HEALER: Relation Learning Guided Kernel Fuzzing

Hao Sun[1], Yuheng Shen[1], Cong Wang[1], Jianzhong Liu[1], Yu Jiang✉[1], Ting Chen✉[2], Aiguo Cui[3*]
KLISS, BNRist, School of Software, Tsinghua University, Beijing, China[1]
Center for Cybersecurity, University of Electronic Science and Technology of China, Chengdu, China[2]
Huawei Technologies Co., Ltd, China[3]

Abstract

Modern operating system kernels are too complex to be free of bugs. Fuzzing is a promising approach for vulnerability detection and has been applied to kernel testing. However, existing work does not consider the influence relations between system calls when generating and mutating inputs, resulting in difficulties when trying to reach into the kernel's deeper logic effectively.

In this paper, we propose HEALER, a kernel fuzzer that improves fuzzing's effectiveness by utilizing system call relation learning. HEALER learns the influence relations between system calls by dynamically analyzing minimized test cases. Then, HEALER utilizes the learned relations to guide input generation and mutation, which improves the quality of test cases and the effectiveness of fuzzing. We implemented HEALER and evaluated its performance on recent versions of the Linux kernel. Compared to state-of-the-art kernel fuzzers such as Syzkaller and Moonshine, HEALER improves branch coverage by 28% and 21%, while achieving a speedup of 2.2× and 1.8×, respectively. In addition, HEALER detected 218 vulnerabilities, 33 of which are previously unknown and have been confirmed by the corresponding kernel maintainers.

CCS Concepts: • **Security and privacy → Operating systems security**.

Keywords: Kernel Fuzzing, System Call Relation Learning.

ACM Reference Format:
Hao Sun[1], Yuheng Shen[1], Cong Wang[1], Jianzhong Liu[1], Yu Jiang✉[1], Ting Chen✉[2], Aiguo Cui[3]. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483547

*Yu Jiang and Ting Chen are the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483547

1 Introduction

The robustness of an operating system kernel is crucial for the overall system's security. Vulnerabilities of any type, such as data leaks and data races, would severely impact the safety of the system and would be catastrophic for user-level applications if exploited by an attacker. For instance, a data race that introduces circular lock behavior will render the kernel unresponsive, and the resulting deadlock may trigger a denial-of-service or a privilege escalation attack [1–3]. However, designing and maintaining kernels are complicated due to the complexity of modern computer architectures and the rapid development of new features. Therefore, modern operating system kernels are too large and too complex to be free of bugs. The Linux kernel has around 27.8 million lines of code in its Git repository and has witnessed hundreds of bugs reported in recent years, proving that hunting for and fixing kernel bugs are very important.

Traditionally, kernel developers mainly rely on handwritten test suites such as the Linux-Test-Project [34] to eliminate bugs in the kernel. However, manually eliminating bugs in the kernel with such a huge code-base is challenging, since it is difficult for handwritten test suites to keep up with the rapid increase of the kernel's size and complexity. Fuzzing [4, 12, 31] is a promising vulnerability detection technique and has been applied to assist kernel testing. Researchers have developed several kernel fuzzers that fill system call parameters with random input data to deliver test payloads to the kernel. This approach has made significant progress in finding kernel vulnerabilities [6, 17, 28]. However, after the low hanging fruit have been found and fixed, the effectiveness of these fuzzers decreased because the generated system call sequences and their parameters will be mostly likely rejected by parameter validation.

Recent coverage-guided kernel fuzzers [20, 21] have made further progress. For instance, Syzkaller [36], a state-of-the-art kernel fuzzer, uses system call descriptions and a choice table to generate system call sequences. The choice table records the probability of a system call being invoked before another to determine the sequence of system calls to invoke. Syzkaller also uses feedback analysis to refine its test case corpus iteratively. As a result, Syzkaller has successfully discovered numerous kernel vulnerabilities with the assistance of kernel sanitizers [14, 15]. Another example is the tool Moonshine [32], which aims to provide high quality initial seeds for Syzkaller using a seed distillation

algorithm. Moonshine extracts relevant system calls from existing handwritten test cases based on the system calls' read-write dependencies. Its generated seeds are used as the basis for further generation and mutation to produce high quality inputs and speed up the fuzzing process.

However, existing work does not consider the *influence relations* between system calls, i.e., the influence of one system call on the execution behavior of another, when generating and mutating inputs. The space of possible system call combinations is vast, where most combinations are invalid or equivalent sequences to another. The efficiency of kernel fuzzers will decrease significantly without employing effective methods that reduce the search space and increase the probability of generating valid test cases. Influence relations exist between two system calls if the execution of a former can alter the latter's execution path. For instance, system call `bind` can influence the execution path of `listen` because `bind` assigns the address to the socket that `listen` marks as accepting connections. System call `listen` may return early with `errno` `EDESTADDRREQ`, which indicates the socket has not been bound to an address without calling `bind` first. The basis for determining the system call invocation sequence is through the influence relations between system calls. By guiding test case generation and mutation with their relevant influence relations, the quality of test cases and the efficiency of the kernel fuzzer can be significantly improved.

Although Syzkaller's system call descriptions express the resource dependencies between system calls, it mainly conveys the structure and partial semantic information of the system call's parameters. Syzkaller's choice table is used to guide the system call sequence synthesis. Each item of the choice table is calculated by an empirical analysis algorithm and records the probability value that a system call should be invoked before another system call. However, each item in the choice table cannot convey the influence relation of system calls. We will demonstrate that Syzkaller's choice table may even hinder its test case generation and mutation capabilities in Section 3. Moonshine, on the other hand, applies static program analysis to infer the system calls' read-write dependencies. The inferred dependencies are then used for seed distillation to provide high-quality initial seeds for Syzkaller. However, its method does not consider the influence relations when generating and mutating inputs either.

To address the aforementioned challenges and improve the efficiency of kernel fuzzing, we propose `HEALER`, a kernel fuzzer that uses system call relation learning for highly effective input generation. `HEALER` borrows from Syzkaller the system call description format `Syzlang` to leverage the structure and semantic information of system calls. Test cases that trigger new coverage will be minimized so that only calls which contribute to the new coverage will be analyzed. These test cases will then be processed by the relation learning algorithm to infer whether there are influence relations between system calls. The learned relations will be stored in the relation table, which is refined step by step during the fuzzing campaign and will be used to guide test case generation and mutation to ensure that each system call in a test case is capable of accessing the kernel's deeper logic. In contrast to Syzkaller's choice table, each entry of `HEALER`'s relation table represents the influence relation between system calls. It is constructed and refined dynamically using the relation learning algorithm whenever an interesting test case is discovered. Compared to Moonshine, `HEALER` continuously learns and updates system call relations throughout the fuzzing process and applies them to guide the generation and mutation of high-quality system call sequences.

We implemented `HEALER` and evaluated its performance on recent versions of the Linux kernel. Our results show that `HEALER` achieves higher coverage than Syzkaller and Moonshine by 28% and 21% on average, respectively. Furthermore, `HEALER` achieves the same amount of coverage as that of Syzkaller and Moonshine with a speed-up of 2.2× and 1.8×, respectively. In addition to coverage improvements, `HEALER` found 218 unique vulnerabilities in total, 33 of which were confirmed as previously unknown. Most of those bugs are critical vulnerabilities. For instance, an uninitialized read bug in the function `fill_thread_core()` in the core dumping module `fs/binfmt_elf.c` may have existed in Linux kernel for as long as 12 years and can result in kernel memory content leakage. The bug has been reported to kernel maintainers and relevant patches have been merged upstream.

Our paper makes the following contributions:

- We propose to refine existing fuzzing techniques with system call relation learning to improve the quality of generated test cases and maximize program coverage.
- We implement `HEALER`, consisting of a relation learning component to initialize and iteratively update the relation table and a guided system call sequence generation and mutation component for kernel fuzzing.
- We evaluate `HEALER` on recent versions of the Linux kernel. `HEALER` covers more branches (28%-21%) than state-of-the-art fuzzers Syzkaller and Moonshine while being able to achieve equivalent coverage faster (2.2×-1.8×). `HEALER` also finds 33 previously unknown bugs, which have been confirmed by the maintainers.

2 Background and Related Work

Fuzzing is a software testing method that attempts to trigger bugs by repeatedly feeding a target program with generated inputs. [4, 5, 11, 13, 25, 27]. It is one of the most effective approaches for vulnerability detection. Most fuzzing tools generate interesting test cases with little domain knowledge or manually constructed inputs, which makes fuzzing more accessible than other bug-finding methods. Take AFL [39], a popular user-space program fuzzer, as an example. AFL has found hundreds of vulnerabilities in widely-used libraries,

while only requiring several input files to bootstrap the fuzzing process. The input generators of similar fuzzers use a genetic mutation algorithm that is especially effective in producing test cases that explore corner cases in program execution paths, which is difficult for manual testers to perform. Many AFL-derived fuzzers have been developed to perform fuzzing in specialized environments or to improve its effectiveness [7, 8, 10, 16, 22, 26, 29, 40].

To perform coverage-guided kernel fuzzing, researchers have ported AFL from user-space to kernel-space by combining AFL and QEMU's full-system simulation. QEMU is used to run the kernel as it runs in privileged mode and requires direct access to the hardware. Then, the fuzzer injects an agent process to the guest virtual machine. The agent process communicates with the outside fuzzer, receives the message sent from the fuzzer, decodes them into a sequence of system calls, and issues the system calls to the kernel accordingly. Furthermore, it also performs bookkeeping functions to drive the fuzzing loop. For instance, kAFL utilizes hardware features in Intel processors to collect branch coverage information and extends AFL's fuzzing techniques with QEMU's full system emulation to fuzz kernels [33]. TriforceAFL [18] is a patched version of AFL that supports full-system fuzzing based on QEMU. Triforce Linux Syscall Fuzzer [19], which is based on TriforceAFL, performs kernel fuzz testing by decoding inputs from AFL and has already found several critical kernel vulnerabilities. Furthermore, some kernel fuzzers perform fuzz testing by directly generating sequences of system calls with initialized parameter values [9, 23, 24, 30, 35, 38]. After the execution of generated test cases, the fuzzers analyze the collected execution traces. Test cases that trigger new branches are saved for additional processing, such as further mutation. This way, coverage-guided kernel fuzzers can explore kernel states more efficiently than non-coverage-guided fuzzers.

Google's Syzkaller is one of the most widely used coverage-guided kernel fuzzers. Syzkaller generates and mutates test cases (system call sequences) based on system call descriptions, the corpus and the choice table. It implements a domain specific language (`Syzlang`) to describe the syntax and partial semantics of system calls, which enables Syzkaller to start the fuzzing process without any initial input system call sequences (seeds). Syzlang consists of basic types and type constructors, e.g., `struct`, as well as special type modifiers that provide the semantics of parameters, e.g., `resource`. The `resource` type indicates that the parameter value is obtained from the output of another system call, such as file descriptors. Syzlang supports resource inheritance. For instance, opening a KVM device to set up virtualization returns a resource value of `KVM` type. Since the KVM device type is a subtype of the file descriptor type, system calls accepting file descriptors also accept KVM devices. Utilizing Syzlang's inheritance capabilities, fuzzers can automatically generate generic file descriptor system calls such as `close` after opening a KVM device. In addition, Syzlang supports system call specializations, which instantiates some of the arguments of the original system call, where the specialized call is named as `original_name$custom`. For instance, `open$kvm()` specializes the first argument of `open()` to "/dev/kvm". The choice table, which records the probability that a call should be inserted before another call, is used to select target system calls. Based on the provided information, Syzkaller continuously generates and executes test cases while monitoring whether the kernel under test crashes. Syzkaller has detected many critical bugs in the Linux kernel, demonstrating its ability to find kernel bugs within the upstream Linux kernel [37]. However, the complexity of system calls' relations limit Syzkaller's effectiveness.

3 Motivation

While the Linux kernel defines over 300 system calls, Syzlang describes nearly 4000 interfaces which contains many specialized calls. Typically, the length of a system call sequence generated by a kernel fuzzer ranges from 8 to 32 individual calls. Therefore, the number of possible call sequences is $\sum_{k=8}^{32} \binom{4000}{k}$, which is on the order of magnitude of 10^{80}, without considering the order of calls. Syzkaller can execute 10^2 to 10^3 call sequences per second, so it would take nearly 10^{69} years to run all possible call combinations. Furthermore, most call combinations are actually invalid and equivalent with other similar sequences. For instance, if most system calls in a test case were executed without the required kernel state, then they would most likely exit early, preventing fuzzers from accessing the kernel's deeper logic. Therefore, the efficiency of kernel fuzzers would be significantly limited without effective methods that reduce the search space and increase the probability of generating valid test cases.

Operating system kernels are large systems with complex internal states, which affect the execution paths of system calls. Almost every system call accesses certain pieces of kernel data during execution. Different system calls may influence each other through accessing and modifying common internal data. At the same time, some calls may not share any common data with another call entirely, i.e., they do not affect each other regardless of the current kernel state. Therefore, the fundamental principle for deciding whether a system call should be called before another call is the influence relations between calls. If the execution of C_i will affect the execution path of C_j, then C_i should have a higher probability of being executed before C_j. On the contrary, if C_i has no effect on the execution of C_j (e.g., C_i and C_j access totally different internal states), then it is meaningless to execute C_i before C_j. Furthermore, some execution paths of one system call may only be executed in certain kernel states, which often hide deep and hard-to-find kernel vulnerabilities. The required kernel states may only be triggered by invoking specific system calls. Therefore, kernel fuzzers should insert more system calls that have influence relations before the

target system call, thereby triggering different kernel states and allowing each system call to enter deep, rare execution paths. The number of invalid test cases and the size of the search space can be reduced significantly by taking relations between system calls into consideration.

Listing 1. Buggy code of kernel routine `search_memslots`.

```
struct kvm_memory_slot *
search_memslots(struct kvm_memslots *slots,
    gfn_t gfn)
{
    ...
    // After the loop, start may equal to end
    while (start < end) {
        slot = start + (end - start) / 2;
        if (gfn >= memslots[slot].base_gfn)
            end = slot;
        else
            start = slot + 1;
    }
    // FLAW: out-of-bound access
    if (gfn>=memslots[start].base_gfn &&...){
        ...
    }
    return NULL;
}
```

Take Listing 1 as an example. It shows a piece of vulnerable code, which is located deep within the `kvm` module of the Linux kernel. While invoking the `ioctl` system call that starts the `kvm`, the Linux kernel will find the slot which contains the input `gfn` through binary search. However, in some corner cases, the slot index returned by the binary search may be in an invalid state, resulting in a subsequent out-of-bounds access. Triggering the aforementioned vulnerability requires a complex combination of system calls and kernel states, specifically: the kernel fuzzer needs to open the `kvm` file with the correct path, then create a virtual machine instance inside the kernel, configure its virtual processor and memory information as well as other attributes, and finally invoke the `ioctl` system call to run the virtual machine. Thus, the fuzzer must at least combine system calls as below (in `Syzlang` description format):

1. `open$kvm` ,
2. `ioctl$KVM_CRATE_VM`,
3. `ioctl$KVM_CREATE_VCPU`,
4. `ioctl$KVM_SET_USER_MEMORY_REGION`,
5. `ioctl$KVM_RUN`

to trigger this vulnerability. Being aware of the influence relations between these system calls (as shown in Figure 5) during test case generation and mutation will speed up the discovery of this vulnerability as related calls have a higher probability to be combined together, otherwise the huge search space will render fuzzing extremely inefficient.

Syzkaller generates test cases based on the system call descriptions, input corpus and choice table. The choice table, which records the probability value that a system call should be invoked before another system call, guides the call selection during test case generation and mutation and plays a significant role in improving the quality of test cases. Each item of the choice table P_{ij} for system calls C_i and C_j is calculated by $P_{ij} = (P0_{ij} * P1_{ij})/1000$. The value of P_{ij} is determined by $P0_{ij}$ and $P1_{ij}$, which are calculated by the static and dynamic analysis algorithms respectively. To scale the values to the same magnitude, both $P0_{ij}$ and $P1_{ij}$ are normalized to 10$\tilde{1}$000 with a factor of 1000. The static algorithm is based on specific common types between two system calls. Each common type has a hard-coded weight, e.g., 10 for `resource` type and 5 for `vma` type. $P0_{ij}$ is the sum of all weight value of common types. Based on the input corpus, the dynamic algorithm counts the adjacent calls in all system call sequences. The more adjacent calls corresponding to (C_i, C_j) occurs, the greater that $P1_{ij}$ will be.

The basis for deciding whether a system call should be invoked before another system call is the influence relations as mentioned above. However, Syzkaller's choice table synthesis algorithm does not conform to these rules. Its static analysis algorithm tends to assign two calls with more arguments in common to have a greater probability value. However, the number of common types does not reflect the influence relations between system calls. Its dynamic analysis algorithm tends to increase the probability of adjacent calls in the corpus. However, the continuous call sequence in the corpus does not necessarily have an influence relation either. For instance, $[C_i, C_j, C_k]$ is the system call sequence after minimization, but C_i may not have an impact on the execution path of C_j. The reason that C_i is not deleted by the minimization algorithm from original sequence is that C_i may set up a specific internal state for C_k. Increasing the probability of C_i and C_j ($P1_{ij}$) will obviously mislead test case generation and mutation.

Moonshine designed a seed distillation algorithm to provide initial seeds for Syzkaller. It first uses `strace` to track the system call sequences during the execution of existing test cases. Then, the distillation algorithm uses a static analysis algorithm to analyze the read-write dependencies of system calls and filter the collected call sequences with obtained dependencies. System calls that do not have read-write dependencies with other calls in the sequence will be removed. It provides high-quality initial seeds effectively. However, the distillation algorithm is only applied before the fuzzing process, thus it does not consider the influence relation between system calls when generating and mutating test cases.

In order to reduce the search space and improve fuzzing efficiency, we propose to consider the influence relations between system calls during test case generation and mutation. We design a dynamic relation learning algorithm to identify

whether there exists influence relations between any two calls. The algorithm refines the influence relations by analyzing the sequence of minimized calls during fuzzing. Based on the relation information, we propose a corresponding guided generation and mutation algorithm to increase the probability of system calls executing deep kernel logic.

4 HEALER Design

Figure 1 shows the main components and the overall workflow of `HEALER`. `HEALER` borrows Syzkaller's `SyzLang` description, which can provide system call information of the kernel under test, such as the input structure and partial semantics of the parameters. Based on the given information, `HEALER` can generate system call sequences that satisfy structural and partial semantic constraints while applying different mutation operators for each specific type. The input corpus is a set of high-quality test cases consisting of call sequences accumulated during fuzzing. The user can optionally provide an initial corpus to help accelerate the whole process.

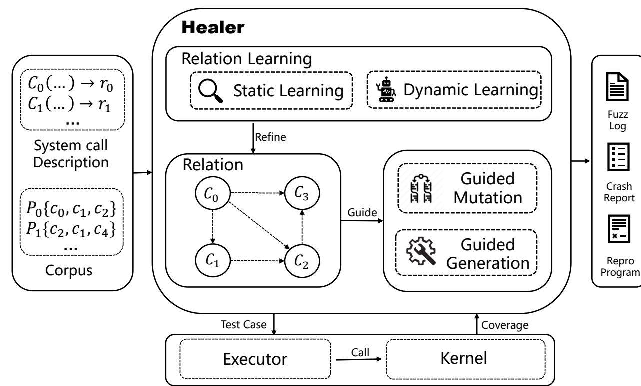

Figure 1. Overview of `HEALER`. System call descriptions contain existing system call information. The relation learning module analyzes the minimized test cases, and refines the relation table during fuzzing. At the same time, guided generation and mutation module generates system call sequences based on the learned relations.

The major improvement of `HEALER` over the state-of-the-art is that `HEALER` uses a relation table to guide generation and mutation. The relation table records the influence relations between system calls. `HEALER` continuously updates and refines the relation table using the learning algorithm. Based on the relation table, `HEALER` can increase the probability of the tested system calls reaching deep kernel logic, thereby improving the effectiveness of fuzzing. `HEALER`'s executor will then execute the generated test cases and monitor the fuzzing process to detect whether the kernel under test has crashed. If so, `HEALER` will collect and parse the crash log, such as symbolizing kernel addresses and filtering out irrelevant information. `HEALER`'s crash reproduction component will try to extract the smallest test case that can trigger the crash based on the execution trace and provide a corresponding minimized system call sequence; if the kernel finishes execution without crashing, `HEALER` will collect and analyze its branch coverage feedback. If a test case achieves new branch coverage, `HEALER` will minimize the test case and learn the influence relations between the calls in the minimized test case, subsequently saving information in the relation table using the relation learning algorithm. The learned relations will be used for improving test case generation and mutation.

4.1 Relation Learning

We propose an effective *relation learning algorithm* so that the fuzzer is aware of the relationships between system calls. We define *influence relations* in `HEALER` as follows:

Definition 4.1 (Influence Relation)**.** A system call C_i has an influence on another system call C_j if the execution of C_i can influence the execution path of C_j by modifying the kernel's internal state.

Relations between system calls refers specifically to the effect of one call on the execution path of another call. The reason for the influence is that the current call modifies the global state on which the other call's execution behavior depends. The static and dynamic relation learning algorithm intends to identify whether such an influence relation exists between any two system calls.

Relation Table. The relation table is a two-dimensional table $R^{n \times n}$ that is used to record the influence relations for any n system calls. The value R_{ij} in the table $R^{n \times n}$ is **1**, if system call C_i has an impact on C_j's execution trace, whereas **0** represents the opposite situation. At the beginning of the fuzzing process, the fuzzer may not be aware of most influence relations due to a limited amount of initial test cases. We set the table entries of unknown relations to zero. For instance, the influence relation between the system call `fcntl$ADD_SEALS` and `mmap` as shown in Figure 2 may be unknown initially, thus R_{23} and R_{32} are set to 0.

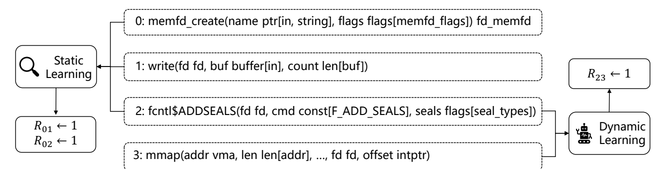

Figure 2. An example of relation learning. With the static part of algorithm, `HEALER` can infer R_{10} and R_{20} equal to 1. The relation between `fcntl$ADD_SEALS` and `mmap` can be identified through dynamic analysis during fuzzing.

Overall, the relation learning algorithm is divided into the static and dynamic learning routines. The static routine identifies influence relations based on the input parameter

types and return types of system calls. The dynamic routine can find influence relations not expressible by system call descriptions. The algorithm first minimizes the system call sequences, then executes the minimized sequence to obtain the coverage feedback for individual system call, and finally analyze the corresponding feedback to determine the influence relations.

Static Learning. The static learning routine initializes the relation table based on the information provided by `Syzlang` descriptions. Parameter types are essential for system call relation identification. Consider the impact of system call C_i on another system call C_j. We determine that C_i has an impact on C_j, thus $R_{ij} = 1$, when the following conditions are satisfied: (1) the return type of C_i is a resource type r_0, or any parameter in C_i is a pointer of this resource type with an outward data flow direction, and (2) at least one of C_j's parameters is a resource type r_0 or resource type r_1 that is compatible with r_0 with an inward data flow direction. `Syzlang` supports inheritance between resource types with the following rules. If r_1 inherits from r_0, then r_0 is compatible with r_1 and r_1 is the subtype of r_0. These derivation rules take inheritance relationships between resource types into account. As Figure 2 shows, `memfd_create()` influences `write` ($R_{10} \leftarrow 1$) because `write()` takes `fd` as a parameter from `memfd_create`. Since the parameter `fd` is resource type and `fcntl$ADD_SEALS` takes it from `memfd_create`, we can infer that R_{20} should also be 1.

Dynamic Learning. While the static learning routine of the relation learning algorithm allows for accurate analysis of influence relations expressible by system call descriptions, the dynamic learning component continuously updates and refines the relation table with information not expressible by the descriptions so that `HEALER` is able to generate high quality test cases. Specifically, instead of viewing the system call sequence as a whole like conventional fuzzers, `HEALER` individually collects coverage for each call in the sequence and stores the sequence of identifiers denoting the triggered basic block or edge to discover the coverage changes. Whenever a test case achieves new coverage, `HEALER` tries to extract as much information from it as possible using the following procedure. First, the relation learning algorithm performs minimization to obtain a smallest possible test case that exhibits the same coverage behavior. The purpose of this step is to filter out system calls that do not contribute to the new coverage and improve the efficiency of the analysis procedure. After minimization, the algorithm will gradually remove each call in the minimized call sequence and analyze the impact of each removal operation on adjacent calls to identify the influence relation.

Algorithm 1 shows the procedure of sequence minimization. The inputs to the minimization algorithm are test cases consisting of system call sequences p and the list of new coverage achieved by each call of p. The coverage is used to determine whether each removal operation allows the test case to preserve its new coverage. The algorithm extracts the subsequence p' for each call C_i that has not yet been included in the other minimal sequences and has triggered new coverage in reverse order (Lines 3-7). This ensures that the resulting subsequences are independent and non-repetitive. The algorithm then attempts to remove each call C' before C_i in p' (Lines 9-10). If the removal does not affect the execution path of C_i, then p' is successfully minimized once (Lines 13-14); otherwise it means that C' cannot be removed and is therefore retained (Lines 15-17).

Take Figure 2 as an example, `HEALER` collects the coverage of [`memfd_create`, `write`, `fcntl$ADDSEALS`, `mmap`] as [cov_0, cov_1, cov_2, cov_3]. Suppose cov_3 contains new coverage. After the iteration of Algorithm 1, system call `write` is removed because it does not contribute to the new coverage of cov_3. In contrast, `memfd_create` and `fcntl$ADDSEALS` are preserved and the minimized sequence [`memfd_create`, `fcntl$ADDSEALS`, `mmap`] is returned.

Algorithm 1: Sequence Minimization

Input: sequence p, coverage of each call *covs*
Output: minimized sequences P

```
1  P ← empty-list
2  R ← empty-list      // indexs of reserved calls
3  for i = (len(p) − 1) → 0 do
4      if i ∈ R or is_empty(cov[i]) then
5          continue
6      R ← append(R, i)
7      p′ ← p[0 : i + 1]
8      last ← i
9      for j = (i − 1) → 0 do
10         p″ ← remove(p′, j)
11         last ← last − 1
12         covs′ ← exec(p″)
13         if covs′[last] = covs[i] then
14             p′ ← p″
15         else
16             R ← append(R, j)
17             last ← last + 1
18     P ← append(P, p′)
```

Algorithm 2 shows the overall procedure of dynamic learning. First, the test case p is minimized to P (Line 1). For each system call C_j in p' except for the first one (Line 4), suppose the previous call of C_j is C_i (Line 5), if the relation between C_j and C_i is unknown (Line 6), then we remove C_i from p' (Line 7) and execute the modified test case (Line 8). If the coverage under the modified test case changes, then we determine that C_i has an impact on the execution of C_j, thus R_{ij} is set to 1 (Lines 9-10). For instance, suppose $[C_0, C_1, C_2]$ is a minimized call sequence, if the removal of C_1 changes the coverage of C_2,

6

then C_1 must have an impact on C_2. However, if the removal of C_0 results in the coverage change of C_2, it is possible that it is caused by the coverage change of C_1 indirectly. Therefore, the algorithm only analyzes consecutive calls because coverage changes that are caused by non-consecutive calls' removals cannot demonstrate the influence relations.

Algorithm 2: Relation Learning

```
Input: call sequence p
Output: updated relation table R_nn
// minimize p first
1  P ← minimize(p)
2  for p' ∈ P do
3      for c_j ∈ p' do
4          if c_j != first(p') then
5              c_i ← previous(p', c_j)
6              if R_ij = 0 then
7                  p'' ← remove(p', c_i)
8                  cov ← execute(p'')
9                  if cov of c_j changed then
10                     R_ij ← 1
```

The aforementioned minimized call sequence [memfd_-create, fcntl$ADDSEALS, mmap] contains two consecutive calls. Algorithm 2 will skip the first one ([memfd_create, fcntl$ADDSEALS]) because it has already been detected by the static learning. HEALER can infer that fcntl$ADD_SEALS has an impact on mmap, i.e., $R_{32} \leftarrow 1$, which cannot be determined by the static learning routine, because the removal of fcntl$ADD_SEALS changes the coverage of mmap. By analyzing the information provided by the minimized test cases, HEALER can refine the relation table during the whole fuzzing process, thereby providing sufficient information for generation and mutation.

4.2 Guided Generation and Mutation

The learned relations can be utilized in multiple parts of fuzzing process with different aims depending on the fuzzer design. In HEALER, relations are mainly used to guide system call sequence mutation and generation. In general, both mutation and generation are divided into call selection and parameter synthesis procedures, where call selection adds new calls to an existing call sequence and parameter synthesis generates specific values based on the parameter type. During parameter synthesis, HEALER uses similar methods to that of existing work, designing different generation strategies and mutation operators for different types of parameters, for example magic-number-based generation and bit-flip mutation for numerical types. The most significant improvement over existing work is that HEALER performs call selection based on the influence relation in the relation table.

HEALER performs mutation on existing system call sequences in the corpus. These are preserved because of their ability to trigger new coverage and represent interesting combinations of system calls. After selecting the mutation target, HEALER randomly chooses insertion points for new calls in the target sequence. The sub-sequence preceding each insertion point is used as the input for call selection and the mutation module selects the new call for the insertion point using the algorithm as shown in Algorithm 3. When the gain from mutation decreases, HEALER will try to generate new system call sequences. The main operation of generation is sequence synthesis, which chooses a list of system calls based on the learned relations and their respective Syzlang descriptions. At the beginning of the generation process, HEALER mainly considers the producer and consumer of resource types based on the information provided by the Syzlang descriptions. HEALER randomly combines producer calls and consumer calls for specific or compatible resources. Subsequently, the generation module uses the Algorithm 3 multiple times to refine the new call sequence.

Algorithm 3: Guided Call Selection

```
Input: relation table R_nn, sub-sequence S
Output: selected call C
1  if rand() > α then
2      return random call // randomly select with
       probability α
3  M ← empty − map // construct candidate list
4  foreach c_i ∈ S do
5      foreach r_ij ∈ R_in do
6          if r_ij = 1 then
7              M[c_j]+ = 1
8  if is_empty(M) then
9      return random call
10 else
11     return random_weighted(M)
```

The guided call selection algorithm decides whether to use the relation table and how to select calls based on the information it provides. At the beginning of the fuzzing process, the relation table may not contain much information, thus excessive use of the relation table with insufficient information to guide call selection may lead to a reduction in test case diversity. On the other hand, the quality of the test cases cannot be guaranteed without using the learned relations. Therefore, the algorithm uses a dynamically adjusted parameter α to strike a balance of whether it should leverage the relation table information (Lines 1-2). Specifically, the call selection method of each test case and whether it yields new coverage are recorded. α is updated to the rate of return

in using the relation table after every 1024 test cases executed. Its value increases if more coverage can be obtained with the use of the learned relations. The algorithm then determines a list of candidate system calls and the weight of each candidate call based on the relation table and the input sub-sequence (Lines 3-7). The candidates are the calls whose execution paths may be influenced by the calls in the input sub-sequence S, while the weight is the number of calls in the sub-sequence that can have an influence on the corresponding candidate call. Once the information is collected, the algorithm makes a random selection based on the weights; a higher the weight corresponds to a higher probability of being selected (Lines 8-11).

Using the guided generation and mutation, HEALER can generate test cases using even the most implicit relations between calls. Each call in a test case has a higher probability of accessing the deeper logic of the kernel because the call it depends on has a higher probability of being executed and the corresponding kernel state has a higher probability of being set, thus improving kernel fuzzing effectiveness.

5 Implementation

Overall, we implemented HEALER using 14,919 lines of Rust. In order for the relation learning algorithm to run efficiently, we designed a fuzzing architecture that is different from existing state-of-the-art kernel fuzzers. From an architectural view, Syzkaller runs the fuzzer and executor inside the kernel under test which is executed in a virtual machine, then synchronizes the state of the different fuzzers via RPC and monitors the virtual machines via an external manager. This architecture incurs significant additional IO overheads. A complex state synchronization mechanism would be required with the increase of the fuzzer's design complexity, as the fuzzing state is scattered across the VMs.

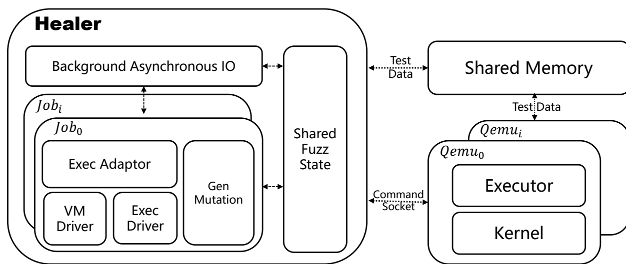

Figure 3. Architecture of HEALER

To address these issues and improve relation learning efficiency, HEALER uses a different architecture, as shown in Figure 3. The core runtime components of HEALER are the worker threads (Job_i), the background IO thread, fuzzing shared state, the QEMU instances and the executor. The worker threads perform the entire fuzzing process, which runs on the host kernel rather than the kernel under test. Different worker threads synchronize directly with each other via the shared fuzzing state. Only the executor runs the generated system call sequences inside the kernel under test. The adapter layer of worker thread defines the execution interface, combines the implementations of underlying modules, which manage the VM and communicate with the specific executors. We implement a high performance background asynchronous IO worker, which monitors and collects log data from multiple QEMU instances. Commands and status data, such as handshake packets, are exchanged between the executor and fuzzer via a control socket. Leveraging QEMU's Inter-VM Shared Memory device (`ivshmem`), the executor uses the shared memory with the fuzzer by accessing the emulated PCI device, where new test cases will be serialized into a compact internal representation. With this framework, HEALER achieves efficient communication between the fuzzer and the executor while retaining scalability and eliminating the need for a corresponding synchronization mechanism for complex fuzzer designs.

The relation table is implemented with a high performance hash-table that is shared between different worker threads and optimized for access speed through read-write lock. The relation learning algorithm utilizes the capabilities provided by the execution module for sequence minimization and dynamic analysis. The static learning routine is implemented by analyzing the compiler-provided Abstract Syntax Tree (AST) of the system call description.

6 Evaluation

In this section, we evaluate the effectiveness of HEALER on recent versions of the Linux kernel. Kernel fuzzers should be able to cover the execution path of the kernel under test as much as possible efficiently while triggering as many kernel vulnerabilities as possible. Therefore, we designed experiments to compare code coverage capabilities and vulnerability discovery capabilities with Syzkaller and Moonshine. We choose Syzkaller because it is the most widely used, effective and still actively maintained kernel fuzzer. Moonshine improves Syzkaller's fuzzing efficiency by performing seed distillation based on static analysis of read-write dependencies. HEALER guides the generation and mutation based on dynamic relation learning for a improved efficiency. This makes the comparison between HEALER and Moonshine's distillation process very tempting. In our evaluation, Moonshine indicates combining Syzkaller with Moonshine's distilled initial seeds. Furthermore, we evaluate the effectiveness of the relation learning algorithm itself and discuss the impact of relations on the overall fuzzing process. We design experiments to address the following questions:

- **RQ1: How well does HEALER perform compared to Syzkaller and Moonshine?** To answer this question, we should evaluate the fuzzers on the amount of coverage achieved under a fixed time constraint.

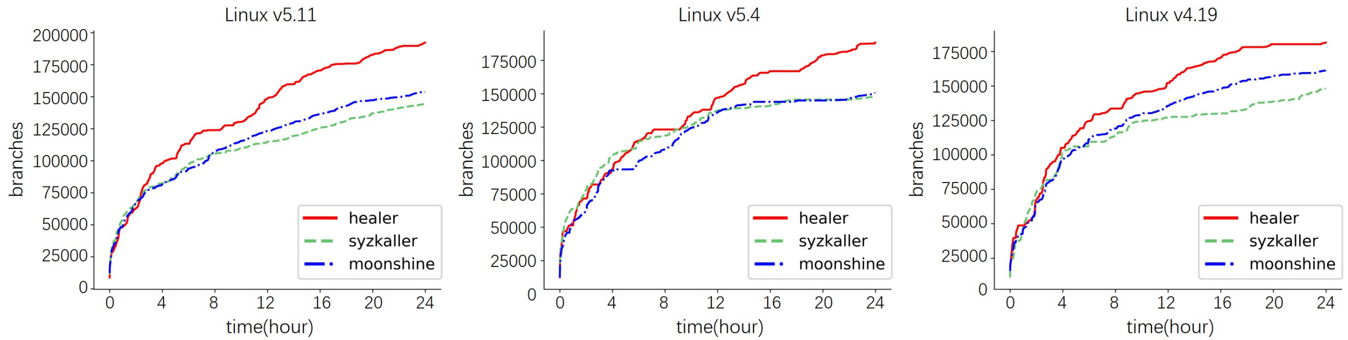

Figure 4. Branch coverage growth of HEALER on Linux kernel versions 5.11, 5.4 and 4.19 over 24 hours compared to those of Syzkaller and Moonshine. In all three kernel versions, HEALER achieves the highest coverage statistics.

- **RQ2: How effective is relation learning in assisting test case generation and mutation?** Relation learning is a significant part of HEALER, thus we need to analyze the impact of the component.
- **RQ3: How does HEALER perform in vulnerability detection?** The ability to discover bugs should be demonstrated across more kernel versions when executed for an extended period of time.

Experiment Setting. The experiments were conducted on a server with a 16-core CPU and 32 GiB of memory running Linux as the host kernel. We chose Linux-5.11, 5.4 and 4.19 as our test kernel targets. Linux 5.11 is the latest version prior to submission, whereas Linux 5.4 and Linux 4.19 are the most widely used kernel versions by many distributions. Each version of the kernel uses the same compilation configuration, while KCOV and KASAN features are enabled in order to collect code coverage and detect memory errors. To evaluate the effectiveness of the relation learning algorithm and exclude the effect of architectural differences on the experiment results, we designed HEALER-, a special version of HEALER that does not use relation learning. Syzkaller, Moonshine, HEALER- and HEALER are configured with the same parameters, such as QEMU configurations, system call descriptions, etc. Specifically, for strict control of computing resources, we started all experiments simultaneously and distributed the resources evenly, including 2 cores and 4 GiB of memory for each virtual machine. All tools use the same version of the `Syzlang` description (revision 0085e0). In addition, Syzkaller, HEALER- and HEALER do not use any initial seeds, whereas Moonshine uses `strong_distill.db` as its seed input, the default configuration used in its paper. To reduce statistical errors, each set of experiments is repeated 10 times and each experiment is executed over a period of 24 hours. We report the average values of the results.

6.1 Performance of HEALER

In this experiment, we monitor the fuzzing process of Linux kernel versions 5.11, 5.4 and 4.19 while comparing the code coverage capabilities to those of Syzkaller and Moonshine. We sample each fuzzer's statistics each minute in the 24-hour run. Finally we calculate the average value of each fuzzer's sampled data over its 10 executions.

Figure 4 shows the comparison of branch coverage between HEALER and the comparison fuzzers. As shown in the figure, HEALER can achieve higher coverage statistics compared to Syzkaller and Moonshine in the same amount of time. Specifically, all tested tools show significant growth in the first 8 hours then gradually slow down. Note that at the beginning of each experiment, the coverage achieved by HEALER is similar to that achieved by the comparison fuzzers. This is because at the beginning of each experiment, HEALER does not have enough information about the relations between system calls. Once the relation table contains more information after being continuously refined by HEALER's learning algorithm, HEALER's coverage statistics becomes significantly better than the comparison fuzzers. At around 16 hours into the experiment, the relative performance of the tools can be established. Table 1 lists HEALER's coverage improvement statistics over Syzkaller and Moonshine. Columns "min-impr" and "max-impr" present the minimum / maximum coverage improvement among all the experiment rounds. The column "Overall" shows the average coverage improvement. For example, on the version 5.11 kernel, HEALER achieves 33.00% more branch coverage than Syzkaller on average. Column "speed-up" presents the average speed-up of HEALER in achieving the same amount of coverage as Syzkaller and Moonshine did on each kernel, respectively.

On average, HEALER achieves 28% more branch coverage than Syzkaller with a 2.2× speed-up on average. Compared to Moonshine, HEALER achieves 21% more branch coverage with a speed-up of 1.8×. The increase in coverage statistics proves that HEALER can explore more code in the Linux kernel than Syzkaller and Moonshine. The reason for the improvement is that HEALER is capable of handling the complexity of the system call relations. Therefore, the call sequences generated

by HEALER would not be rejected early and can be more effective in exploring code branches.

Table 1. Branch coverage statistics of HEALER compared to Syzkaller, Moonshine

(a) HEALER vs. Syskaller

Version	min-impr	max-impr	Average	Speed-up
5.11	+28%	+39%	+33%	+2.0×
5.4	+15%	+35%	+27%	+1.9×
4.19	+17%	+28%	+22%	+2.7×
Overall	+20%	+34%	**+28%**	**+2.2×**

(b) HEALER vs. Moonshine

Version	min-impr	max-impr	Average	Speed-up
5.11	+18%	+35%	+25%	+1.9x
5.4	+17%	+27%	+25%	+1.9x
4.19	+10%	+25%	+12%	+1.8x
Overall	+15%	+29%	**+21%**	**+1.8x**

6.2 Effectiveness of Relation Learning

In order to evaluate the effectiveness of HEALER's relation learning algorithm and exclude the effect of architectural differences, we compared the performance of HEALER with HEALER- and collected the learned relations during each round of experiments.

Table 2 shows the detailed statistics of coverage improvement. We can see that HEALER achieves 34% more branch coverage than HEALER- with a 2.4× speed up. HEALER outperforms HEALER-, Syzkaller, Moonshine by 34%, 28%, 21%, respectively. Since the only difference between HEALER and HEALER- is whether it leverages relation learning, we can deduce that the relation learning algorithm is the main reason for the improvements, rather than architectural differences. This experiment result also demonstrates that relation learning's benefits outweigh its potential overheads. The overhead of the learning algorithm is very low because overhead of both static learning and dynamic learning are minimal. For static analysis, it is only executed once upon HEALER's initialization. For dynamic analysis, the analysis process is invoked only when discovering new branches, an infrequent event. The overhead for each invocation is still minimal, due to the linear complexity related to system call length. According to Figure 6, 90% of test cases have less than 5 system calls, where HEALER can learn the relation in 4 extra executions.

Figure 5 demonstrates the evolution of relations learned by HEALER in the first three hours of the experiment. We only list the data in the first three hours as the amount of relations after three hours is very large and difficult to visualize. In the first hour of the experiment, the relatively more explicit relations of system calls were learned first, thus forming multiple sub-graphs as shown in the left part of the Figure 5. As HEALER continues fuzzing, deeper relations were discovered and the sub-graphs gradually began to connect, forming the complex graph as shown in the middle and right parts of the Figure 5. We extract the KVM-related system calls from each graph and visualize the formation of their relations in the bottom half of the figure. With the learned relations, the bug mentioned in Section 3 can be triggered efficiently.

Table 2. Branch coverage statistics of HEALER compared to HEALER-

Version	min-impr	max-impr	Average	Speed-up
5.11	+30%	+45%	+38%	+2.5×
5.4	+31%	+47%	+37%	+2.2×
4.19	+20%	+35%	+27%	+2.5×
Overall	+28%	+43%	**+34%**	**+2.4×**

The Syzlang description (revision 0085e0) contains a total of 3579 system calls. Note that it contains far more system calls than there actually are in the Linux kernel, because the Syzlang description contains a large number of specialized, customized calls, which HEALER's relation learning algorithm supports. Table 3 shows the minimum, maximum and average number of relations learned by HEALER in 10 rounds of experiments in each version of the kernel. For instance, the number of learned relations in 10 rounds of experiments in Linux-5.11 is distributed between 5901 and 6820, with an average of 6320. The result is reasonable because most of the system calls in the Syzlang description are specialized calls that explicitly belong to a particular module of the kernel, therefore the graph determined by the influence relations is overall sparse and locally dense.

Table 3. HEALER's learned relations count

Version	Min	Max	Average
5.11	5901	6820	6320
5.4	4873	6462	5880
4.19	4890	5829	5434
Overall	**5221**	**6370**	**5878**

To evaluate the correctness of the relation learning algorithm, we examined the learned relations manually. Specifically, we merged the relation data from 10 rounds of experiments within the same version of the kernel. The experiment results from different versions of the kernel need to be analyzed independently, because of the differences in the internal implementation. Within each version of data, we evaluated each learned relation, e.g., C_i and C_j, by executing the output test cases that contain C_i and C_j in the

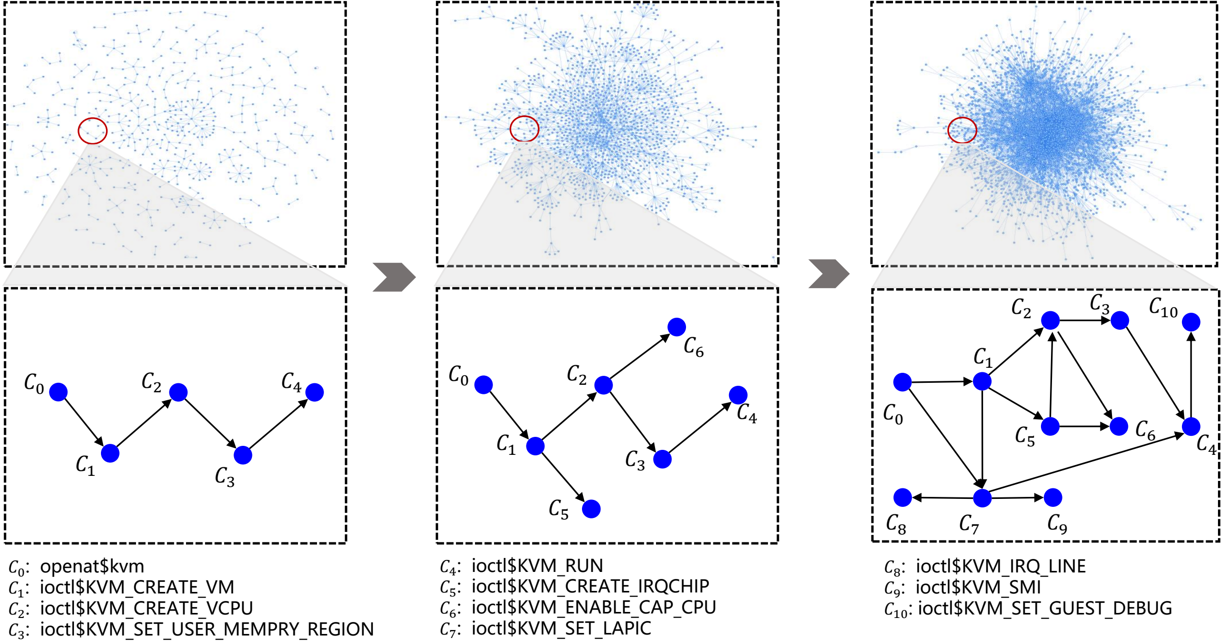

Figure 5. The evolution of the relations learned by HEALER in the first three hours. In the first hour of the experiment, the more obvious relations of system calls are learned first, thus forming multiple sub-graphs. As the fuzzing process continues, deeper relations are discovered and the sub-graphs gradually begin to connect, forming the complex connected graph. The bottom half of the figure shows how the relations between KVM related system calls evolves.

corresponding kernel, and checked if C_i has influence on C_j's execution path utilizing `gdb remote debugging`. The results of this check showed that all the relations are correct, a reasonable result given that the relation learning algorithm is based on dynamic analysis. In principle, the correctness of the learned relations is ensured by dynamic analysis between adjacent system calls, which guarantees reliability and causality, respectively. For example, when removing system call `open` in the sequence [`open`, `read`] , the change in coverage of `read` can only be attributed to the removal of its preceding and adjacent call `open`. The relation table is constructed iteratively from the aforementioned process, so it is theoretically correct.

We have shown that the relation learning algorithm plays a key role in the coverage improvement of HEALER. Here we discuss the reasons that contribute to HEALER's improvement over the state-of-the-art. In principle, HEALER uses the learned relations to guide mutation and generation, eventually producing system call sequences. We analyzed the output corpus of Syzkaller, Moonshine, HEALER- and HEALER, which consist of all minimized system call sequences. At the end of each 24-hour experiment, Syzkaller, Moonshine, HEALER- and HEALER output 6210, 5951, 6894 and 4937 call sequences on average, respectively.

Figure 6 shows the distribution of the lengths of all minimized sequences in the corpus. The length of the most sequences produced by Syzkaller, Moonshine and HEALER- is around 1 or 2. In contrast, HEALER produced a significant amount of call sequences with lengths greater or equal to 3. As shown in Figure 6, the number of call sequences that are longer than 3 in the corpus (generated test cases) of HEALER (46%) is 2.1 and 1.8 times higher than that of Syzkaller (21%) and Moonshine (25%), respectively. HEALER outputs 4 times as many sequences with lengths greater or equal to 5 as Moonshine and Syzkaller, respectively. Since only system

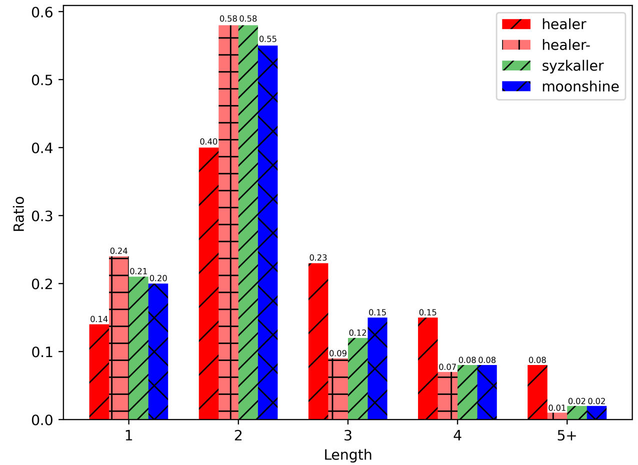

Figure 6. Distribution of the lengths of all minimized sequences output by each tool.

calls that contribute to new coverage are saved after minimization, the longer sequence might contain more types and combinations of system calls, which indicates more complicated sequences and have a greater probability of reaching deeper into the kernel. Guided by the learned relations, HEALER's generation and mutation process produces test cases with more complex combinations of system calls, thus it is able to use fewer sequences to achieve deeper coverage and trigger more vulnerabilities, which also explains the coverage improvement.

6.3 Bug Detection Capabilities of HEALER

To evaluate HEALER's vulnerability detection capabilities, we collected and compared the collected vulnerability data reported by Syzkaller, Moonshine, HEALER- and HEALER in the 10 rounds of experiments over 24 hours in each version of the Linux kernel. Over a 24-hour period, the four experiment subjects found a total of 35 vulnerabilities in the three versions of the Linux kernel, of which HEALER found 32, whereas Moonshine, Syzkaller and HEALER- found 20, 17 and 10, respectively. Note that all the 35 vulnerabilities in this 24-hour experiment are *previously-known*. Moonshine and Syzkaller found 3 vulnerabilities in total that HEALER did not find, because these 3 vulnerabilities require specific executor features that HEALER currently does not support, such as USB emulation. HEALER found 15, 18 and 22 vulnerabilities on each kernel version, respectively, that Moonshine, Syzkaller and HEALER- did not find. Table 4 lists the vulnerabilities not reported by either Syzkaller, Moonshine, or HEALER-. The column "Length to Reproduce" presents the length of the system call sequences required to reproduce the vulnerabilities. Most of these vulnerabilities require a combination of 5 or more calls to be reproduced, so Moonshine, Syzkaller and HEALER- were not able to find them efficiently. Those vulnerabilities cannot be detected by Syzkaller and Moonshine because they are related to the internal state of kernel, which can only be triggered by some relation-aware system call sequences with proper parameters.

Furthermore, we tested HEALER on more versions of the Linux kernel, including 4.19, 5.0, 5.4, 5.6, and 5.11. We found 218 unique vulnerabilities over a period of 2 weeks, 33 of which are confirmed by corresponding maintainers as previously unknown bugs. Table 5 lists the details of those previously unknown bugs. Among them, Syzkaller had reported two similar vulnerabilities, `cma_cancel_operation` and `rdma_listen` over a year ago, where the corresponding fixing patches were added and merged into the kernel upstream at that time. Since then, Syzkaller has been unable to trigger these two vulnerabilities, thus the community believed that the bugs were properly fixed. However, in our experiments, these two vulnerabilities were reproduced by HEALER with more complex system call combinations, which demonstrates HEALER's effectiveness in detecting deeply-hidden vulnerabilities. We analyzed the property

Table 4. Vulnerabilities found by HEALER while missed by Syzkaller, Moonshine and HEALER- in the 24h experiment. The column "Length" presents the length of system call sequence required to reproduce the vulnerability.

Vulnerability	Version	Length
deadlock in console_unlock	5.11	18
null-ptr-deref in put_device	5.11	8
refcount bug in l2cap_chan_put	5.11	7
null-ptr-deref nbd_disconnect_and_put	5.11	6
kernel bug in ioremap_page_range	5.11	6
null-ptr-deref in kvm_hv_irq_routing_update	5.11	6
null-ptr-deref in ieee802154_llsec_parse_key_id	5.11	5
out-of-bounds read in bit_putcs	5.4	8
kernel bug in tpk_write	5.4	6
null-ptr-deref nl802154_del_llsec_key	5.4	5
null-ptr-deref in llcp_sock_getname	5.4	5
null-ptr-deref in vivid_stop_generating_vid_cap	4.19	10
kernel bug in bitfill_aligned	4.19	9
out-of-bounds in fbcon_get_font	4.19	6
out-of-bounds in vcs_write	4.19	5

of 218 unique vulnerabilities, where 44.4% are memory errors which were detected with the help of KASAN and KMSAN, 25.9% were triggered by assertions indicating kernel logic bugs, and 11.1% are deadlock or data-race issues which are detected with the help of KCSAN. Although Syzkaller has been testing the Linux kernel continuously with large amounts of computing resources, these 33 vulnerabilities have not been reported before.

7 Case Studies

Uninitialized memory generally contain junk data with the contents of stack or heap memory. If the kernel exposes its internal data through uninitialized memory to an attacker, then this may lead to more serious consequences. For instance, if the kernel allocates a block of memory without initialization and copies that memory block to the user-space program through operations like `copy_to_user`, it may cause kernel information leakage.

With the help of KMSAN, HEALER found uninitialized data being written to disk when dumping core. On Unix-like operating system kernels, the kernel sends a signal to kill a process that performed an illegal operation. The default action of handling these signals is to terminate a process and produce a core dump file. During filling thread related information of a process (`fill_thread_core_info`), the Linux kernel allocates a block of memory without initialization with length `size` (Line 8 in Listing 2). The kernel fills the information of current process piece by piece (Lines 8 to 10 in Listing 2). However, some fields may not be written into the allocated memory (Lines 10 to 13 in Listing 2), which

Table 5. 33 previously unknown vulnerabilities detected by HEALER.

Subsystem	Operations	Risk	Version Introduced
Ext4	ext4_mark_iloc_dirty / jbd2_journal_commit_transaction	data race	5.11
Ext4	__jbd2_journal_file_buffer / jbd2_journal_dirty_metadata	data race	5.11
Ext4	__ext4_handle_dirty_metadata / jbd2_journal_commit_transaction	data race	5.11
Ext4	ext4_fc_commit / ext4_fc_commit	data race	5.11
VFS	__fput / ep_remove	data race	5.11
Network	e1000_clean / e1000_xmit_frame	data race	5.11
VFS	cdev_del	refcount bug	5.11
Rdma	cma_cancel_operation	use after free	5.11
Network	macvlan_broadcast	use after free	5.11
Rdma	rdma_listen	use after free	5.11
Network	ieee802154_tx	use after free	5.11
Network	__qdisc_calculate_pkt_len	out of bounds	5.11
TTY	n_tty_open	paging fault	5.11
Network	__build_skb	paging fault	5.11
KVM	kvm_vm_ioctl_unregister_coalesced_mmio	general protection fault	5.11
Block	blk_add_partitions	paging fault	5.11
KVM	kvm_io_bus_unregister_dev	memory leak	5.11
IO-uring	io_uring_cancel_task_requests	null-ptr-deref	5.11
TTY	gsmld_attach_gsm	null-ptr-deref	5.11
VFS	drop_nlink / generic_fillattr	data race	5.6
KVM	kvm_gfn_to_hva_cache_init	out of bounds	5.6
NFS	nfs23_parse_monolithic	memory leak	5.6
Network	rxrpc_lookup_local	memory leak	5.6
VFS	fill_thread_core_info	uninit value	5.6
Network	rds_ib_add_conn	null-ptr-deref	5.6
TTY	vcs_scr_readw	out of bounds	5.0
TTY	n_tty_receive_buf_common	use after free	5.0
Video	soft_cursor	out of bounds	5.0
VFS	io_submit_one	deadlock	5.0
VFS	free_ioctx_users	deadlock	5.0
Video	fb_var_to_videomode	divide error	4.19
VFS	fs_reclaim_acquire	inconsistent lock state	4.19
Reiserfs	reiserfs_fill_super	kernel bug	4.19

leaves that part uninitialized. As a result, several kilobytes of kmalloc'ed memory may be written to the core dump file and then read by a non-privileged user, which can be an easy way of exposing quite a large amount of kernel memory contents. This vulnerability may have existed for 12 years. Listing 2 shows the source of uninitialized memory.

HEALER can find this vulnerability since it can generate test cases with adequate amounts of randomness in addition to test cases of high quality. The generated use case with fault injection causes the kernel to kill the executor process, after which the kernel enters the core dump process, which directly leads KMSAN to capture the aforementioned uninitialized memory error.

8 Discussion and Limitations

We have demonstrated the effectiveness of HEALER. In this section, we describe some of the limitations of our current implementation and potential solutions.

The use of system call descriptions to guide mutation and generation can improve the quality of the generated test cases, as the correctness of structure and partial semantics of the parameters can be guaranteed. However, the descriptions themselves are in most cases written by kernel experts, which increases the cost of manual labor significantly, while the correctness and completeness of the descriptions can not be fully guaranteed either. In order to reduce the cost of writing descriptions, HEALER reuses the existing descriptions

Listing 2. Code that creates the uninitialized memory

```
1  static int fill_thread_core_info(struct
       elf_thread_core_info *t, ...)
2  {
3    if(...) {
4      int ret;
5      size_t size = regset_size(t->task,regset);
6      // uninitialized memory created here
7      void *data=kmalloc(size, GFP_KERNEL);
8      ...
9      if(...){
10       // uninitialized memory stored out here
11       fill_note(&t->notes[i], ..., data);
12     }
13   }
14 }
```

of Syzkaller. However, the effect to write Syzlang descriptions still need to be further reduced. One possible solution is to automatically convert the definitions in the C header files into Syzlang descriptions,. The primary goal of the converter is to preserve the original structural definition. To add more semantic information, manually modifying the translated description is necessary.

9 Conclusion

In this paper, we present HEALER to overcome one of the core challenges in kernel fuzz testing: the complexity of system calls' relations. First, we propose an effective relation learning algorithm so that HEALER can continuously collect the relations between system calls during fuzzing. Then, HEALER leverages the learned relations to guide the system call sequence generation and mutation process for vulnerability detection. We evaluate the effectiveness of HEALER on recent Linux kernels. Compared to state-of-the-art fuzzers such as Syzkaller and Moonshine, HEALER improves branch coverage by 28% and 21% and improves fuzzing efficiency by 2.2× and 1.8× to reach the same amount of coverage, respectively. Furthermore, HEALER successfully detects 33 previously unknown vulnerabilities, which have been confirmed by the corresponding maintainers. These results demonstrate that relations between system calls play a significant role in generating and mutating high-quality test cases.

10 Acknowledgment

We sincerely appreciate the shepherding from Taesoo Kim. We would like to express our deep gratitude to Mingzhe Wang and Jie Liang for their help on this work. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046, U1911401), National Key Research and Development Project (Grant No. 2019YFB1706203), the Huawei-Tsinghua Trustworthy Research Project (No. 20192000794).

References

[1] CVE-2016-8655. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655, 2016.

[2] CVE-2017-17712. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17712, 2017.

[3] CVE-2017-2636. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2636, 2017.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In David Evans, Tal Maklin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, United States of America, 2017. Association for Computing Machinery (ACM).

[5] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, May 2019.

[6] Costin Carabas and Mihai Carabas. Fuzzing the linux kernel. In *2017 Computing Conference*, pages 839–843. IEEE, 2017.

[7] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 499–513, 2019.

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

[9] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[10] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114. ACM, 2019.

[11] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[12] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[13] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[14] Google. The kernel address sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[15] Google. The kernel concurrency sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[16] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. Antifuzz: impeding fuzzing audits of binary executables. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1931–1947, 2019.

[17] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[18] Jesse Hertz. https://github.com/nccgroup/TriforceAFL, 2016.

[19] Jesse Hertz. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2016.

[20] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *IEEE Symposium on Security and Privacy*, pages 754–768. IEEE, 2019.

[21] Dave Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2012.

[22] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: anti-fuzzing techniques. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1913–1930,

2019.

[23] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[24] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[25] Jie Liang, Yuanliang Chen, Mingzhe Wang, Yu Jiang, Zijiang Yang, Chengnian Sun, Xun Jiao, and Jiaguang Sun. Engineering a better fuzzer with synergically integrated optimizations. *30th ISSRE*, pages 28–31, 2019.

[26] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 809–814. ACM, 2018.

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

[28] Shuaibing Lu, Zhechao Lin, and Ming Zhang. Kernel vulnerability analysis: A survey. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 549–554. IEEE, 2019.

[29] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):93, 2019.

[30] Valentin Manes, HyungSeok Han, Choongwoo Han, sang cha, Manuel Egele, Edward Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2019.

[31] Pedram Amini Michael Sutton, Adam Greene. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007.

[32] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[33] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[34] Bull SGI, OSDL. Linux test project. http://linux-test-project.github.io/.

[35] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. Industry practice of coverage-guided enterprise linux kernel fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 986–995, 2019.

[36] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2015.

[37] Dmitry Vyukov and Andrey Konovalov. Syzbot. https://syzkaller.appspot.com/upstream/fixed, 2020.

[38] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.

[39] Michal Zalewski. American fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl.

[40] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.