called `Syzlang`, involving functions, arguments, and types to describe the syntax and structure of syscalls. Based on `Syzlang`, Syzkaller establishes a choice table to guide the syscall sequence generation. The choice table records the probability value that a syscall should be invoked before another syscall. Specifically, the choice table is comprised of static and dynamic priority. The static priority considers the direction of arguments. For given syscalls, a higher priority will be assigned if one syscall produces a resource and the other consumers it. Syzkaller leverages an intuitive analysis algorithm based on initial seeds to calculate the dynamic priority. The core idea of the algorithm is that if two syscalls always come adjacently in seeds, they are more likely to have dependencies on each other. Static and dynamic priority are finally combined together on weights as the choice table.

Inspired by Syzkaller, many works strive to improve kernel fuzzing from various perspectives. By statically analyzing syscall traces and global variables, MoonShine [37] captures the explicit and implicit dependencies among syscalls. It adopts a seed distillation algorithm relying upon the captured dependencies to prepare high-quality seeds for Syzkaller. SyzVegas [50] dynamically and automatically adapts task selection and seed selection in Syzkaller with EXP3 algorithm, to improve coverage performance. In contrast to previous works, HEALER [48] dedicates to deduce dependencies during the fuzzing campaign. HEALER defines a binary influence relation to describe the dependencies of syscalls, in which a syscall's execution can affect the execution path of another. Relation learning is divided into static and dynamic learning routines. The static routine recognizes explicit relations if a syscall's output meets another's input. The dynamic routine aims at finding implicit relations not expressed by syscall templates. When a test case triggers new coverage, HEALER removes syscalls in the test case individually, re-executes it and monitors whether the new covered lines still exist. Syscalls that do not contribute to new coverage will be discarded in the minimization. Finally, HEALER determines the last pair of syscalls in the minimized test case as *related*. The relation table, a two-dimensional table, preserves the influence relations for all the syscalls. The value of R_{ij} in the table $R_{n \times n}$ is 1 if syscall C_i affects C_j's execution, whereas 0 represents the opposite.

B. Language Model

A language model is basically a statistical model that determines the probability of a given sequence of words, which takes center stage in Natural Language Processing (NLP). Language models are utilized in a variety of tasks, including machine translation, text generation, and sentiment analysis. Generally, these models are built by analyzing the likelihood of word sequences in a training set and applying that knowledge to new, similar contexts. An N-gram language model is constructed by calculating the probabilities that the N-gram's last word follows the preceding N-1 words in history. N-gram works based on Markov assumption that the probability of a word depends only on the previous N words. However, this approach lacks long-range dependency and cannot cope with non-occurring grams, which are named *context* and *sparsity* problems. These shortcomings of N-gram models lead to poor prediction in real-world settings.

Neural network language models (NNLM) ease the sparsity problem by the way that encodes each word. Word embeddings create a sized vector as a feature vector, which incorporates the semantic information of each word. The NNLM was first proposed by Bengio *et al.* [8] and implemented as a feed-forward neural network (FNN) model. Word2Vec [35] further ported Recurrent Neural Network as a substitution for FNN to achieve better performance. It includes two architectures: continuous bag-of-words (CBOW) and skip-gram. A CBOW model is trained to predict the word in the middle based on the context, while the skip-gram does the opposite. In practice, the NNLM first establishes a vocabulary *V* containing all words from the corpus and maps each word in *V* into a feature vector. For a given context, a concatenation of feature vectors, the model is designed to output a conditional probability distribution over *V* for the next word.

C. Multi-Armed Bandit Problem

The Multi-Armed Bandit (MAB) problem [49] is a classical decision problem that exemplifies the *exploration-exploitation* tradeoff dilemma. In the problem, a gambler at a row of slot machines decides which machines to play and how many times to play each machine. The objective of the gambler is to maximize the expected gain. The MAB problem well models various decisions (e.g., seeds selection, mutation strategy selection) in coverage-guided fuzzing settings in which gains correspond to code coverage. There are several algorithms designed for the MAB problem. $\epsilon - Greedy$ is a simple method that randomly selects an action at the probability ϵ, and selects the one with the highest gain at the probability $1-\epsilon$ [52]. *Thompson Sampling* selects actions based on a prior distribution and posterior distribution decided by past observations [41]. *Upper Confidence Bound* (UCB) [6] algorithm is one of the most popular and impressive bandit algorithms. UCB is a deterministic algorithm while *Thompson Sampling* is a heuristic. Specifically, the algorithm constructs a confidence boundary to measure each action's reward on each round of exploration and play the one with the highest one on each round of exploitation.

The historical knowledge could be effective but an over-reliance on it may compromise the fuzzing diversity and the quality of the context-aware dependencies. We leverage UCB-1, a variant of UCB algorithm, to cope with this *exploration-exploitation* dilemma in fuzzing due to its good performance and low overhead.

III. MOTIVATION

An effective kernel fuzzer requires an in-depth understanding of the interfaces of the target kernel and the syscall dependencies for fuzzing. The dependencies facilitate the syscall selection in test case generation and mutation, and play a significant role in improving the quality of test cases and accelerating the fuzzing process. With a specific sequence of syscalls, a test case builds up a context in which kernel states are properly set for triggering deep code paths. While a fuzzer may pick low-hanging fruits with random combinations of syscalls, deep vulnerabilities are likely to be missed due to their subtle dependencies on complex states. For instance, Listing 1 shows a simplified proof-of-concept of a real bug discovered by MOCK. To trigger the bug, a syscall combination in a particular order is required and complicated states are supposed to be established: a fuzzer first creates a socket with the correct type, associates the socket with its local address, then connects it with a remote address; importantly, the fuzzer invokes `setsockopt` syscall to carefully configure the socket with specific level, option name as well as other attributes; finally, the fuzzer sends the socket to the `EVDEV` interface. MOCK can uniquely find the bug by considering the contextual information and giving more priority to `setsockopt` syscall, i.e., calling `setsockopt` twice.

Listing 1: A simplified proof-of-concept of a kernel bug found by MOCK. A syscall sequence in a particular order is required to trigger the bug.

```
socket$INET
bind$INET
connect$INET
setsockopt$INET
setsockopt$INET
write$EVDEV
```

Fuzzing sophisticated stateful software like kernel remains a big challenge. One of the difficulties is synthesizing stateful syscall sequences to cover deep code paths efficiently. Syzkaller and HEALER have shown promising results in using choice and relation tables, respectively, to guide mutation.

However, their effectiveness is hindered when dealing with interrelated sequences and exploring deeper state spaces in the kernel due to their reliance on context-free dependencies. As a result, these approaches struggle to adequately address the challenge of statefulness in kernel fuzzing. We find that either the choice table or the relation table is unequal to the role and has difficulty generating complicated syscall sequences. To illustrate this point, we conduct the following case study to present the constitution of the input produced during the fuzzing campaign. We use HEALER and Syzkaller, considered the most cutting-edge kernel fuzzers, to fuzz a Linux kernel for 24 hours. We collect the generated syscall sequences that trigger new coverage. The distribution of the generated syscall sequence length is illustrated in Figure 1. The distribution reveals that the performance of Syzkaller and HEALER notably declines when they generate more interrelated and longer syscall sequences. The syscall sequences whose lengths are more than three merely account for 8% of the corpus.

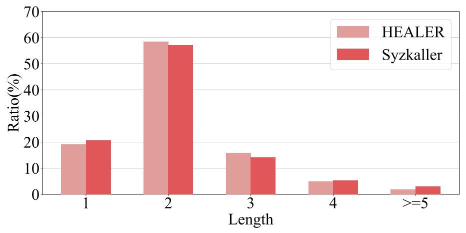


Fig. 1: The distribution of length of the syscall sequences generated by the fuzzers with the context-free dependencies.

The above results indicate that existing fuzzers are insufficient to produce complicated input and effectively explore the kernel space for vulnerabilities deep in states. We infer the reasons as follows:

1) Context-free Dependency. Existing kernel fuzzers model the interaction of syscalls as context-free dependencies. They develop the dependencies on a point-to-point basis, in which the dependencies work only in a range of two syscalls, whereas dependencies are supposed to be conditional probabilities and dynamically adjusted under different contexts. Such context-free dependencies modeling approaches also largely ignore the directions of dependencies. A dependency learned from the pair $\langle C_a, C_b \rangle$ is supposed to demonstrate that the syscall C_a has an impact on C_b (denoted as $C_a \rightarrow C_b$) and the reversed pair does the opposite. However, neither Syzkaller nor HEALER conforms to the rules. They consider the dependency to have a mutual effect on $\langle C_a \rightarrow C_b \rangle$ and $\langle C_b \rightarrow C_a \rangle$, and apply it to the prefixes $\langle C_a, \ \rangle$ and $\langle C_b, \ \rangle$ even though the latter makes no sense or even hinders the fuzzing efficiency.

2) Inadequate Utilization of Corpus. The runtime corpus consists of the syscall sequences collected in the fuzzing process, which reveals how the syscalls are organized and embody the syscall dependencies. Unfortunately, it has been vastly underutilized by existing kernel fuzzers. For example, Syzkaller does not use the runtime corpus when establishing the choice table. HEALER leverages the corpus but has certain drawbacks, leading to inadequate utilization. First, it studies the dependencies from the corpus in a coarse-grained manner. The learned dependencies are considered to be static and binary, i.e., related or not. However, despite the fact that the syscalls may be labeled as related, their relationship depends on the specific context and does not always take effect. Second, HEALER only captures the partial dependency information in the generated syscall sequences, i.e., it only uses the last pair of the test case for relation inference. However, in the corpus, each syscall in a test case contributes to new coverage, and the test case maintains the necessary states to trigger code deep in the kernel. Therefore, HEALER grossly neglects the overall dependency information behind the preceding syscalls; in other words, it fails to capture long-distance dependencies.

IV. Design of Mock

In this section, we present the design detail of Mock. Figure 2 illustrates Mock's overall design and workflow. First, Mock pre-processes `Syzlang` and the target kernel to infer the available syscalls and the static dependencies in ①. The available syscalls compose the syscall vocabulary, which is used for the context-aware dependency learning. Then, Mock picks up a seed from the corpus for mutation (② in Figure 2). Given a syscall sequence, the context-aware dependencies or the static dependencies guide the syscall selection to generate a new test case, and the scheduler determines the specific strategy with UCB-1 algorithm to balance the exploration and the exploitation in ③. After mutation, Mock executes the test case in ④ and checks whether new coverage or kernel crash is triggered. If a syscall sequence achieves new coverage, Mock minimizes it by re-executing it and analyzing the feedback for individual syscalls. As the fuzzing campaign advances, Mock collects more valuable test cases as the training set and updates the language model periodically to learn the latest context-aware dependencies (⑤ in Figure 2). The scheduler also records the execution and the performance of each strategy for better decisions.

A. *Pre-Processing*

The purpose of this phase is to prepare the groundwork for the fuzzing process and language model training. It conducts *static dependency* analysis and *syscall vocabulary* construction.

Static Dependency. The static dependencies play multiple roles in the fuzzing process. First, the static dependencies serve as a baseline for guiding the mutation of syscall sequences. A diverse and high-quality corpus of syscall sequences is a prerequisite of the context-aware dependencies, which means the context-aware dependencies cannot be enabled at the initial stage. Therefore, we use the static dependencies to ensure the normal operation of Mock before collecting a corpus with sufficient test cases and activating the context-aware dependencies. Second, the static dependencies also contribute to the diversity of the fuzzing campaign when combined with the context-aware dependencies. While context-aware dependency based on historical knowledge may prove effective for exploitation, an over-reliance on it could hinder further exploration of the input space. Specifically, Mock regards the static dependencies as an optional mutation strategy responsible for generating diverse syscall sequences and designs a customized task scheduling algorithm for balance, which will be further described in Section IV-C.

In the context of detecting the static dependencies, Mock utilizes the `Syzlang` descriptions and adopts HEALER's

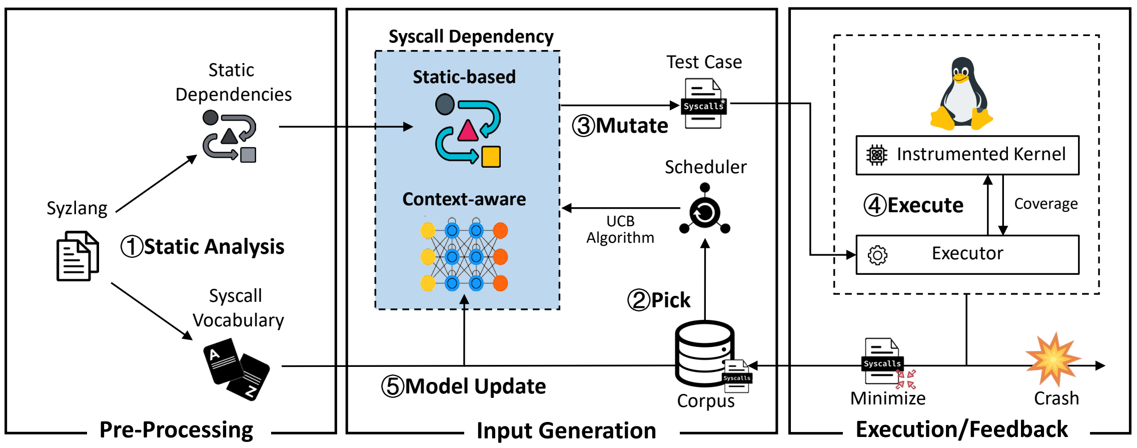


Fig. 2: The framework of MOCK.

strategy. The analysis focuses on the argument types and directions of the syscalls. The syscalls that output resources are identified as producers, while those that accept resources are regarded as consumers. Through this principle, the producers and consumers are understood to be interrelated. Specifically, given the syscall C_i and C_j, we identify that they share a static dependency following two guidelines: (1) C_i produces a return value that could be compatibly consumed by any parameter in C_j; (2) the parameters of C_i includes an outward direction pointer that could be compatibly consumed by any parameter in C_j.

Syscall Vocabulary. Vocabulary construction is a standard procedure for building a language model. In the kernel fuzzing setting, all the available syscalls compose the vocabulary to build a language model. With the syscall vocabulary, MOCK maps the syscall sequences into vectors in the training stage and the predicted vectors back to promising syscalls. Note that though `Syzlang` provides a set of information of syscalls, not every syscall in the `Syzlang` is available. The available syscalls may vary from kernel to kernel. Consequently, we perform a pre-run on the target kernel to check which syscalls could be invoked.

B. Context-aware Dependency Learning

As the core of MOCK, it models the syscall relations as the context-aware dependencies in the form of the language model and makes the most of the corpus collected from the fuzzing process. MOCK starts to train and utilize the language model when collecting sufficient samples. In this phase, we detail the process of context-aware dependency learning from the corpus.

Corpus Aggregation. The technique of coverage-guided fuzzing involves tracing the code coverage achieved by each input fed to a target program. The test case that achieves new coverage proves its potential and is collected as a seed in the corpus for further mutation. In kernel fuzzing, the test cases consist of various syscall sequences, which tell how the syscalls are organized and work jointly. As a result, the corpus reveals the syscall dependencies and the state transitions in addition to the code coverage. This historical knowledge could help fuzzers maintain necessary states and trigger code or vulnerabilities deep in the kernel. Consequently, MOCK chooses the collected corpus as a dataset for training. Unfortunately, when a syscall sequence triggers new coverage, it builds up the necessary states but does not necessarily get all the syscalls involved. There might be several noisy syscalls in the sequence that do not interact with others or contribute to new coverage. It is a necessity to filter out such individual working syscalls to maintain exact dependencies, which will have a significant impact on the model's generalization.

Algorithm 1: Test Case Minimization

```
Data: $s$: syscall sequence
      $C_{new}$: syscall with new coverage
      $pos$: position of $C_{new}$
      $cov$: coverage of $C_{new}$
Result: $S$: minimized syscall sequence
Function MINIMIZE($s$, $cov$, $idx$):
    $S \leftarrow s[0 : pos]$
    $i \leftarrow len(S) - 2$
    while $i \geq 0$ do
        $S' \leftarrow remove(S, i)$
        $cov' \leftarrow exec(S')$
        if $cov' == cov$ then
            $S \leftarrow S'$
        $i \leftarrow i - 1$
    return $S$
```

Once a test case achieves new coverage, it will be minimized before saving it into the corpus. MOCK adopts the same minimization method as HEALER and Syzkaller [15], but they implement it in different programming languages. Algorithm 1 shows the procedure of test case minimization. The algorithm takes in a syscall sequence s, position pos of the syscall C_{new} that achieves new coverage and new coverage cov. First, MOCK iterates over the sequence in a reversed order. For each syscall prior to C_{new}, the algorithm attempts to remove it individually and executes the reduced sequence. The algorithm removes a syscall permanently on the condition that the removal has no impact on the execution of C_{new}, which means the syscall does not share a dependency with C_{new}. Otherwise, the syscall engages with the kernel states and shall be reserved. In this way, MOCK minimizes test cases, which preserves the distinct dependencies without noise and provides a high-quality corpus for model training.

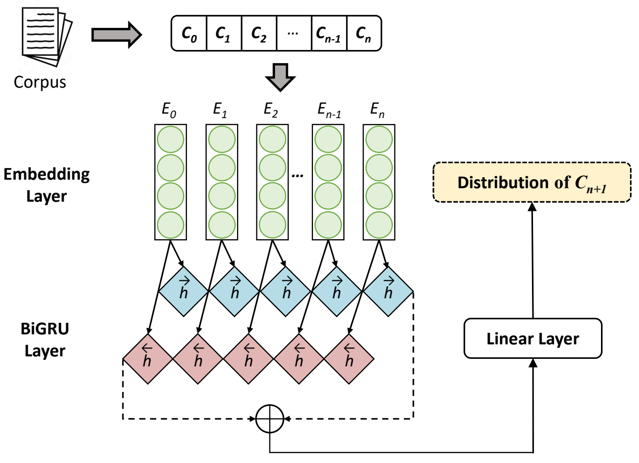

Corpus
C_0 C_1 C_2 ... C_{n-1} C_n
E_0 E_1 E_2 E_{n-1} E_n
Embedding Layer
BiGRU Layer
Distribution of C_{n+1}
Linear Layer

Fig. 3: The architecture of the language model.

Algorithm 2: Context-aware Mutation

```
Data: $s$: syscall sequence
      $pos$: insert position
      $M$: language model
Result: $C$: selected syscall
Function MUTATE($s$, $M$, $pos$):
    $ctx_f \leftarrow s[0 : pos]$  // the front context
    $d_f \leftarrow softmax(M(ctx_f))$
    $cand \leftarrow topK(d_f)$  // with probabilities

    if $pos$ ¡ $len(s)$ then
        $ctx_r \leftarrow reverse(s[pos : len(s)])$
        // the rear context
        $d_r \leftarrow softmax(M(ctx_r))$
        $cand_r \leftarrow topK(d_r)$  // with probabilities
        $cand \leftarrow merge(cand, cand_r)$

    $C \leftarrow random\_weighted(cand)$
    return $C$
```

Language Model. Mock uses the neural network language model to capture the context-aware dependencies of syscalls. Figure 3 depicts the architecture of the model. Our model is composed of one embedding layer, one Bidirectional Gate Recurrent Unit (BiGUR) [9], and one linear layer. Mock trains the language model using the corpus $(x, y) \in D$ as training data, which is collected from syscall sequences that trigger new coverage. Given a syscall context $x = [C_0, C_1, \ldots, C_n]$, the goal of the model is to output a conditional probability distribution for the next syscall. The detailed process is as follows.

When the model takes in a syscall sequence, it first converts the sequence into a numerical sequence in compliance with the syscall vocabulary. After passing the embedding layer, each syscall turns into a distributed vector representation named *embedding*. Thus the syscall sequence is represented in the form of a numerical matrix. The *embedding* vector is then fed to the BiGRU layer. At each time step, the BiGRU layer takes four inputs: (1) a forward hidden state $\overrightarrow{h}_{t-1}$ and (2) a backward hidden state $\overleftarrow{h}_{t-1}$ from the previous time step; and (3) the embedding of a given syscall. We take advantage of the BiGRU to predict the promising syscall based on the cumulative context of preceding syscalls. Finally, the output of the BiGRU layer is forwarded to the linear layer, and the layer yields a probability distribution $f(x)$ over the vocabulary. To meet the goal mentioned above, the language model is designed to minimize the following loss L over the training set $(x, y) \in D$:

$$
\begin{aligned}
g(x) &= softmax(f(x)) \\
l(g(x), y) &= -\sum_{i=1}^{n} y_i \log g(x)_i \\
L &= \frac{1}{|D|} \sum_{(x,y)\in D} l(g(x), y)
\end{aligned}
\tag{1}
$$

In this work, we utilize the CBOW model to tackle the syscall sequence generation task. This model combines surrounding words to predict the word located in the middle of a given sequence. There are front and rear surrounding words in the text, and so do the syscall sequences. To address this, we employ a bidirectional RNN, where the forward GRU handles the front context while the backward GRU handles the rear context. By utilizing this approach, Mock can improve the context-aware dependencies.

We update the language model on a regular basis in the fuzzing process. This is because with the state change of the kernel and the growth of the corpus, the latest dependencies can be learned and the existing dependencies can be refined. The corpus is split into a training set and a validation set in the updating phase to train the model and inspect its performance intermediately. After Mock finishes several epochs in training, the updating phase ends, and the model that achieves the best performance on the validation set will be employed.

C. Dependency-guided Mutation

In this phase, Mock leverages the learned context-aware dependencies to mutate a syscall sequence according to context. In addition, we design a novel scheduling strategy to prevent mutation from stereotypes raised by excessive reliance on historical knowledge.

Context-aware Mutation. Mock mutates a syscall sequence in a context-aware manner in order to intentionally trigger code paths on complex states. Kernel fuzzers generally possess various mutation strategies, such as sequence-level operations and parameter synthesis. The sequence-level operations involve inserting syscalls into an existing sequence and removing syscalls from a sequence, and the parameter synthesis produces concrete values according to `Syzlang`'s types. The greatest strength of Mock is that it optimizes the syscall selection approaches of existing kernel fuzzers by incorporating context-aware dependencies.

Mock adds a new syscall to a given syscall sequence for mutation. In the preparatory stage, Mock picks up a seed and randomly chooses the position for the syscall to be inserted. Given a seed syscall sequence and the target position, Mock leverages the context-aware dependencies to select a promising syscall in consideration of the context. Algorithm 2 outlines the process of mutating a syscall sequence with the model. Specifically, the algorithm first extracts the sub-sequence prior to the position as the front context, which is then fed to the model to generate an output vector. The vector is further transformed into a probability distribution of syscalls via `softmax` function, which is often used in machine learning to scale arbitrary values into probabilities. Based on the probability distribution, the `topK` function identifies the *top-k* suggestions (with their probabilities) as the candidates. MOCK adopts the same measures to the rear context, assuming that the insert position is not at the end of the sequence. The front context determines which syscalls are influenced by it, while the rear context decides which syscalls can influence it. It takes in the rear context in reverse order and yields another *top-k* suggestions with their probabilities. MOCK then merges the two suggestions into the overall weighted candidates by adding up the probabilities of the same syscall in the suggestions. Finally, MOCK randomly makes a syscall selection from the overall candidates according to their probabilities. After the syscall selection, MOCK synthesizes parameters for the newly added syscall according to parameter types and sends it to the kernel image for execution. In the procedure, no significant alteration is required when constructing and using the context-aware dependencies, which conveniently serve as an extension of existing fuzzers.

It is worth noting that the rear context is fed into the model in reverse order. As mentioned previously, the language model utilizes the BiGRU to better develop the context-aware dependencies where the backward GRU passes the hidden states to predict the next syscall from back to front (see Figure 3). Hence, MOCK conforms to the rule as well when mutating a syscall sequence and generates high-quality test cases. It is important to bear in mind that the context-aware dependencies we propose are definitely directional though it combines the bidirectional contexts to mutate a syscall sequence. Having known an influence relation $C_a \to C_b$, the fuzzers with the context-free dependencies neglect the relation's direction and equally apply it to the prefixes $\langle C_a, \ \rangle$ and $\langle \ , C_a \rangle$. MOCK overcomes the problem by taking the context into account. For instance, MOCK learns the dependencies from the syscall sequences [C_0, C_1, . . ., C_n] in the corpus (the sub-sequence [C_0, C_1, . . .] denoted as ctx). MOCK may apply the dependencies to a similar context $\langle ctx', \ \rangle$ whereas disregard of the context $\langle \ , ctx' \rangle$. So does it when the context is on the back.

Task Scheduling. MOCK aims to find bugs by building solid seeds which require a combination of interrelated syscalls to establish states in the kernel. These solid seeds are more difficult to produce by existing approaches and enable the exploration of deeper logic. To achieve this, we propose and learn the context-aware dependencies from the corpus. However, while the historical-knowledge-based dependencies prove effective, an over-reliance on it may undermine the diversity and lead the mutation candidates to a limited set. This will compromise the quality of the training set and the context-aware dependency. Consequently, the input mutation eventually may be limited to some fixed patterns, hindering further exploration of input space (shown in III). Hence, we model the fuzzing dilemma as a MAB problem and introduce the MAB algorithm to increase the diversity of mutation and corpus mainly. Additionally, the diversity also improves the chance of unpredictable syscalls for bug hitting, which is the secondary objective.

Algorithm 3: Task Scheduling

```
Data: $t_{explore}$: time budget for exploration
      $t_{exploit}$: time budget for exploitation
Function SCHEDULE(s, M, pos):
    $T \leftarrow$ [static-based, model-based]
    // different tasks for syscall selection
    do
        if stage = EXPLORE then
            $init\_weights(w)$
            $init\_zeros(N, R)$
            for $t = 0 \to t_{explore}$ do
                $i \leftarrow random\_weighted(w)$
                // selected task index
                $input \leftarrow insert\_call(T[i])$;
                $cov \leftarrow execute(input)$
                $N_i \leftarrow N_i + 1$
                $R_i \leftarrow R_i + is\_new(cov)$
            $stage \leftarrow$ EXPLOIT
        else if stage = EXPLOIT then
            $i_{best} \leftarrow UCB-1(N, R)$
            for $t = 0 \to t_{exploit}$ do
                $input\_call(T[i_{best}])$;
                $execute(input)$
            $stage \leftarrow$ EXPLORE
    while true
```

To address the above challenges, we design a task scheduling strategy based on the UCB-1 algorithm. We choose UCB-1 instead of EXP3 adopted by SyzVegas [50] for three reasons. First, the UCB-1 algorithm requires fewer computational resources, making it suitable for fuzzing where the decision-making process needs to be fast. Second, UCB-1 is a relatively simple algorithm, making it easier to grasp and apply in practice. The objective of coverage-guided fuzzing is to maximize the code coverage within the same time. Thus actions in MAB problems correspond to distinct mutation tasks, and gain corresponds to code coverage. The UCB-1 algorithm divides the fuzzing campaign into the exploration and exploitation stage: in the exploration stage, the algorithm estimates each action's confidence interval, which represents the uncertainty and combines the average award (exploitation term) with the upper confidence bound (exploration term) as the overall rewards; in the exploitation, only the action with the highest reward is selected. Moreover, the confidence interval gets narrower and more accurate with the attempts increasing. Given the number of executions N_i and the number of test cases R_i that trigger new coverage or crashes at time step t, we formally define the reward and the algorithm as follows:

$$\begin{gathered} G_i = \frac{R_i}{N_i} + c\sqrt{\frac{\ln t}{N_i}} \\ i_{best} = \arg\max_i G \end{gathered} \tag{2}$$

Algorithm 3 shows the workflow of task scheduling integrated with the fuzzer. The algorithm takes in the time budget $t_{explore}$ for the exploration stage and $t_{exploit}$ for the exploitation stage. After the model's employment, MOCK iteratively switches between the exploration and exploitation stage under the time budget until the fuzzing loop ends. In the exploration stage, MOCK first assigns a pre-defined selection weight for each task. It also resets the numbers of execution and that of the test cases which triggers new coverage or crashes as zero. After the initialization, MOCK picks up a syscall selection task i by the weights w randomly and applies it to a seed to generate a new input. The scheduler updates the correspondent execution time N_i when a task gets employed. The executor runs the new input and returns the code coverage information to the scheduler, which updates R_i if the input achieves new coverage. MOCK repeatedly follows the procedure before the time budget $t_{explore}$ is consumed and then changes to the other stage. From the estimated reward, the schedule determines the task with the best performance, and MOCK focuses on the task to generate effective input and trigger more code coverage within the time budget $t_{exploit}$. Then MOCK switches to the exploration stage again, and so the cycle continues.

V. IMPLEMENTATION

The context-aware dependency could be implemented as an extension module of existing kernel fuzzers. Our implementation of MOCK inherits `Syzlang` from Syzkaller and incorporates the model training, the model-guided mutation, and the task scheduler on top of the fuzzing engine of HEALER , which takes roughly 1,600 lines of code.

The model training module uses PyTorch 1.11.0 to implement a lightweight NNLM where each embedding has a dimension size of 64, and the BiGRU layer has a hidden state size of 128. The BiGRU also randomly sets units to zero with a frequency of 0.5 to help prevent overfitting. The Adam optimizer is adopted and its learning rate adaptively decays every ten training epochs. The module preserves the trained model in the form of TorchScript. In the fuzzing process, MOCK uses the first hour to collect the initial training set. Since then, it invokes the training module every two hours to dynamically adjust the dependencies.

The model-guided mutation module then loads the model and performs the mutation operation via binding code (e.g., tch-rs [34]), which provides some thin wrappers around the C++ PyTorch API (LibTorch [40]) in the underlying. When the model outputs a probability distribution, we sample candidates from it with *top-k* as 15. Due to a conflicting issue between PyTorch and LibTorch when they run in the same process, we deploy the training module in a paralleled routine with the main fuzzing loop in which the mutation module resides. The mutation module invokes the training module periodically via restful APIs and loads the model from the return value.

In the task scheduling module, each task is assigned with even weights in the exploration stage. We determine the budget ratio of exploration and exploitation by running various settings and choose the one with the most branch coverage, which is shown in Table IV.

VI. EVALUATION

In this section, we evaluate the following research questions:

• **RQ1:** How does MOCK perform in code coverage?

• **RQ2:** How effective is context-aware dependency compared to context-free dependency?

• **RQ3:** Do various setups (e.g. initial seeds, pre-trained models) reduce warmup time and boost fuzzing performance?

• **RQ4:** How does MOCK perform in vulnerability detection?

• **RQ5:** Can MOCK discover new vulnerabilities in real-world kernels?

• **RQ6:** How is the significance and overhead of key components in MOCK?

To be specific, we compare MOCK with the state-of-the-art fuzzers Syzkaller, HEALER and SyzVegas on the recent versions of Linux kernels. The evaluations are designed to measure the efficacy of context-aware dependency to build up kernel states to discover new vulnerabilities and reach execution paths. We also conduct experiments to demonstrate the advantage of context-aware dependency over context-free dependency by various metrics. In addition, the fuzzers start with various setups including initial seeds and a pre-trained model to verify if the settings help reduce the warmup time and improve the efficiency of fuzzing. Moreover, we conduct further analysis to evaluate the significance of key components in MOCK by ablation experiments and discuss the overhead of our designs.

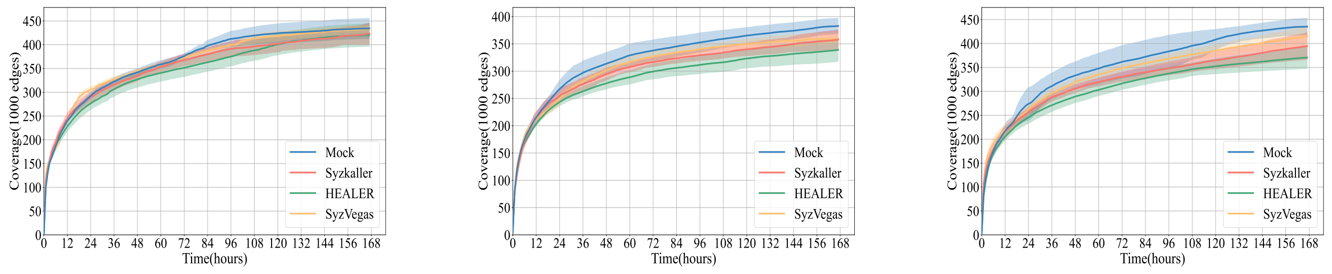
A. Experiment Setup

All the experiments are conducted on four machines, each of which runs 64-bit Ubuntu 16.04LTS with Intel Xeon E5-2650 v4 (2.2GHZ, 48cores) CPU and 256GB of main memory. To mitigate randomness, each set of experiments is repeated five times and each experiment has a time budget of 168 hours [24]. MOCK and other fuzzers use only CPU and the same virtual machine configuration (2 cores, 2 GB of memory) to ensure a fair comparison. We evaluate the fuzzers on three long-term Linux kernels: Linux-5.10, Linux-5.15, and Linux-5.19, which are widely deployed all over the world and heavily tested by many other works. All the kernels are compiled under the same configuration with the common features enabled (e.g., `KCOV` [45] and `KASAN` [13]). Unless stated otherwise, we employ Linux-5.10 as the target kernel and run the fuzzers without any initial seeds.

B. Coverage Performance

To evaluate MOCK's capability of exploring code execution paths with the same time budget, we inspect the fuzzing process on three versions of Linux kernel 5.4, 5.10, and 5.15, and record the branch coverage in comparison with Syzkaller, HEALER and SyzVegas.

Figure 4 shows the branch coverage growth achieved by each fuzzer. MOCK can achieve higher code coverage than HEALER, Syzkaller, and SyzVegas during the same period. Specifically, MOCK has a similar coverage performance compared to the other fuzzers at the beginning of fuzzing. There are several reasons for this. First, MOCK relies solely on the static dependencies and does not initiate the model in the setup time. Second, while the model is enabled after startup, the inadequate size of the training set also leads to the limited



(a) Linux-5.4 (b) Linux-5.10 (c) Linux-5.15

Fig. 4: The branch coverage growth of running MOCK, Syzkaller and HEALER on three versions of Linux kernel.

capacity of the model, and the model may not properly master the context-aware dependencies. Therefore, the task scheduler favors prioritizing the static dependencies in the early stage. Once the fuzzing campaign advances and the training set grows, we observe that the context-aware dependencies get to manifest their might, and MOCK's coverage gradually outperforms the other fuzzers. Finally, MOCK performs the best in respect of branch coverage. MOCK averagely achieves 12%, 6% and 3% branch coverage improvement in comparison with HEALER, Syzkaller, and SyzVegas, respectively[1]. The speed-up measures the speed of MOCK achieving the same branch coverage as the baseline. Meanwhile, MOCK performs more prompt in exploring the same amount of execution paths with a 2.23×, 1.58×, and 1.32× speed-up over HEALER, Syzkaller, and SyzVegas.

The above statistics indicate MOCK's efficacy in triggering more unique code branches with an average of 7% increase and a 1.71× speed-up. Under the guidance of the language model and the context-aware aspect, MOCK manages to grasp the subtle dependencies between syscalls and make syscall selections adapting to the calling context. In this way, MOCK could prevent pointless combinations, build up necessary states more efficiently and improve the quality of sequences, which accounts for the growth in code coverage.

C. Effectiveness of Context-aware Dependency

As described in Section IV-C, MOCK leverages the context-aware dependency to explore deep kernel space more effectively and efficiently. To evaluate whether the context-aware dependency can construct more interrelated inputs, we investigate the lengths of all the minimized syscall sequences generated by the fuzzers. Next, in order to show a more fine-grained effect of context-aware dependencies in selecting promising syscalls, we evaluate the number of test cases that cover new coverage and new states under various contexts.

Testcase Analysis. An effective fuzzer is designed to generate diverse as well as in-depth input to cover different corner cases. Fuzzers using context-free dependencies may reach various and shallow code execution paths with simple syscall combinations. However, they practically suffer from generating in-depth test cases that properly set up kernel states and trigger deep code logic. Therefore, we use the length of the syscall sequence as the metric to measure the test case's interrelation. The syscall sequences in the corpus are minimized and their syscalls all get engaged to maintain necessary states and finally perform an execution. The more kernel states are required, the longer a syscall sequence is, and it is increasingly difficult for a fuzzer to construct such a test case.

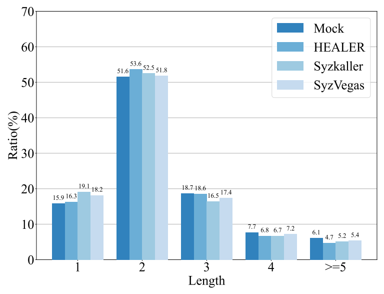
We collect and analyze the corpus from the four fuzzers. Figure 5 depicts the distribution of lengths of the minimized syscall sequences produced by the fuzzers. Syzkaller and HEALER concentrate on generating syscall sequences whose lengths are less than three while behaving poorly in constructing long sequences. MOCK by contrast significantly improves the ability to produce in-depth syscall sequences, especially for the sequences with lengths greater than or equal to five. As is shown in Figure 5, for syscall sequences no shorter than three, the execution times assigned by MOCK (33.5%) are 1.11, 1.17 and 1.12 times higher than that of HEALER (30.1%), Syzkaller (28.7%) and SyzVegas (30.0%); in particular, for in-depth syscall sequences greater or equal than five, the execution times assigned by MOCK (7.9%) is 1.61, 1.49 and 1.41 higher than that of HEALER (4.9%), Syzkaller (5.3%) and SyzVegas (5.6%). Under the guidance of context-aware dependencies, MOCK could improve the quality of test cases by producing more interrelated combinations. As a result, MOCK has a greater opportunity to explore deeper code space in the kernel with the same time budget, which is generally vulnerable and far from fully tested.

TABLE I: The performance of context-free and context-aware dependencies under various contexts.

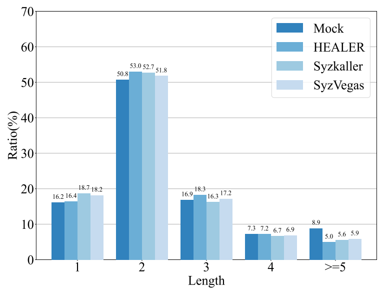
Dependency Model	**Context Size**				
	1	**2**	**3**	**4**	**>=5**
No. of test cases that trigger new coverage					
context-free	4,381	8,275	3,966	1,846	1,921
context-aware	5,488	10,199	4,906	2,308	2,374
Improvement (%)	25	25	24	25	24
No. of test cases that increase the length of syscall sequences					
context-free	604	701	183	68	84
context-aware	533	940	270	81	104
Improvement (%)	-13	34	48	19	24

Contextual Mutation Analysis. It is difficult for existing fuzzers to generate in-depth inputs because they capture relations of syscalls ignorant of the calling context. To overcome the shortcoming, MOCK adopts the context-aware dependencies to guide the mutation and generation procedure.

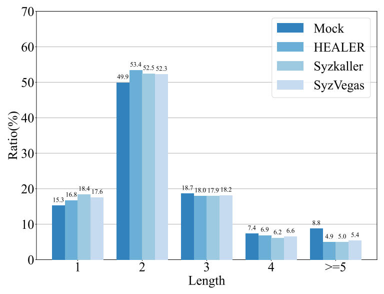
[1]Our evaluation contradicts findings in [48], which could be due to [1].



(a) Linux-5.4



(b) Linux-5.10



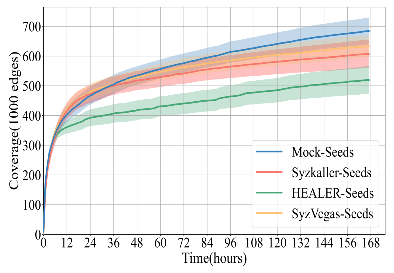
(c) Linux-5.15

Fig. 5: The length distribution of the corpus collected by MOCK, Syzkaller, and HEALER on three versions of Linux kernel.

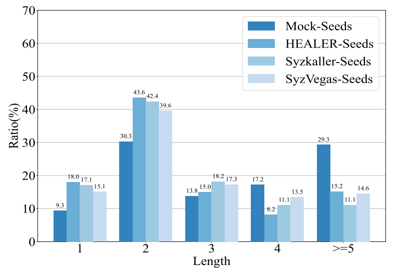
Hence, we conduct experiments to compare the ability of the context-free and context-aware dependencies to trigger new coverage and new states under different contexts and take a closer look at MOCK's superiority. We choose the relation table from HEALER as the representation of the context-free dependencies in comparison. To be specific, we count the number of executions of inserting new syscalls and inspect whether syscall sequences after insertion achieve new coverage or increase sequence length after minimization.

Table I presents a summary of the performance of context-free and context-aware dependencies across different contexts. The table indicates that the number of sequences that achieve new coverage through context-aware and context-free syscall selection is 25,275 and 20,389, respectively. Notably, we find that the context-aware dependencies generate an average of 1.24 times more syscall sequences than the context-free dependencies. Regarding length, the context-aware dependencies generate 1,771 inputs that can increase the lengths of the syscall sequences, which is comparable to the context-free dependencies (1,523). However, it is worth noting that the context-aware dependencies demonstrate better performance as the context size increases. Specifically, while they underperform the context-free dependencies when the context size is one, they exhibit a significant advantage over more extended contexts. For instance, they achieve a 34% improvement in increasing the lengths of the syscall sequences when the context size is two, and this improvement remains 30% for context sizes greater than or equal to three.

We conclude from the result that the context-aware dependencies (1) facilitate building up complicated states and exploring the deep kernel space, and (2) are marginally deficient in the face of a simple context while reaching more coverage by possibly allocating more energy to promising syscalls. Overall, this result definitely demonstrates the effectiveness of MOCK against existing kernel fuzzers in terms of tailoring the kernel context to make better syscall selections. The context-free dependencies that leverage the point-to-point approach may take effect with a single syscall context, but their abilities ultimately decline when coping with long dependencies and states. In contrast, MOCK employs the context-aware dependencies to capture the complicated dependencies and prevents them from getting stuck in the shallow input space. This is the reason why MOCK improves the code coverage as well as the bug findings.



(a) Branch coverage



(b) Length distribution of corpus

Fig. 6: The fuzzing performance of MOCK-Seeds, HEALER-Seeds, Syzkaller-Seeds, and SyzVegas-Seeds.

D. Various Setups

To effectively direct MOCK to learn the context-aware dependency, we introduce a novel strategy by using the fuzzing campaign to train the model. Hence, various settings including initial seeds and a pre-trained model, may help reduce the warmup time and accelerate fuzzing process. To evaluate how the various setups contribute to the efficiency of MOCK, we conduct experiments by (1) feeding the fuzzers with initial seeds and (2) integrating the pre-trained context dependence model with MOCK.

Fuzzing with Initial Seeds. The initial seeds may greatly influence the fuzzing performance. We prepare the seeds by crawling syz reproducers from fixed Linux Kernel upstream crashes in syzbot [16], following the guideline of [36]. To ensure a fair comparison,We then feed the fuzzers with the same initial seeds and denote them as MOCK-Seeds, HEALER-Seeds, Syzkaller-Seeds, and SyzVegas-Seeds. Figure 6 showcases the performance of the fuzzers with the initial seeds. In particular, MOCK-Seeds gains an 84% coverage growth in comparison to MOCK without seeds. The results demonstrate that MOCK-Seeds outperforms HEALER-Seeds, Syzkaller-Seeds, and SyzVegas-Seeds in terms of branch coverage growth, achieving 32%, 13%, and 8% higher coverage, respectively. Moreover, when considering the same level of branch coverage, MOCK-Seeds exhibits a remarkable speed-up, outperforming HEALER-Seeds, Syzkaller-Seeds, and SyzVegas-Seeds by 4.31×, 1.87×, and 1.56×, respectively. In addition to branch coverage, the length distribution of the generated corpus is another significant metric. In this regard, MOCK stands out by producing the highest number of interrelated sequences whose lengths are equal to or greater than five, amounting to 29.3%. Notably, comparing the results of fuzzing with and without seeds, we find that MOCK-Seeds could gain a higher speed-up (2.58× v.s. 1.71×) and produce considerably more interrelated syscall sequences.

The reason for this advantageous situation can be explained as follows. As described in Section IV-B, MOCK relies on the minimized syscall sequences as a golden training set during the fuzzing process, which embodies the valuable dependency information. With the aid of the initial seeds, MOCK could gather more valuable sequences in a shorter time and reduce the warmup time. Consequently, the context-aware dependency takes effect earlier and boosts the fuzzing performance.

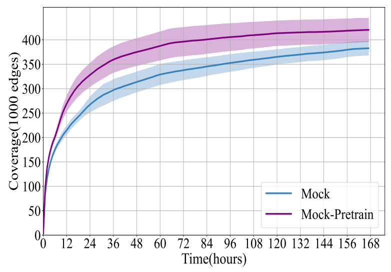


Fig. 7: The branch coverage of MOCK and MOCK-Pretrain.

Pre-trained Dependency. The utilization of initial seeds has been shown to offer benefits in terms of dependency learning and acceleration. This raises an intriguing question regarding the potential advantages of integrating a pre-trained dependency model into a fuzzer. To address the question, we enable MOCK with a pre-trained model and without initial seeds, denoting this setup as MOCK-Pretrain. The preparation of the pre-trained model involves combining syzbot's seeds with the accumulated corpus of MOCK to create the training set. Both syzbot's seeds and MOCK's accumulated corpus have undergone minimization and have contributed to new coverage or crashes, thus validating their high-quality nature. With the golden training set, the pre-trained model is trained and updated in the identical manner as described in Section IV-B.

Figure 7 illustrates the branch coverage of MOCK and MOCK-Pretrain. With the aid of the pre-trained model, MOCK-Pretrain gains knowledge of syscall dependencies from prior runs. This knowledge allows the fuzzer to avoid unnecessary exploration and focus on meaningful combinations of syscalls. Hence, MOCK-Pretrain demonstrates significant advantages over MOCK soon after startup. On average, MOCK-Pretrain averagely earns a 10% coverage growth and a 3.29× speed-up compared to MOCK.

In summary, our experiments have highlighted that MOCK can indeed benefit from the various setups, including initial seeds and pre-trained model. The initial seeds contribute to a reduction in warmup time for MOCK, enabling the context-aware dependency mechanism to bear fruit more quickly and leading to an 18% coverage growth and a 2.58× speed-up. On the other hand, the incorporation of a pre-trained dependency model also allows MOCK to conduct input mutation more effectively, resulting in a 10% coverage growth and an impressive 3.29× acceleration in the fuzzing campaign. These results further prove the significance of context-aware dependency information, an aspect that has been previously undervalued. MOCK successfully captures the dependency through a language model in a context-aware manner.

E. Vulnerability Detection Ability

To demonstrate MOCK's efficiency in vulnerability detection, we conduct experiments on an earlier version of Linux kernel 5.10 and compare the number of unique vulnerabilities reported by each fuzzer. We collect the count of vulnerabilities in the process together and manually inspect the syscall sequences that triggered the vulnerabilities.

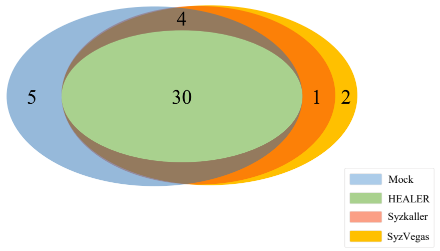


Fig. 8: The vulnerabilities discovered by MOCK, HEALER, Syzkaller and SyzVegas on Linux-5.10.

As is shown in Figure 8, MOCK can discover more vulnerabilities compared to HEALER, Syzkaller and SyzVegas within the same time. In the experiment, MOCK finds 39 unique vulnerabilities while HEALER, Syzkaller and SyzVegas find 30, 35 and 37 vulnerabilities, respectively, achieving 30%, 11% and 5% growth in capability in vulnerability detection. Especially, MOCK finds all the vulnerabilities discovered by HEALER and detects eight extra vulnerabilities. The input space of the vulnerabilities triggered by HEALER is not only covered but exceeded by MOCK. Moreover, MOCK detects 34 vulnerabilities that were also found by Syzkaller and SyzVegas, and five extra vulnerabilities, which means MOCK covers the majority of the input space of Syzkaller and SyzVegas. It is worth noting that the average length of syscall sequences to reproduce the five vulnerabilities uniquely discovered by MOCK is 16. MOCK has a similar performance over detecting shallow vulnerabilities that take shorter syscalls compared to the other two fuzzers. Regarding detecting the complicated vulnerabilities that are triggered by no less than five syscalls, MOCK outperforms Syzkaller and HEALER. Benefiting from the context-aware dependency, MOCK mutates syscall sequences in view of the calling contexts. Hence, it can prevent shallow, fruitless combinations and focus on producing in-depth syscall sequences more efficiently in the given time.

This is the reason why Mock outperforms those context-free fuzzers in vulnerability findings.

Overall, Mock finds 15% more vulnerabilities than the state-of-the-art fuzzers, which indicates Mock's effectiveness in vulnerability detection. The context-aware dependency also proves to be an effective approach that contributes to not only code coverage but also vulnerability findings, especially for the ones requiring complicated syscall sequences to trigger.

F. Real-World Vulnerabilities Discovery

We evaluate the capabilities of Mock in discovering read-world vulnerabilities. We run Mock on various versions of Linux kernels, including 4.19, 5.4, 5.10, 5.15, and 5.19, over a period of two weeks. Our fuzzer uncovers 15 unique practical vulnerabilities in total, of which four are confirmed and four are fixed by developers. Besides, we also have received two CVE IDs (CVE-2022-2978, CVE-2022-40476) , which have a 7.8 and 5.5 CVSS Score according to *NVD* [25]. The CVEs can lead to a high-security impact and attackers can potentially exploit the vulnerabilities to escape privilege or run denial-of-service attacks, breaking working systems. Table II shows brief information on the vulnerabilities found by Mock. The results indicate that Mock is capable of finding real-world vulnerabilities effectively, including two high-risk security vulnerabilities. We further describe one of them as a case study.

Case Study. The null pointer dereference is a prevalent memory failure, which takes place when a pointer with a value of null is used as though it pointed to a valid memory area. If a system runs and attempts to deference a null pointer, attackers could exploit the vulnerability to cause severe security consequences, e.g., overwhelming the system or running a denial-of-service attack.

Mock finds a null pointer dereference vulnerability in the `io_uring` module facilitated by KASAN. The `io_uring` module is designed as a new asynchronous I/O framework for Linux. It provides an efficient interface with rich features for applications that require asynchronous I/O functionality through the kernel. To properly execute system actions, kernel code sometimes needs to know the information of the current process driving it by accessing a global variable `current`. It works fines on most occasions as well as the preparatory stage of `io_uring`. Unfortunately, due to the characteristic of asynchrony, the process soon vanishes after issuing an asynchronous I/O task, and consequently, the variable `current` turns into a null pointer. A subsequent visit of the variable finally triggers the null pointer dereference vulnerability.

It takes a series of preparatory actions, including setting up contexts, submitting I/O tasks and etc., before accessing the null pointer. Mock could find the vulnerability since it effectively maintains or extends the calling states with the help of context-aware dependency. The other fuzzers may generate a similar combination occasionally, whereas they have trouble producing sufficient test cases in line with the required states to examine the targeted module fully. Mock learns dependency from historical knowledge and assigns more execution times to the in-depth syscall sequences to explore deep code space, leading to the discovery of the vulnerability.

TABLE II: List of new vulnerabilities discovered by Mock.

Subsystem	Crash Type	Operation	Kernel	Status
filesystem	use-after-free	nilfs_mdt_destro^C	4.14	Fixed
filesystem	kernel bug	btrfs_init_reloc_root	5.10	Reported
filesystem	kernel bug	__btrfs_drop_extents	5.10	Reported
filesystem	null-ptr-deref	io_req_track_inflight^C	5.15	Fixed
filesystem	use-after-free	ntfs_are_names_equal	5.15	Fixed
filesystem	deadlock	io_poll_double_wake	5.15	Reported
filesystem	kernel bug	ntfs_readpage*	5.15	Reported
filesystem	kernel bug	ntfs_read_folio*	5.19	Reported
drivers	warning	md_probe	5.19	Reported
drivers	deadlock	sch_direct_xmit	5.19	Reported
drivers	deadlock	rfcomm_sk_state_change*	5.19	Confirmed
network	refcount	bpf_exec_tx_verdict	5.19	Fixed
network	use-after-free	__fib6_clean_all$^\times$	6.0	Reported
network	use-after-free	nexthop_flush_dev$^\times$	6.0	Reported

* : Also reported by Syzkaller or HEALER.
$^\times$: Without syz repro. ^{C}: Received CVE ID.

G. Further Analysis

Stepwise Analysis. In order to measure the contribution of each core design, we construct another two fuzzers and compare them with Mock in terms of reaching code space. Table III illustrates the fuzzing results. To be specific, the base fuzzer (denoted as B) is only equipped with static dependencies; the model fuzzer (denoted as B+M) implements the context-aware dependencies with the language model based on the base fuzzer, but invokes the static-based and the context-aware dependencies by even weights without any schedule. Mock, denoted as B+M+S, is constructed as previously described with the static and context-aware dependencies as well as the task scheduler. From Table III, we make the following conclusion:

• The context-aware dependencies have little effect in the first few hours. This is because it takes time for Mock to prepare the training set in the initial stage. The minor training set also limits the language model's performance. However, the model takes an increasingly proactive role as fuzzing reaches later stages. Compared to the base fuzzer, the model fuzzer eventually brings 17% coverage improvement, which proves the value of context-aware dependencies.

• It is necessary for Mock to dynamically schedule the task selection to balance the exploration and exploitation. As Table III shows, a fixed weight increases the probabilities of picking model-based syscall selection, which could boost coverage growth over a short period. However, it leads to a local optimum and loses the initial advantages due to the lack of diversification in the final. The adaptive task scheduler adds a 14% coverage growth.

• Combining both the dependency-guided mutation and the task scheduling brings considerable benefits for code coverage growth. Every component in Mock has a crucial role to play. Notably, task scheduling internally serves as a close part of the mutation strategies to support the construction and the utilization of context-aware dependencies, which is the true meaning of task scheduling.

Performance Overhead. In this phase, we discuss the various overheads incurred by the system, which are mainly brought by the language model training, model-guided mutation, and task scheduling.

To eliminate the time cost of model training, several measures are taken. First, the neural language model is trained and loaded in parallel during every training cycle, ensuring that the fuzzing process is not suspended or interrupted. Second, a lightweight neural network structure is employed, which does not require substantial computing resources or time. As a result, it takes an average of 23 minutes to update the language model. The model-guided mutation module also benefits from the lightweight model. Compared to the static-based syscall selection, invoking the model costs extra 1.6 hours during a 24-hour experiment. Hence, the overall overhead of this module is no more than 7%. In terms of the task scheduling module, MOCK has to record additional information for each mutation task in each exploration stage and calculate the performance value in each exploitation stage. It totally takes eight minutes for MOCK to implement task scheduling, which accounts for less than 1% of the fuzzing time. Additionally, although the experiments are done on CPU in the paper, MOCK is highly recommended to be used with GPU for better performance. The reason is that MOCK is a GPU-friendly fuzzer and can further accelerate the mutation process with GPU which is widely used and generally available in the real world. The other approaches (e.g., HEALER, Syzkaller, SyzVegas) can hardly be migrated to GPU.

TABLE III: The stepwise analysis on each component of MOCK.

	B	**B+M**	**B+M+S**
design	-	context-aware dependency	task scheduling
branch coverage	286k	335k	383k
overhead	-	<7%	<1%

VII. DISCUSSION

Owing to minor modifications to the architecture and low overhead, the context-aware dependency proposed by MOCK could be conveniently applied to existing kernel fuzzers as a pluggable mutation operator. Despite the effectiveness and applicability of MOCK, we discuss the potential limitations of the work and future directions in the section.

Model Structure. MOCK employs the language model to learn the context-aware dependencies from a training set. MOCK is highly dependent on both the model structure and the composition of the training data. Therefore, selecting an appropriate model structure and training dataset is crucial for obtaining accurate results. In this work, we have used a lightweight neural network structure for a trial, but more complex neural networks may potentially enhance the performance. Additionally, incorporating extra features from the training set, such as parameter types and data flows of syscalls, could be introduced to the neural network to augment the model's perceptibility.

Concurrency Dependency. Although MOCK has demonstrated its outstanding capability in code coverage and vulnerability discovery compared to the fuzzers with context-free dependencies, its designs are mainly oriented towards the relations of sequential syscall executions. If two syscalls contain threads that affect each other when they are concurrently executed, it will be difficult for MOCK to capture such a concurrency dependency. This is because the sequence minimization algorithm is naturally based on the sequential executions of syscalls. For a huge-scale system like a kernel, it is common thing for threads to have concurrency issues. Scheduling threads and recording feedback to infer the potential concurrency influences in a more fine-grained manner could be a preferable solution. As a future work, it is interesting and challenging to model the concurrency dependencies on complex systems.

VIII. RELATED WORK

A. API-aware Fuzzing

Software such as cloud service, library, and kernel generally exposes a series of APIs for interaction. A large number of works have been proposed to find vulnerabilities via API-aware fuzzing [33], [52], [5], [20], [22], [10], [17]. RESTler [5] is a REST API fuzzer designed to automatically test cloud service via REST API. RESTler analyzes API specifications to determine producer-consumer relations, which guide the request combination. In terms of libraries, FuzzGen [20] leverages a whole system analysis to extract control-flow and data-flow dependencies from source code, and synthesizes target fuzzers for complex libraries. Jiang et al. [22] presented a fuzzer named RULF dedicated to Rust libraries. The core of RULF is to build an API dependency graph and then mutate sequences with the graph. SMARTION [10] infers the constraint of transactions and analyzes smart contract bytecodes. Based on the above information, SMARTION constructs initial seeds and evolves the seed pool to enhance smart contract fuzzing. IMF [17] introduces model-based API fuzzing, a novel method that exploits runtime logs to generate test cases for closed-sourced kernels. Nevertheless, due to the large code base and the rich states of the Linux kernel, the aforementioned approaches could not directly be applied to the fuzzing scenarios for syscalls of the Linux kernel.

B. Machine-Learning based Fuzzing

Machine learning techniques have played an important role in improving the effectiveness and efficiency of various fuzzing phases [42], [32], [11], [19], including seed synthesis, input generation, and mutation scheduling. Seed synthesis uses machine learning techniques to build high-quality seeds and boost the fuzzing process. Skyfire [51] employs probabilistic context-sensitive grammar (PCSG), which combines the syntactic and semantic features to automatically extract information from crawled corpus. Then the trained PCSG follows the rules laid by the author and generates an initial seed pool.

A series of works have been devoted to constructing coverage-oriented or vulnerability-oriented input. For example, NEUZZ [44] trains a neural network model on real-world program behaviors. After that, it deduces critical bytes that may influence a program's branches in a test case by combining the model and smooth techniques. Li et al. [29] proposed an approach called V-Fuzz, to train a vulnerability prediction model based on a graph-embedded neural network. The model orients the fuzzer towards the areas with potential vulnerabilities. Montage [26] is another work that uses a language model to generate syntactically and grammatically correct JavaScript programs. It transforms JavaScript AST into sequences and takes the program generation as a language modeling problem.

In regard to mutation scheduling, MOPT [31] is the first attempt to adaptively schedule mutation operations according to runtime feedback for AFL [54]. MOPT decides the probability distribution of operators with a customized particle swarm optimization method and selects the optimal accordingly. Wang et al. [50] suggested using a reinforcement learning algorithm to adjust the mutator and seed distribution in the process of fuzzing. They implemented a prototype named SyzVegas on top of Syzkaller to guide kernel fuzzing.

IX. Conclusion

In this paper, we present Mock, a kernel fuzzer equipped with the context-aware dependency, which selects proper syscalls according to the calling context so as to produce interrelated and in-depth input. In this regard, Mock learns the dependencies by training a language model on the dynamically collected corpus and applies a context-aware mutation schema. To balance the exploration and exploitation, Mock also dynamically schedules the context-aware mutation tasks. Our evaluation shows that Mock can achieve $1.07\times$ more branch coverage, produce $1.50\times$ more interrelated sequences, and discover $1.15\times$ more unique bugs than the state-of-the-art kernel fuzzers. In addition, various setups including initial seeds and a pre-trained model can enhance Mock's performance. Mock also finds 15 real-world bugs on the most recent Linux kernels with two CVEs assigned. Especially, Mock integrates a neural network model into the fuzzing system without relying on a pre-positioned training set, which will motivate further research.

Acknowledgement

We thank anonymous reviewers for their constructive feedback. This work was partly supported by National Natural Science Foundation of China under Grant No. U1936215, No. 61833015, No. 62293511 and No. 62088101, the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform, 226-2022-00107, 226-2023-00111), Meta Research Award, Okawa Fundation Grant and Google Research Scholar Award.

References

[1] "Healer's issue," https://github.com/SunHao-0/healer/issues/37.

[2] "Linux test project," https://github.com/linux-test-project/ltp, 2001.

[3] "The common vulnerability scoring system (cvss)," https://nvd.nist.gov/vuln-metrics/cvss, 2010.

[4] "Dirty cow (cve-2016-5195)," https://dirtycow.ninja/, 2016.

[5] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

[6] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[7] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency {Use-After-Free} bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 255–268.

[8] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," *Advances in neural information processing systems*, vol. 13, 2000.

[9] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[10] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[11] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[12] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[13] Google, "Kernel address sanitizer: a fast memory corruption detector for the linux kernel," https://google.github.io/kernel-sanitizers/KASAN, 2015.

[14] Google, "syzkaller: an unsupervised coverage-guided kernel fuzzer," https://github.com/google/syzkaller, 2015.

[15] Google, "syzkaller's minimization algorithm," https://github.com/google/syzkaller/blob/master/prog/minimization.go, 2015.

[16] Google, "syzbot," https://syzkaller.appspot.com/upstream, 2018.

[17] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2345–2358. [Online]. Available: https://doi.org/10.1145/3133956.3134103

[18] S. He, K. Shi, C. Liu, B. Guo, J. Chen, and Z. Shi, "Collaborative sensing in internet of things: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 3, pp. 1435–1474, 2022.

[19] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: a gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 138–145.

[20] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "{FuzzGen}: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287.

[21] J. Jackson, "Nasdaq's facebook glitch came from 'race conditions'," https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-.html, 2012.

[22] J. Jiang, H. Xu, and Y. Zhou, "Rulf: Rust library fuzzing via api dependency graph traversal," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 581–592. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678813

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

[24] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[25] I. T. Laboratory, "Nvd: National vulnerability database," https://nvd.nist.gov/.

[26] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2613–2630.

[27] C. Li, S. Ji, H. Weng, B. Li, J. Shi, R. Beyah, S. Guo, Z. Wang, and T. Wang, "Towards certifying the asymmetric robustness for neural networks: Quantification and applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3987–4001, 2022.

[28] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

[29] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," *arXiv preprint arXiv:1901.01142*, 2019.

[30] A. Lochmann, H. Schirmeier, H. Borghorst, and O. Spinczyk, "Lockdoc: Trace-based analysis of locking in the linux kernel," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–15.

[31] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[32] C. Lyu, H. Liang, S. Ji, X. Zhang, B. Zhao, M. Han, Y. Li, Z. Wang, W. Wang, and R. Beyah, "Slime: Program-sensitive energy allocation for fuzzing," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, p. 365–377.

[33] C. Lyu, J. Xu, S. Ji, X. Zhang, Q. Wang, B. Zhao, G. Pan, W. Cao, and R. Beyah, "Miner: A hybrid data-driven approach for rest api fuzzing," *arXiv preprint arXiv:2303.02545*, 2023.

[34] L. Mazare, "tch-rs: Rust bindings for the c++ api of pytorch," https://github.com/LaurentMazare/tch-rs, 2019.

[35] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[36] P. B. Oswal, "Improving linux kernel fuzzing," Ph.D. dissertation, 2023. [Online]. Available: https://www.proquest.com/dissertations-theses/improving-linux-kernel-fuzzing/docview/2812311865/se-2

[37] S. Pailoor, A. Aday, and S. Jana, "{MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[38] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1629–1642.

[39] H. Pu, L. He, P. Cheng, M. Sun, and J. Chen, "Security of industrial robots: Vulnerabilities, attacks, and mitigations," *Netwrk. Mag. of Global Internetwkg.*, vol. 37, no. 1, p. 111–117, jul 2022. [Online]. Available: https://doi.org/10.1109/MNET.116.2200034

[40] Pytorch, "Libtorch: Installing c++ distributions of pytorch," https://pytorch.org/cppdocs/, 2018.

[41] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, Z. Wen *et al.*, "A tutorial on thompson sampling," *Foundations and Trends® in Machine Learning*, vol. 11, no. 1, pp. 1–96, 2018.

[42] G. J. Saavedra, K. N. Rodhouse, D. M. Dunlavy, and P. W. Kegelmeyer, "A review of machine learning applications in fuzzing," *arXiv preprint arXiv:1906.11133*, 2019.

[43] K. Serebryany, "{OSS-Fuzz}-google's continuous fuzzing service for open source software," 2017.

[44] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[45] SimonKagstrom, "Kcov: code coverage for fuzzing," https://docs.kernel.org/dev-tools/kcov.html, 2010.

[46] stack.watch, "Linux kernel - security vulnerabilities in 2022," https://stack.watch/product/linux/linux-kernel/, 2022.

[47] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[48] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation learning guided kernel fuzzing," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 344–358.

[49] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *European conference on machine learning*. Springer, 2005, pp. 437–448.

[50] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, "{SyzVegas}: Beating kernel fuzzing odds with reinforcement learning," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2741–2758.

[51] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[52] M. Wunder, M. Littman, and M. Babes, "Classes of multiagent q-learning dynamics with epsilon-greedy exploration," ser. ICML'10. Madison, WI, USA: Omnipress, 2010, p. 1167–1174.

[53] Z. Yang, L. He, H. Yu, C. Zhao, P. Cheng, and J. Chen, "Detecting plc intrusions using control invariants," *IEEE Internet of Things Journal*, vol. 9, no. 12, pp. 9934–9947, 2022.

[54] M. Zalewski, "American fuzzy lop: a security-oriented fuzzer," https://lcamtuf.coredump.cx/afl/, 2013.

[55] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 442–454.

[56] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "{StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3273–3289.

X. Appendix

TABLE IV: The results of various exploration & exploitation ratios used by MOCK.

exploration & exploitation ratio	**branch coverage**
1:4	347k
2:2	379k
4:1	383k