HORUS: Accelerating Kernel Fuzzing Through Efficient Host-VM Memory Access Procedures

JIANZHONG LIU, YUHENG SHEN, YIRU XU, HAO SUN, and YU JIANG*, Tsinghua University, China

Kernel fuzzing is an effective technique in operating system vulnerability detection. Fuzzers such as *Syzkaller* and *Moonshine* frequently pass highly structured data between fuzzer processes in guest virtual machines and manager processes in the host operating system to synchronize fuzzing-relevant data and information. Since the guest virtual machines' and the host operating system's memory spaces are mutually isolated, fuzzers conduct synchronization operations using mechanisms such as Remote Procedure Calls over TCP/IP networks, incurring significant overheads that negatively impact the fuzzer's efficiency and effectiveness in increasing code coverage and finding vulnerabilities.

In this paper, we propose HORUS, a kernel fuzzing data transfer mechanism that mitigates the aforementioned data transfer overheads. HORUS removes host-VM memory isolation and performs data transfers through copying to and from target memory locations in the guest virtual machine. HORUS facilitates such efficient transfers through using *fixed stub structures* in the guest's memory space, whose addresses, along with the guest's RAM contents, are exposed to the host during the fuzzer's initialization process. When conducting transfers, HORUS passes highly-structured *non-trivial* data between the host and guest instances through copying the data directly to and from the stub structures, reducing the overall overhead signficantly compared to that of using a network-based approach. We implemented HORUS upon state-of-the-art kernel fuzzers *Syzkaller*, *Moonshine* and kAFL and evaluated its effectiveness. For *Syzkaller* and *Moonshine*, HORUS increased their transfer speeds by 84.5% and 85.8% for non-trivial workloads on average and improved their fuzzing throughputs by 31.07% and 30.62%, respectively. *Syzkaller* and *Moonshine* both achieved a coverage speedup of 1.6× through using HORUS. For kAFL, HORUS improved specifically its Redqueen component's execution speeds by 19.4%.

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

Additional Key Words and Phrases: kernel fuzzing, testing, security, performance enhancement

1 INTRODUCTION

Fuzzing is a popular program testing technique that has gathered much momentum in both academia and industry due to its effectiveness and scalability. Since the number of bugs in a project usually grow exponentially with the amount of code, operating system kernels such as Linux, which comprise of code on the scale of several million lines of code or more, have no shortage of critical vulnerabilities. As kernels execute in the processor's privileged mode, triggering any of its bugs can lead to catastrophic results, including loss of data, exposure of sensitive information, unauthorized code execution, etc. For instance, a recent vulnerability labelled CVE-2021-42008 [Hutchings 2021] is classified as Slab-Out-Of-Bounds that can result in kernel memory corruption, and

*Yu Jiang is the corresponding author

Authors' address: Jianzhong Liu, liujz21@mails.tsinghua.edu.cn; Yuheng Shen, shenyh20@mails.tsinghua.edu.cn; Yiru Xu, xuyr21@mails.tsinghua.edu.cn; Hao Sun, sun-h20@mails.tsinghua.edu.cn; Yu Jiang, jiangyu198964@126.com, School of Software, Tsinghua University, 30 Shuangqing Road, Haidian District, Beijing, 100084, China.

1049-331X/2023/8-ART $15.00

https://doi.org/10.1145/3611665

when properly exploited it can ultimately result in privilege escalation. Such a vulnerability is not an exception: there have been numerous kernel vulnerabilities [Deucher 2021; Google 2021a; Lantz 2021] found and exploited by malicious parties over the past few years. Therefore, it is of paramount importance to extensively study and develop kernel fuzzing techniques to improve the safety of operating systems and to protect users and the public.

Fundamentally, fuzzers test programs through repeatedly feeding the target program with different inputs, such as raw data bytes, structured objects and network packets, and observing the program's execution for any exceptional or faulty behavior. State-of-the-art fuzzers generally employ techniques such as coverage feedback, input mutation and grammar-based generation to effectively explore the state space and find bugs on a wide variety of programs.

Kernel fuzzers generally follow the testing paradigms of userland program fuzzers. Using *Syzkaller* [Vyukov 2016] as an example, kernel fuzzers generate *system call sequences* to feed as input into the kernel. *Syzkaller* is feedback-driven and employs mutation-based and grammar-based input generation, allowing it to generate sophisticated system call sequences. While userland fuzzers run in the same machine as its target program, kernel fuzzers differ in design by runing the target kernels in an emulated environment, such as QEMU [Bellard 2005], since fully testing kernels requires allowing it to run in the processor's privileged mode. Therefore, in order to run multiple fuzzing instances at once and prevent bugs from crashing the fuzzer itself, kernel fuzzers are generally split into two components, specifically 1) a fuzzing manager running on the host machine with a stable kernel, and 2) fuzzer instances running on guest virtual machines with target kernels. The manager performs the following operations: synchronizing global fuzzing statistics and information between all guest fuzzing instances, bookkeeping global data including overall coverage and new test cases. The fuzzer instances generate (or mutate) and execute test cases, monitor the target kernel's execution status during test case execution, and synchronizing information to and from the manager process. The test cases consist of system call sequences, which are organized as highly structured data when transferred between the host process and guest instances. Many kernel fuzzers have followed a similar design path to *Syzkaller*'s to transfer sophisticated coverage data or test case inputs to and from the guest instance, such as Moonshine, HEALER, RtKaller, Tardis, etc. kAFL[Schumilo et al. 2017] is another popular kernel fuzzer optimized towards fuzzing kernels for the x64 architecture. Similar to Syzkaller, kAFL is designed with the guest kernel running in a QEMU/KVM instance and the manager process residing in the host operating system, thus it also performs data transfers from the host process to guest instances. What is different, however, is its kernel fuzzing technique, which is similar to that of AFL [lcamtuf 2013], where it feeds generated inputs as linear bytes into certain kernel buffers and tries to trigger bugs within the kernel itself. Thus, kAFL's data transfers are mainly serialized buffers rather than highly structured data, which is easier to handle compared to Syzkaller-like kernel fuzzers.

These host-guest data transfer characteristics in kernel fuzzers result in the need for data transfer mechanisms, which can be inefficient depending on its design and implementation. For instance, *Syzkaller* uses Remote Procedure Calls (RPCs) [Birrell and Nelson 1984] to invoke specific functions in the guest fuzzers. Specifically, *Syzkaller*'s fuzzer and manager send data into RPC stubs, which in turn converts the data into RPC payloads to transfer over TCP/IP networks. This method is used for a broad range of functions within *Syzkaller*, including the fuzzer sending new inputs to the manager and the manager synchronizing inputs and coverage information to the individual fuzzer instances. In our preliminary survey, these procedures take roughly more than one-third of the entire execution time during *Syzkaller*'s fuzzing campaign, which is significant as the amount of time that system call sequences are executed to actually test the underlying kernel can only account for roughly one-half of the entire campaign.

We observe that the guest virtual machine's memory pages physically reside in the host's memory. However, the host is barred from accessing the data stored within the guest's memory pages due to the process isolation mechanisms in modern systems. Therefore, if we allow the fuzzer and manager instances to communicate and transfer data through directly accessing data in another instance's memory space, we can devise methods to

transfer *non-trivial* highly-structured data efficiently by leveraging the ability to directly copy the information to and from exact memory locations in either instance, thus greatly reducing the data transfer overheads.

We use this insight to design HORUS, a kernel fuzzing data transfer mechanism that proposes such memory access procedures in kernel fuzzing, thus providing efficient and consistent data transfers between the two instances through directly accessing another instance's memory space. We explain our designs on *Syzkaller* to illustrate our approach, since the relevant mechanisms require some modifications to the fuzzer itself. HORUS facilitates efficient data transfers using the following procedures. During the kernel fuzzer's initialization process, HORUS creates *fixed stub structures* in the guest fuzzer instance. It generates *memory layout descriptions* for the stub structures and registers these structures to the host manager process over RPC. When the fuzzing campaign is underway, whenever the fuzzer or manager wishes to send highly-structured *non-trivial* data structures to the other side, they transfer the data through the stub structures using algorithms designed to conduct efficient and consistent transfers. Briefly speaking, the transfer stubs intercept the original RPC calls and offload the actual data to transfer stubs. The stubs copy the data structure's metadata and actual data, including all referenced data, into the stub structures. Stubs in the destination instance then retrieve the data from these stub structures using the corresponding layout descriptions.

We implemented prototypes of HORUS on kernel fuzzers, including *Syzkaller*, *Moonshine* and kAFL, all popular kernel fuzzers, and evaluated HORUS's effectiveness in reducing data transfer latency and improving execution throughput on recent and major versions of the Linux kernel. For *Syzkaller* and *Moonshine*, which conducts structured data transfers during fuzzing, when integrated with HORUS, their data transfer latencies decreased by 84.5% and 85.8% on average while their execution throughputs increased by 31.07% and 30.62% on average, respectively. Furthermore, *Syzkaller* and *Moonshine*'s coverage statistics achieved a speedup of 1.6× and 1.6×, and improved by 6.9% and 8.2% over 12 hours, respectively. In addition, *Syzkaller* and *Moonshine* were able to find more bugs in a limited amount of time by using HORUS, of which 5 were previously unknown and confirmed by the kernel maintainers. In kAFL's instance, HORUS increased its Redqueen component's execution speeds by 19.4%, but does not present a significant advantage overall over vanilla kAFL, which is within expectations, as kAFL mostly only performs linear buffer transfers during execution. Regardless, HORUS still demonstrates its effectiveness to improve fuzzing efficiencies for kernel fuzzers, as many kernel fuzzers conduct data transfers of *non-trivial* data structures in methods similar to that of *Syzkaller*.

In summary, this paper makes the following contributions.

- We identify that host-guest communication and data transfers in state-of-the-art kernel fuzzers incur significant overheads when transferring highly structured data, resulting in reduced performance during kernel fuzzing.
- We observe that the efficiency of kernel fuzzers can be improved by developing efficient host-VM memory access procedures, through directly accessing memory located within a virtual machine instance, thus allowing direct memory access to transfer the relevant data.
- We design and implement HORUS, a kernel fuzzing data transfer mechanism that provides efficient data transfers through efficient data transfer techniques for improved kernel fuzzing performance.
- We demonstrate that HORUS can significantly improve kernel fuzzers' execution throughput, speed of coverage growth, and bug detection, demonstrating that these data transfer procedures can be beneficial towards a kernel fuzzer's effectiveness.
- To facilitate open research, we have open-sourced the code for HORUS on Github (https://github.com/Wingtecher-OSLab/Horus).

2 BACKGROUND AND RELATED WORK

2.1 Kernel Fuzzing

Fuzzing is an automated software testing technique first proposed in 1988 by Miller et al. [Miller 1988] Fuzz testing programs, a.k.a fuzzers, generally test other programs through repeating the following procedure: 1) it generates an input using various methods, 2) it feeds the input to the target program, and 3) it monitors the program for any crashes or exceptional behavior [Serebryany et al. 2012; Serebryany and Iskhodzhanov 2009]. Since its inception, fuzzing has gained much attention from various research fields due to its effectiveness in discovering concrete bugs [Chen et al. 2019; Gan et al. 2018; Godefroid et al. 2008; Liang et al. 2018, 2022; Wang et al. 2021; Zheng et al. 2019].

A general rule of thumb in program testing is that the greater the code base, the more bugs the program contains. Operating system kernels generally have enormous code bases. Linux, for instance, recently reached 27.8 million lines of code. Hence, kernels inevitably contain a plethora of critical vulnerabilities. Many researchers have attempted to utilize fuzzing for finding kernel bugs, thus improving the kernel's overall security [Jeong et al. 2019; Kim et al. 2019; Shen et al. 2021; Shi et al. 2019; Xu et al. 2020]. Generally speaking, fuzzing a kernel involves the following steps: 1) the fuzzer runs the target kernel within a virtualized or emulated environment; 2) it feeds the target kernel with generated test cases, usually consisting of system calls sequences; 3) the fuzzer then leverages kernel feedback information such as coverage to find bugs and guide further input generation; 4) it also monitors for any exceptional behavior and reports any crashes found, typically with kernel sanitizers [Elver 2019; Ryabinin 2014].

There have been much effort in designing and improving kernel fuzzers. Using *Syzkaller* as an example, we introduce common components and procedures of a typical kernel fuzzer. *Syzkaller* is a widely-used kernel fuzzer developed by Google and has excellent vulnerability detection capabilities [Vyukov and Konovalov 2020]. By far, *Syzkaller* has successfully discovered thousands of vulnerabilities in a wide range of kernels, including Linux, Windows, MacOS, etc. *Syzkaller* at runtime consists of a manager process running in the host machine and fuzzer instances and executors running in the guest virtual machines, as shown in Figure 1. The manager is responsible for managing the fuzzing campaign, analyzing exceptions, reproducing crashes and logging fuzzing stats, etc. Fuzzer instances are spawned by the manager and perform test case generation, mutation, feedback analysis and sending test cases to the executor for testing. Executors decode the test cases and perform the system calls with the corresponding arguments. It also collects coverage information and sends it back to the fuzzer for further analysis.

On the input generation side, *Syzkaller* uses system call specifications to generate new test cases. A system call specification consists of a set of system call abstractions, where each contains a system call description and the specific information of its parameters. Before the fuzzer starts, *Syzkaller* parses all the system call specifications written in domain language and translates them into Abstract Syntax Tree (AST) representations. Fuzzers generate system call sequences also in AST format that are then packed into binary format and forwarded to the executor. Meanwhile, each fuzzer utilizes a separated thread to poll the manager to receive any new fuzzing-relevant data. Afterwards, the executor will translate the generated inputs into to actual system calls and execute it. If a test case triggers a crash or finds any new coverage, the fuzzer will send this test case back to the manager via RPC for further test case generation.

Moonshine [Pailoor et al. 2018] is an effective kernel fuzzer built on top of *Syzkaller*. It's advantage over *Syzkaller* is by proposing a distillation algorithm, which traces and further distills actual execution traces of real-world applications to obtain a set of system call sequences that allows the fuzzer to initialize with a rich corpus, thus speeding up the fuzzing process. Therefore, it can generate high-quality test case and cover more kernel code more efficiently. Healer [Sun et al. 2021] is another promising *Syzkaller*-like kernel fuzzer that promotes a relation learning algorithm which dynamically learns relations between system calls over the fuzzing

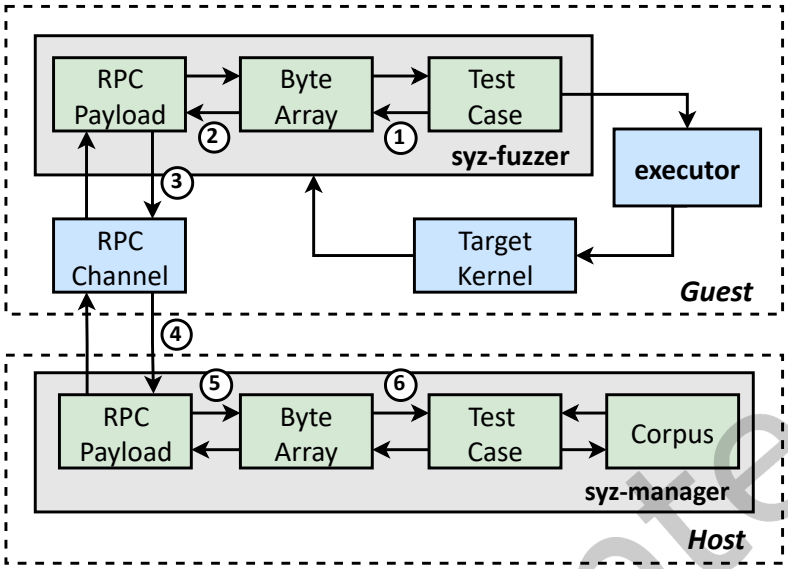


Fig. 1. Diagram of *Syzkaller*'s fuzzer instance sending a new test case to the manager process, which consists of the following six steps: ① the test case (a system call sequence) is serialized into a byte representation, ② passed to the RPC system as a payload for the manager process, ③ sent through the RPC channel via TCP/IP, ④ received by the manager RPC server, ⑤ assembled as the same byte array as in the fuzzer, and finally ⑥ deserialized as a fully structured test case. Commonly used RPC calls include `Connect()`, which establishes a connection between the manager process and a fuzzer instance; `NewInput()`, which the fuzzer process initiates to send the manager a system call sequence that triggers new kernel behavior; `Poll()`, which the fuzzer uses to ask the manager for any new inputs or information that the other fuzzer processes may have contributed; `Check()`, which the fuzzer uses during initialization to verify the parameters and versions of the testing harness.

campaign. Using this information, Healer can generate higher-quality test cases therefore achieving higher kernel code coverage. There have also been works that attempt to automatically generate system call specifications, such as [Sun et al. 2022], thus relieving kernel developers of the tedious task of crafting detailed specifications.

kAFL [Schumilo et al. 2017] is another kernel fuzzer, which differs from *Syzkaller*'s approach by utilizing an AFL-like fuzzing strategy. Specifically, it fuzzes the kernel by delivering generated payloads into certain manually designated kernel buffers. It uses mutation strategies derived from those initially engineered in AFL, such as arithmetic operations, bit and byte flips, as well as purely random generations. kAFL's major advantage lies in its effective utilization of hardware features in Intel processors such as Intel PT to collect coverage statistics, and thus minimize the fuzzing overhead to boost the fuzzing efficiency. Due to its design choices, kAFL does not require sending a highly structured input to and from the guest virtual machine instances, thus potentially lowering the data transfer overhead during runtime. However, it employs many techniques, such as Redqueen, that requires fetching a multitude of structured data from the kernel's runtime image. Redqueen accelerates kAFL's kernel coverage through fetching comparison information from the kernel under test, identify the corresponding input segments to these comparison operands, and guides the fuzzer's input generation towards directions that can possibly reach a different comparison branch of the conditional jump, thus increasing the fuzzer's coverage and potentially triggering new bugs.

Currently, kAFL is based on the Nyx [Schumilo et al. 2021] backend, which provides the facilities for its PT-based execution tracing and virtual machine management. Nyx itself is designed towards hypervisor fuzzing and augments such workloads by allowing for efficient snapshots and recoveries. Many works have been based on the foundations laid down by kAFL/Nyx, including Redqueen [Aschermann et al. 2019], GRIMORE [Blazytko et al. 2019], Nyx-net [Schumilo et al. 2022], etc.

In addition, many works aim to improve fuzzing performance by introducing hybrid execution and multi-threaded execution, like HFL[Kim et al. 2020] and Razzer[Jeong et al. 2019].

2.2 System Emulation

Full system emulation provides a virtual CPU, dedicated memory space and emulated peripherals to provide kernels and userland programs with a emulated device to run upon. Compared to user mode emulation, it can fully support running a full-fledged kernel, thus facilitating kernel fuzzing. Popular emulators include QEMU, *Bochs* [Mihoka et al. 2008], and *VirtualBox* [Dash 2013].

Full system emulators can run programs compiled for other architectures by using a technique called dynamic binary translation. Specifically, the emulator reads binary code intended for another architecture from the target program, translates it into instructions for the host machine, then executes it to emulate the target program.

In addition, for system emulation, the emulator uses a software memory management unit (SoftMMU), which manages the entire memory space required by the emulated programs. In detail, the emulator can provide all the ram and disk memory space for kernels, thus the entire memory space can be accessed through the emulator. For the emulator, it performs a virtual-to-physical address translation on each memory access operation, allowing each base block to be indexed by its physical address. Because the emulator provides a relatively independent runtime environment and memory space, it is ideal for running sophisticated bare-metal systems, like kernels. Hence many researchers use system emulators to facilitate the kernel's execution during kernel fuzzing.

2.3 Host-VM Communication

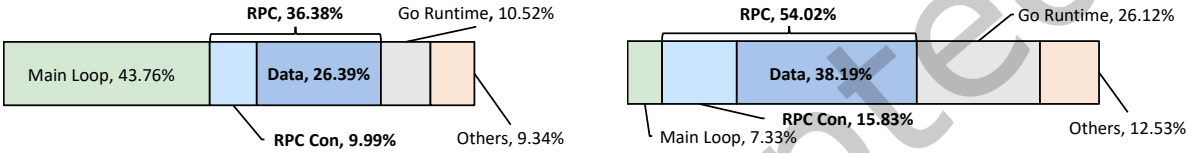
There have been much research in the field of improving communication efficiencies between the hosts and guest virtual machines. Such research has resulted in conceiving techniques such as VM-host or inter-VM memory sharing. For instance, some works have proposed to utilize the guest virtual machines' network stack, such as *VSock* [Garzarella 2020] or character devices such as Virtio [Patni et al. 2015], to facilitate such memory sharing techniques. These approaches, while being using generalized mechanisms, are not the best solution for sharing data between fuzzers running in guest virtual machine instances and manager instances. We will delve into the relevant details in the subsequent sections.

Another such technique is Virtual Machine Introspection (VMI), which allows users to control the virtual machine's behavior, as well as access its internal states, including its memory contents. Some kernel fuzzers and fuzzing techniques, such as Redqueen, utitilze VMI to access the contents of its target kernel and perform corresponding mutation strategies. However, VMI only provides kernel fuzzers with the means to access the guest memory region; Horus provides the tools to transfer data efficiently between the host and guest memory spaces.

3 MOTIVATION

Kernel fuzzers generally rely on traditional remote transmission procedures or virtual machine management techniques to communicate and transfer data with the guest instances, thus possibly incurring insignificant overheads. For example,*Syzkaller* and *Moonshine* use Remote Procedure Calls (RPC) over TCP/IP to communicate and send data between guest fuzzer instances and the host manager to synchronize fuzzing-relevant information, including new inputs, crash information, runtime statistics, etc. Using *Syzkaller* as an example, we visualize the

process of sending a test case from the fuzzer instance running in the VM to the manager instance running in the host machine in Figure 1. We observe that sending structured data from the guest fuzzer requires invoking computation and memory-use intensive operations, such as serialization and deserialization, for the data to be transferred using common methods such as TCP/IP and reach the host manager. Intuitively, using RPC to transmit data requires highly structured data structures to be encoded and decoded, incurring significant overheads for non-trivial workloads. The data that kernel fuzzers transmit to synchronize relevant information is highly-structured, i.e. containing many layers of structures, and *non-trivial*, i.e. containing references to external data. Therefore, using RPC will only exacerbate the problem. Furthermore, the fuzzer instance runs in a emulator, thus incurring even more overhead due to the instruction translation and address conversion processes during its execution. Thus, the less instructions executed during such a data transfer on the guest side, the more efficient the fuzzing process will become. This issue also affects kernel fuzzers with similar data transfer designs, where highly-structured *non-trivial* data is sent between host and guest instances.



(a) Performance profiling of *Syzkaller*'s fuzzer in the guest VM. Similar to the manager side, the fuzzer also spends a considerable amount of time on RPC-related events.

(b) Performance profiling of *Syzkaller*'s fuzzing manager. The RPC system uses a significant amount of time transmitting information to and from the respective fuzzers.

Fig. 2. Performance profiling results of *Syzkaller*'s fuzzing manager and guest fuzzer instances during actual fuzzing scenarios. The runtime is divided into the following chunks. *Main loop*: the fuzzer's main loop performs input mutation and execution, while the manager's main loop functionalities are delegated to *Goroutines*. *RPC*: execution time proportions of the entire RPC system, including encoding and decoding arguments (*Data*) and sending and receiving RPC requests and responses over TCP/IP (*RPC Con.*). *Go Runtime* represents the execution time for Goroutines and its runtime.

To quantitatively understand the severity of this problem, we broke down *Syzkaller*'s performance metrics using *pprof* [Google 2021b]. *pprof* profiles and reports each component's proportions of the entire execution time. As shown in both Figure 2a and Figure 2b, a significant proportion of *Syzkaller*'s fuzzing manager's and guest fuzzer's execution time is spent on RPC calls, thus justifying our concerns. A more detailed investigation reveals that encoding and decoding highly structured and non-trivial data such as new inputs and coverage are the root cause of this overhead.

Fuzzers can circumvent these mechanisms by making the kernel fuzzing-aware, i.e. modifying the kernel and emulator to expose an interface that facilitates direct transfers between the guest fuzzer and the host manager. However, a general rule of thumb in fuzzing is to avoid modifying the test target if possible to avoid introducing any new bugs, thus rendering this approach inadvisable. Additionally, these fuzzers can leverage OS-provided facilities such as the network stack, hardware buses and character devices to perform direct memory accesses. However, not all operating systems provide such facilities, while adapting such techniques to the respective operating systems require a non-trivial amount of human labor.

In reality, however, kernel fuzzers are merely processing and moving one memory object to another memory location, when the guest VM's pages already reside in the host's memory space, only that it is isolated by the host operating system's virtual memory mechanisms. Therefore, if we can expose the guest VM's memory to the host manager, we can perform data transfers between the guest fuzzer and host manager through directly accessing the other instance's memory space, thus offloading data movement from RPC calls.

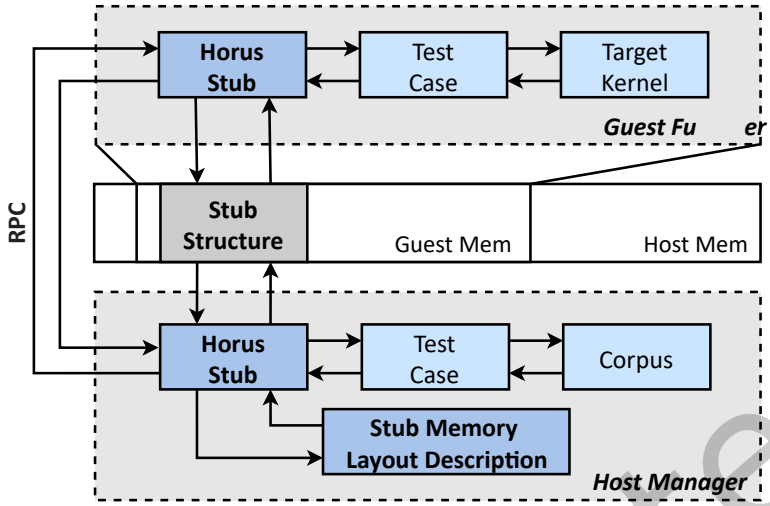


Fig. 3. Overall diagram of Horus when adopted to *Syzkaller*. Stub structures are used to place the relevant data during inter-memory-space transfers. During *Syzkaller*'s initialization process, Horus's fuzzer generates memory layout descriptions of the stub structures ①, send the descriptions to the manager ② and allow the manager to find the locations of the stub structures in the guest's physical memory space ③ (§ 4.1). When transferring data from the fuzzer to the manager during a fuzzing campaign, Horus's fuzzer-side stubs offload the relevant data ④ to the stub structures ⑤ while the manager retrieves the data ⑥ and reconstructs the corresponding data structure ⑦ (§ 4.2).

To transfer data efficiently between the host manager and the guest fuzzers, we come across the following challenges:

1. Accessing data structures in another instance's memory space correctly. We need some form of description to find relevant fields of structured data in a foreign memory space. Then, the host manager can access specific parts of the guest's memory to retrieve and send data to the fuzzer instances. First, we need to expose the guest instances' memory into the host's memory space. Then, the host manager instance needs to understand the location in the guest instance's memory of the data being transmitted to read and process the relevant data.

2. Transmitting the data efficiently and consistently to another memory space. Inter-memory-space data movement poses significant challenges towards maintaining the consistency of the target data structures. Specifically, retrieving data from a foreign memory space requires us to correctly reconstruct all structure entries and externally referenced data, while transferring data demands that we keep structural pointers and metadata intact while filling the actual contents. If not addressed adequately, this may result in situations such as loss of data, invalid pointers, incorrect amount of data transferred, etc. In addition, performance is also a significant factor in our considerations, since inefficient transfer methods will reduce the benefits of such a method.

4 HORUS DESIGN

We design Horus, a kernel fuzzing data transfer mechanism that facilitates efficient and consistent highly structured data passage between the host manager and guest fuzzers, and improves overall kernel fuzzing throughput. Horus addresses the aforementioned challenges by removing memory barriers between the host manager and guest fuzzers and providing resources and methods for efficient memory transfers. Specifically: 1) we perform modifications to existing tools and propose new primitives that allow the host manager to correctly find target data structures within the guest's memory space; 2) we offload the data transfer functionalities of

RPC calls to HORUS's stubs by designing algorithms to efficiently and consistently transfer data between the host manager and guest fuzzers.

We illustrate HORUS's design when implemented for *Syzkaller* in Figure 3. As shown in the figure, HORUS's design consists of manager-side and fuzzer-side *stubs* that transparently transfers the data to and from the relevant memory regions of *fixed stub structures*. These stub structures are created during *Syzkaller*'s initialization process and persist throughout the entire lifecycle of the virtual machine. As their locations are constant, the fuzzer sends the relevant descriptions of their layout and locations to the manager process (Section 4.1). Their purpose is to hold data corresponding to the relevant data structures being transmitted between the fuzzer and manager during fuzzing, as the manager will understand exactly where to find the data that it intends to receive or place the data it wishes to transfer. When the fuzzing campaign is underway, if the fuzzer wishes to pass data to the manager, such as in cases where the fuzzer sends the manager process new system call sequences that trigger new kernel behavior, HORUS stores these system calls and their relevant data within the corresponding stub structures. HORUS will then send the same RPC call minus the actual data to the manager process to notify the manager that the transfer of data is ready to commence. HORUS's stubs on the manager side intercept the RPC calls and transfers the data from the guest instance's memory space into the manager process's memory space using the descriptions sent by the fuzzer instance during initialization. After this is completed, the reconstructed structures are then returned to the manager for further processing, whereas in the fuzzer instance the RPC call finalizes, allowing it to continue its fuzzing operations. If the manager wishes to pass information to a fuzzer instance, instead of the fuzzer process first placing the data within the stub structures, the RPC call is first initialized, where the manager process places the data to be transferred within the guest's memory space, after which the RPC call finalizes with the fuzzer instance retrieving the target data from the relevant stub structures (Section 4.2). Though our description is *Syzkaller*-oriented, HORUS is not restricted to one single fuzzer. HORUS can be designed and implemented on kernel fuzzers that separate their fuzzing logic into distinct parts that run in both the host and guest operating systems and require transferring highly structured data between the respective instances, such as Moonshine and Healer. Adapting HORUS to other fuzzers will follow a similar approach, including identifying data transfer entities, using HORUS to create stub structures, and devise transfer routines based on the overall data structure.

4.1 Correctly Finding Inter-Memory-Space Data Structures

Syzkaller's model of communication between fuzzer and manager instances only consists of data transfer pathways. Therefore, we can expose each guest virtual machine's memory space to the manager process, allowing for direct access to each fuzzer instance's memory to transmit and receive data requested and sent from the respective fuzzers.

Since QEMU is widely used to run the fuzzer instances, we modify QEMU's guest machine RAM allocation scheme to expose the guest's memory space to the host machine. Specifically, when QEMU is initializing the guest machine's memory, instead of allocating a chunk of memory backed by anonymous pages, we redirect the allocation to shared memory and place a file descriptor under the system's shared memory directory. During *Syzkaller*'s initialization process, after successfully booting the guest virtual machine, *Syzkaller*'s manager maps its RAM section, backed by shared memory, into its own memory space.

Now that the host manager is able to access the guest fuzzer's physical memory, we need to inform the host manager of the source or target data structure's locations in the guest's physical memory. The data structures used in *Syzkaller*'s synchronization process are highly-structured and non-trivial. We denote highly-structured to be data structures with multiple levels of hierarchies, while non-trivial data structures are those with external references, such as pointers pointing to buffers in another memory location. The *memory layout description* of these data structures needs to address the following issues.

First, for highly structured non-trivial data structures, transferring between the fuzzer and manager instances requires moving and reconstructing not only the structure or array itself, but also all references that the member values may point to, in addition to fixing the references in the *non-trivial* data structures for the target memory space. For instance, the `Input` struct, defined in the module *syzkaller/rpctype*, is frequently used in RPC call transfers to represent a test case. We show its structure in Figure 4, where we see that the member variables reference other buffers which contain the actual data. Apparently, copying the `Input` struct itself is insufficient: we also need to copy the buffers that the fields *Name*, *Prog* and *Cover* point to. Furthermore, we need to fix the data pointer values so that they are valid in the target address space. Therefore, the description needs to encompass the memory information of the structure itself and all referenced data structures.

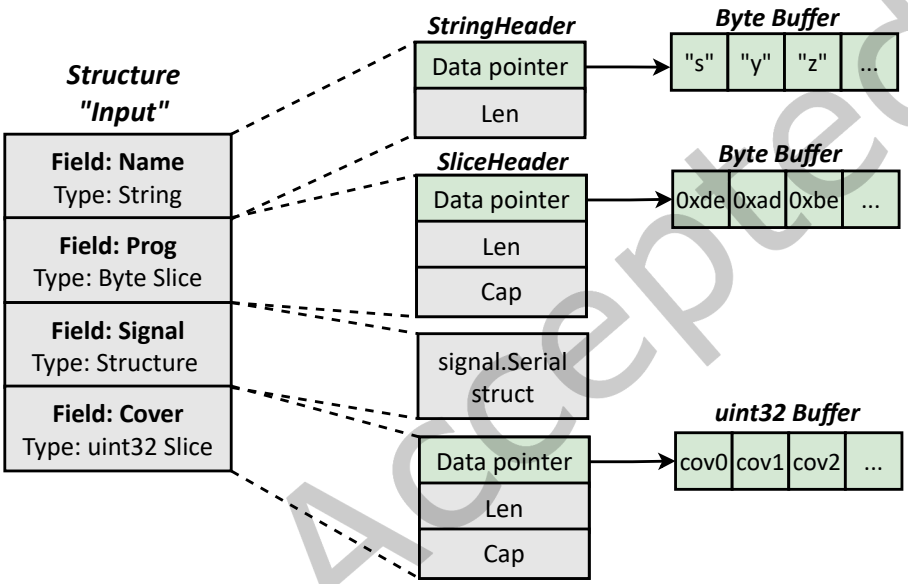


Fig. 4. The `Input` structure, a *non-trivial* data structure frequently sent in *Syzkaller*'s RPC calls. The *Prog* and *Cover* fields are Go slices while the *Name* field is a Go string, all of which maintain a `Data` pointer pointing to the actual buffer containing the actual slice or string elements. In order to send the entirety of an `Input` structure instance to another memory space, we need to copy not only the structure body itself, but also all buffers its members may refer to, and re-validate the data pointers for the target address space.

Second, virtually contiguous memory addresses may fragment when translated to physical memory addresses as a result of the state of the page tables during the instances' execution. Thus, when constructing memory layout descriptions for virtually contiguous memory areas, such as buffers, we need to determine, at each page boundary, the corresponding physical pages and describe the entire area using a sequence of physical pages.

Therefore, we design the following primitives and use them to describe the memory layout information of any data structure. First, we define the `ContiguousArea` structure to describe a contiguous physical memory area. It consists of two members, a base physical address and the length of the area. A contiguous physical memory area spanning multiple physical pages can be represented by a single `ContiguousArea` instance. This is the most basic building block for more complex structures. Next, we describe a virtually-contiguous memory chunk as `VirtualChunk`. `VirtualChunk` consists of a sequence of `ContiguousArea` instances, thus representing virtual memory chunks correspond to one or multiple physical memory areas. Now, we can describe the memory

layout of structures and arrays. Arrays can be represented simply as a `VirtualChunk`, since they only consist of a contiguous memory chunk. *Self-contained* structures, i.e. structures that have no additional references, can also be represented using a `VirtualChunk` instance. Finally, we can describe *non-trivial* data structures using combinations of the aforementioned description primitives. Specifically, the description first contains a `VirtualChunk` that describes the memory layout of its own structure body. All referenced data structures can be represented as instances of the aforementioned primitives, such as member arrays using `VirtualChunk` instances.

Using the *Syzkaller*'s `Input` structure as an example, we briefly cover the components of its memory layout description. First, a `VirtualChunk` instance *StructMem* describes the memory location of the struct body. Next, we use three `VirtualChunk` instances to describe the memory locations of the individual data buffers of the member variables *Name*, *Prog* and *Cover*, respectively. Finally, the description for `Input` contains the description for the `signal.Serial` member structure `Signal`.

Given the complexity of generating such a description, it is not viable to transfer any arbitrary structure on demand. Doing so would require the fuzzer instance to iterate through the structure itself and all referenced data, find the corresponding physical memory areas, and send the description to the manager over RPC calls each time the fuzzer and manager instances need to transfer data. This can significantly reduce the overhead reduction of using direct memory copying, even when not considering the complexity of the design itself.

Thus, we propose to use *fixed stub structures* to facilitate data transfers between the fuzzer and manager instances. These structures are statically allocated during the fuzzer's initialization process and contain the same data as the corresponding structures transferred through RPC. This allows the transmitting and receiving end to know the location to and from which to move data. Therefore, we generate the memory layout descriptions of these stub structures and notify the manager with their respective descriptions during the fuzzer instance's initialization process.

In practice, we find that *Syzkaller* most frequently uses the `Input` and the `PollRes` structures during RPC data transfers. The former is described above and the latter is used to transfer synchronized inputs and coverage information from the host manager to the individual guest fuzzers. Therefore we create *fixed stub structure* instances for both structures and implement their respective memory layout descriptions.

4.2 Efficiently and Consistently Transferring Data Structures

During the fuzzer's execution process, HORUS intercepts the RPC calls used to transfer and synchronize data between the manager and fuzzer instances and offloads the data into HORUS's transfer stubs. These transfer stubs are present on both fuzzer and manager instances to facilitate the movement of data to and from the respective stub structures. Since the *fixed stub structures* reside in the guest's memory space, the manager needs to map its data to physical memory locations in order to facilitate transfers. Apparently, the methods for transferring to and from the stub structures are significantly different in the fuzzer and manager instances. Therefore, we propose the following algorithms to transfer data structures across memory space boundaries efficiently and consistently.

Transferring Data to the Manager: When the fuzzer instance transfers data to the manager instance, for instance when the `NewInput()` RPC call is invoked to send a new input to the manager, it performs the following procedure.

For the transfer stub on the fuzzer size, it first fills the corresponding *fixed stub structure* with the data it wishes to transfer through a top-down manner. Specifically, the stub first assigns all *self-contained* member variables and structures without external references, such as integers, boolean values and other nested structures bodies, with the desired values. Then, for member variables with external references, such as strings, byte arrays and slices of structures, the fuzzer copies the data in the original target buffers into the *fixed stub structure*'s corresponding data buffers. Finally, the fuzzer modifies the length metadata of the stub buffer. Specifically, Go's slices and strings are represented at runtime using a `*Header` structure from the *reflect* module. For instance, headers for string

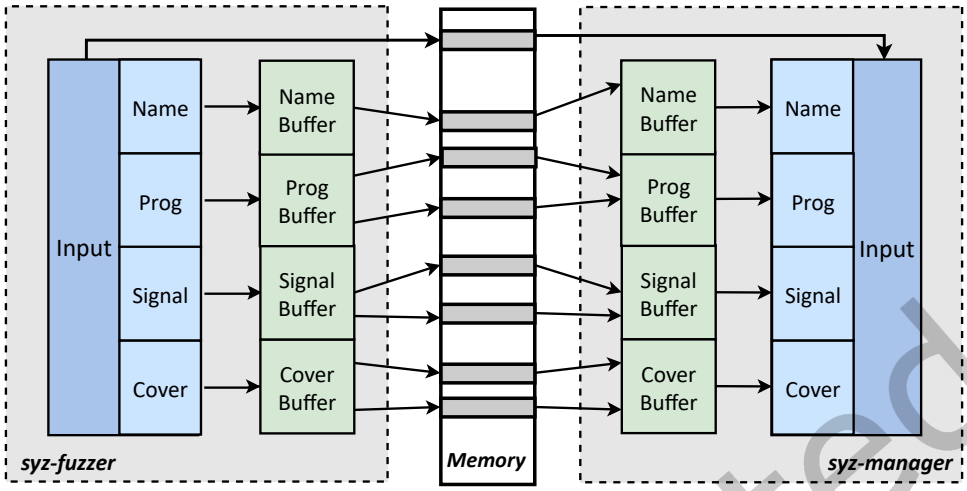


Fig. 5. Diagram of how the manager uses *fixed stub structures*' descriptions to transfer data to and from the fuzzer instance within the guest VM. The shaded segments in the memory represent the locations of the *fixed stub structure* in physical memory. A contiguous virtual memory chunk can consist of multiple physical memory chunks due to how virtual memory is mapped in the guest VM, as shown by the *Prog Buffer* and *Cover Buffer* entries in the Figure.

variables are named `StringHeader`. This is also present in Figure 4, where the headers for strings and slices both contain pointers to the actual data buffer, the length of the buffer, and in the case of slices, the capacity of the buffer. We simply change the length value to modify its metadata. For composite structures or arrays of non-trivial data structures, the stub will then recursively perform the data transfer on the corresponding structures.

We illustrate this process in the upper half of Figure 5. To fill the *fixed stub structure* for `Input` structures, since there are none *self-contained* member variables, the stub first sets all corresponding metadata values, specifically the length values for the string *Name* and slices *Prog* and *Cover*. Next, it copies all relevant buffers to the stub's corresponding buffers. This concludes the procedures on the fuzzer's side.

For the manager to recover the structure data, the manager-side stub uses the memory layout descriptions of the stub structure to identify the locations to read from in a top-down manner. Specifically, the manager first retrieves the body of the data structure from the location indicated by its memory layout description, then proceeds to recursively recreate all its references. The lengths of the slices and strings can be recovered through reading the header structures in the structure body. When a virtually contiguous memory section in the guest's memory space is physically discrete, the manager performs multiple reads from locations indicated by the sequence of `ContiguousArea` instances in the corresponding `VirtualChunk` description.

This process is illustrated in the lower half of Figure 5. To retrieve the `Input` structure transferred from the fuzzer instance, the manager first reads the physical memory chunks that contain the body of the `Input` stub structure. Then it allocates buffers to place the contents of the stub's *Name*, *Prog* and *Cover* buffers. Next, it copies the contents of the stub buffers into the corresponding buffers allocated in the previous step. Finally, the manager fixes the data pointers in the member variable's slice and string headers to produce the target structure.

Transferring Data to the Fuzzer: When the manager sends data to a fuzzer instance, for example when the manager returns the result of a `Poll()` RPC request initiated by a fuzzer instance, which contains inputs and coverage synchronized from other fuzzer instances, the manager performs the following procedure:

Similar to the aforementioned inverse process, the manager fills the stub structures with the corresponding data. However, since the stub structure is in the guest VM's virtual memory space, the manager needs to avoid modifying `Data` pointers in the header structures of strings and slices to avoid invalidating the stub structures. Therefore, the manager performs data transfers recursively using the following procedure. First, the manager copies the stub structure's body into its own memory space. Next, it modifies all *self-contained* variables to the corresponding values. Then, it sets the length metadata of all member slices and strings by modifying their respective headers. Finally, it copies the body back to the stub memory locations. The manager then recursively conducts this procedure on the referenced data structures.

To retrieve the data at the fuzzer side, the fuzzer performs a *deep copy* of the stub structure, which copies all structure variables and their referenced objects. This procedure is much more straightforward since the manager has set the length metadata of slices and strings properly while maintaining the integrity of the pointers of the actual buffers.

This procedure is also demonstrated in Figure 5 with the flow of data reversed. Specifically, the manager first assigns the length metadata of all string and slice headers. Then, the manager copies the buffers of the *Name*, *Prog* and *Cover* variables to the respective physical memory chunks of the stub buffers. Copying a buffer may require multiple memory copies, since, as explained before, a contiguous virtual memory chunk in the guest VM may not be physically contiguous. This completes the operations on the manager side. On the fuzzer side, the fuzzer simply performs a deep copy of the stub's contents, producing the transferred `Input` structure.

5 IMPLEMENTATION

Here, we discuss the implementation details regarding Horus's adaptation Horus to Syzkaller, Moonshine and kAFL, including relevant details towards the data transfer primitives used, transfer routine implementation and fuzzer-specific implementation details.

5.1 *Syzkaller* and *Moonshine*

We implemented Horus for *Syzkaller* and *Moonshine*. As *Moonshine* re-uses most of the RPC primitives from *Syzkaller*, we implement Horus for both fuzzers using the methods introduced in the previous section. We modified the manager and fuzzer's RPC mechanisms to conduct transfers using Horus's transfer mechanisms.

Specifically, as aforementioned, Horus intercepts the data transfer process of `NewInput` and `PollRes` structures, as these are the most used during fuzzing. To facilitate Horus's transfer mechanisms, we inserted interception routines before the actual RPC invocations in both the manager and fuzzer processes. Specifically, we replaced relevant RPC calls that contains data sections with ones that exclude the relevant fields. When the fuzzer or manager process wishes to send an RPC call, Horus fills the fixed stub structures with the relevant data and invokes the relevant calls. When the receiving side processes this request, instead of extracting the data from the RPC request itself, it calls the Horus routines to recover the relevant data from the specified locations.

When allocating fixed stub structures, Go's memory allocator *lazily* allocate pages to contain the structures, i.e. the pages are not allocated immediately and fully, but only when they are first accessed. Therefore, during the initialization process, Horus traverses all allocated pages to ensure that they are properly allocated, allowing both the host and guest to use the memory pages properly.

5.2 kAFL

While as aforementioned, kAFL mainly transfers linear data buffers to kernel buffers in the guest instance, thus potentially not having an overhead comparable to that of *Syzkaller*, we also implemented Horus for kAFL to understand the effects of Horus on these kernel fuzzers. kAFL mainly performs host-VM data transfers in the following two scenarios. First, the kAFL *worker*, which is the equivalent of the fuzzer component in *Syzkaller*,

transfers inputs, which are linear byte buffers, to the guest instance's predetermined byte buffers upon each execution cycle. In contrast, *Syzkaller* and *Moonshine*'s inputs consist of *non-trivial*, highly structured data, which encapsulates an entire system call sequence with all argument types and contents. Second, during the Redqueen stage of a given input seed's fuzzing process, the worker retrieves comparison information extracted through hooking to each comparison instruction on the execution trace. It then identifies possible input positions that can influence the result of a comparison expression and guide input generation towards reversing the comparsion result.

In contrast to *Syzkaller* and *Moonshine*, where the fuzzer invokes manager routines through the use of RPC calls, in kAFL, the *agent*, which executes the payload on behalf of the worker in the target kernel, performs such duties through the use of *hypercalls*. Therefore, when adapting Horus to kAFL, the corresponding interception is performed before and after such hypercalls.

We implement Horus for the aforementioned two scenarios. For the former scenario, where the worker is transferring generated inputs to the guest instance, Horus intercepts the vanilla transfer routines and directly transfers the intercepted payload data to the location specified by the agent during initialization. For the latter scenario, when kAFL runs the Redqueen routines once for each new input before conducting traditional mutation operations, the addresses of each comparison operation is recorded into a shared memory buffer through hooks inserted into QEMU's execution stream. When the agent finishes its execution, Horus on the worker side fetches the operands using methods similar to that discussed in the previous section from each comparison operation and returns the values for further use in Redqueen's logic.

6 EVALUATION

We evaluate Horus's effectiveness when adapted to kernel fuzzers *Syzkaller*, *Moonshine* and kAFL. To analyze and understand Horus's performance improvements over the original approaches, we propose and strive to answer the following research questions:

- **RQ1**: Can Horus transfer fuzzing-relevant data faster than *Syzkaller*'s and *Moonshine*'s RPC mechanisms and kAFL's Nyx-based transfer mechanisms?
- **RQ2**: Can Horus achieve a empirically significant execution throughput improvement compared to using RPC in *Syzkaller* and *Moonshine* and using Nyx in kAFL?
- **RQ3**: Does Horus assist kernel fuzzers achieve the same coverage as non-Horus versions faster?
- **RQ4**: How does Horus affect the kernel fuzzer's abilities in finding kernel bugs?

To answer these research questions, we designed the following experiments.

For *Syzkaller* and *Moonshine*, we first probe measure the round-trip-time (RTT) of sending a new input from the fuzzer to the manager when using Horus's mechanisms and *Syzkaller*'s original RPC systems; then, we profile the processor execution time proportions for each component in both the fuzzer and manager instances during fuzzing to acquire the reduction in data transfer overhead through using Horus and examine whether Horus can improve the fuzzer's execution throughput; next, we run *Syzkaller*, *Moonshine*, *Syzkaller*+Horus and *Moonshine*+Horus for 12 hours to compare their coverage statistics; finally, we examine the number of kernel bugs found through using Horus.

For kAFL, we first measure the average time for an input to be transferred to the agent under Horus and kAFL's Nyx backend as well as the average time for Horus and Nyx to recover Redqueen's required operands; we then collect the coverage statistics of kAFL with Horus and with Nyx over a period of 12 hours.

6.1 Experiment Setup

We conducted our experiments with *Syzkaller* and *Moonshine* on a server with an AMD EPYC 7742 64-Core processor, 512 GiB of memory and running 64-bit Ubuntu 20.04.2 LTS. The tested kernels are the mainline, stable

and most-recent long-term versions, which, at the time of writing, are 5.16, 5.15 and 5.10, respectively. The kernels are compiled with Kernel Address SANitizer (KASAN) enabled to detect any kernel crashes. The fuzzers obtain kernel coverage using KCOV. We augment *Syzkaller* and *Moonshine* with Horus, which we refer them as *Syzkaller*+Horus and *Moonshine*+Horus, or *Syz+Hor* and *Ms+Hor* for short. For the comparison experiments, we configure the virtual machine instances for the four fuzzers with identical parameters. Specifically, each fuzzer instance has two virtual machine instances with 4 GiB of memory and two processor cores each. Each set of experiments is repeated 5 times where each individual experiment is executed for 12 hours on a dedicated core.

Our experiments on kAFL are conducted on a server with an Intel Xeon Silver 4210R 20-core processor with 32GiB of memory and running 64-bit Ubuntu 20.04.2 LTS. The host system's kernel is replaced with Kernel 5.10.73-kafl+, the patched version provided by kAFL, which integrates Intel PT support for KVM. The kernel under test is Linux 5.15-4 with the kAFL agent installed as `/arch/x86/kernel/kafl-agent.c`. The kernel is compiled with KASAN support to detect any address violations triggered by kAFL. Each experiment is run with identical parameters and allocated the same amount of resources. Specifically, each kAFL instance is alloted with 1GiB of memory and 1 dedicated CPU thread. Each set of experiments is run for 12 hours. A total of 5 experiments were executed.

All statistics are then verified through the Mann-Whitney U test, as per fuzzing guidelines laid down by Klees et al [Klees et al. 2018], to determine whether there exists statistical differences between the sets of data. The relevant statistics are shown in Table 4, where we analyze the values for each entry in their respective sections.

6.2 Data Transfer Speed

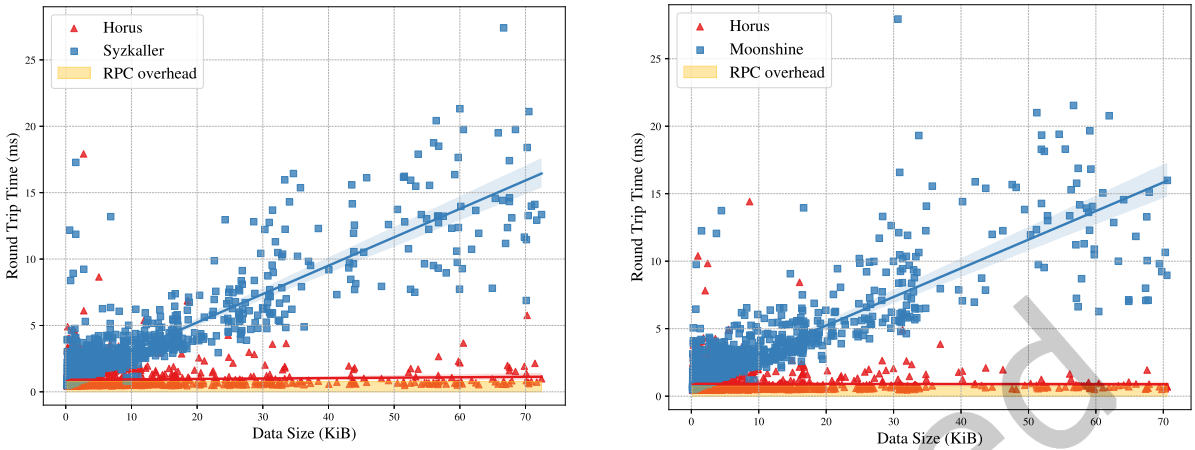
To answer **RQ1**, we measure the time it takes for kernel fuzzers to complete a data transfer operation.

6.2.1 Syzkaller and Moonshine. We measure the round trip time statistics for data transfers between the manager and fuzzer instances in order to understand the data transfer speed speedups of Horus. The data transferred can be of different sizes, since system call sequences can have a varying number of system calls and length of the arguments, therefore, we also need to take into account the size of the transferred data in relation to its round trip time.

We measure these statistics by inserting timer probes before initiating and after the conclusion of the relevant RPC calls, including the relevant calls to Horus's stubs. Since RPC calls itself have a considerable overhead, we also implemented an empty RPC call in the manager's RPC server and measured its round trip time when called from the fuzzer instances.

The relevant results are shown in Figure 6. As we see in the graph, the regression lines have significantly different slopes, indicating that Horus allows both *Syzkaller* and *Moonshine* to achieve a significant speedup in round trip time statistics. On average, *Syzkaller* and *Moonshine* achieve a 34.0% and 33.5% speedup in round trip time statistics when using Horus. However, these numbers include the time for an RPC call itself, thus, when the amount of data transferred is relatively small, the benefits of using Horus are overshadowed by the RPC call's overhead itself. To obtain a clear picture of the actual speedup of Horus, we take the base time for a RPC call into consideration. Considering that Horus's round trip time statistics for data transfers under 15KiB have negligible difference to the RPC call itself, we filter out data with a length of less than 15KiB to assess the actual speed of data transfers. In this case, Horus achieves a speedup of 84.5% and 85.8%.

As shown in Table 4, we have tested these statistics using the Mann Whitney U test and found that the p values for *Syzkaller* and *Moonshine* are approximately 0.007 and 0.006, both less than 0.05, thus indicating that Horus's performance is statistically significant than that of vanilla *Syzkaller* and *Moonshine*. Thus, we can conclude that, for *Syzkaller* and *Moonshine*, Horus is able to transfer data significantly faster than the RPC mechanisms provided by traditional fuzzers.



(a) *Syzkaller* vs. *Syzkaller*+Horus. (b) *Moonshine* vs. *Moonshine*+Horus.

Fig. 6. Round trip time statistics of sending different sized data structures between the fuzzer and manager instances. We plot the regression lines for each dataset and shaded its 99% confidence intervals. The average overhead of the RPC call itself is shown as a yellow area near the x-axis.

Table 1. Latency statistics of kAFL with Horus vs. vanilla kAFL. The transfer latency during execution does not exhibit a significant difference, while Horus is capable of accelerating Redqueen's retrieval efficiency.

Fuzzer Configuration	kAFL/Horus	kAFL/Nyx
Payload Transfer Latency (ms)	4.62	4.58
Redqueen Retrieval Latency (ms)	138.6	172.1

6.2.2 kAFL. We measure the average delay of using Horus and Nyx for performing data transfers both during executing a new input and when fetching comparison operands for Redqueen. The results are shown in Table 1.

As is evident in the chart, Horus unfortunately does not exhibit a significant difference when transferring input data to the agent during fuzzing. We believe that this is due to kAFL's input being linear in nature, unlike the structured inputs in Syzkaller, therefore Horus's designs that facilitate efficient structured data transfer does not offer any observable advantage. Our statistical tests tell the same story, with the p value in the Mann Whitney U Test being significantly greater than 0.05.

However, for the Redqueen scenario, Horus exhibits an average latency of 138.6 while kAFL/Nyx has 172.1, thus reducing latencies compared to vanilla kAFL by 19.4%. We believe that this demonstrates Horus's effectiveness when encountering structured data between the host and guest instances, as multiple offsets in a memory image is similar to that of a simple structure. The statistical tests show a p value of around 0.042, which indicates statistically significant differences between the data.

6.3 Execution Throughput

To answer **RQ2**, we measure the overall execution throughputs of each fuzzer configuration and determine whether Horus delivers an uplift to their fuzzing performances.

6.3.1 Syzkaller and Moonshine. We first evaluate and compare the execution throughput of *Syzkaller* and *Moonshine* when using HORUS or RPC to perform data transfers. To this end, we gathered the number of executions of each kernel fuzzing trial over a period of 12 hours. For each fuzzer, we averaged over the total number of executions. The relevant results are shown in Table 2. As listed in the table, HORUS assists *Syzkaller* and *Moonshine* to achieve an execution throughput speedup of 31.07% and 30.62%, respectively. Statistical significance is established, as shown in Table 4, where the *p-values* of both Syzkaller and Moonshine's evaluations are below 0.05.

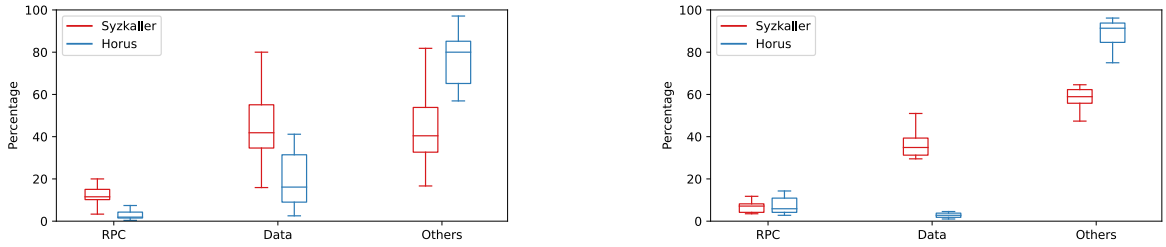
Table 2. Execution count and throughput statistics over 12 hours of *Syzkaller*, *Syzkaller*+HORUS, *Moonshine* and *Moonshine*+HORUS for the respective Linux kernel versions.

Fuzzer	**5.16**	**5.15**	**5.10**	**Average**
Syzkaller	1.62E+06	1.73E+06	1.96E+06	1.77E+06
Syz+HORUS	2.50E+06	2.47E+06	2.73E+06	2.56E+06
Syz, exec/m	2252.7	2397.9	2715.4	2455.3
Syz+Hor, exec/m	3469.1	3429.0	3788.7	3562.3
Improvement	**+54.00%**	**+43.00%**	**+39.53%**	**+45.08%**
Moonshine	1.56E+06	1.62E+06	1.89E+06	1.69E+06
Ms+HORUS	2.12E+06	2.03E+06	2.45E+06	2.43E+06
Ms, exec/m	2171.9	2246.3	2620.3	2346.2
Ms+Hor, exec/m	2949.3	2820.2	3403.8	3381.9
Improvement	**+35.79%**	**+25.55%**	**+29.90%**	**+44.14%**

To verify that HORUS is the root cause for the respective performance improvements, we then profile the runtime compositions of various functional components during *Syzkaller* and *Moonshine*'s fuzzing campaign. Similar to the preliminary analysis in Section 3, we use *pprof* to break down each fuzzer instance's runtime performance. In particular, we focus our attention on the execution time for the RPC systems and data transfer mechanisms. A significant decrease in their execution time proportions will indicate that HORUS can effectively increase kernel fuzzer's execution throughput by lowering the overhead of data transfer mechanisms.

The relevant results are shown in Figure 7a for the manager instance and Figure 7b for the fuzzer instance. Since *Moonshine*'s runtime composition is very similar to that of *Syzkaller*'s, we omitted the information from the plots for greater clarity. In this graph, *RPC* represents the execution time proportions of the RPC systems itself, such as encoding call arguments, sending and receiving on network sockets, etc.; *Data* represents the execution time proportions for *Syzkaller* to perform data transfers, using either HORUS or the RPC systems for the HORUS-improved and original versions, respectively. The statistics obtained during the preliminary analysis are also included for comparison. Obviously, the execution time proportions of the RPC systems and data transfer mechanisms both decreased significantly with the use of HORUS, demonstrating that it is HORUS that lowered *Syzkaller* and *Moonshine*'s overhead, thus increasing their overall execution throughput statistics.

We further conducted an experiment to evaluate the individual effectiveness of each component in HORUS, i.e. exposing the guest's memory space into the host and conducting transfers using HORUS's transfer techniques. The results are shown in Figure 8. As is evident in the graph, using VMI to transfer the data blobs directly results in a significant reduction in overhead compared to using RPC calls to transfer the data blobs. However the overhead still grows significantly as the size of the blob grows. In comparison, using full Horus with Syzkaller/Moonshine is significantly faster than transferring data blobs, either through RPC or VMI. The fluctuation in Horus's transfer latency is due to the variable time it takes for the empty RPC call to reach its receiver. The actual time used

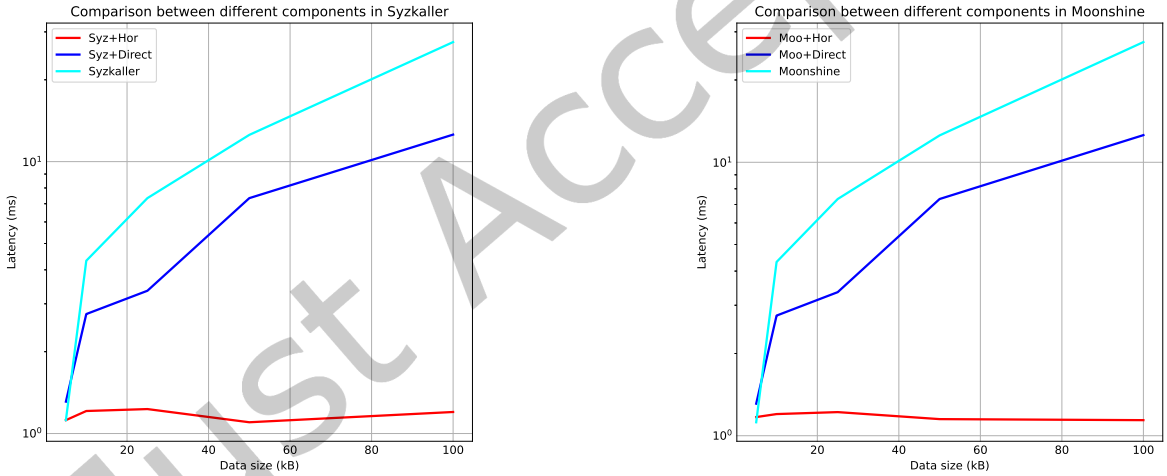


(a) Performance profiling of *Syzkaller*'s manager instances with and without Horus implemented.

(b) Performance profiling of *Syzkaller*'s fuzzer instances with and without Horus implemented.

Fig. 7. Performance profiling of *Syzkaller*'s manager and fuzzer instances. The *y*-axis represents the execution time proportions of the individual components. *RPC* represents the execution time proportions used for sending data through TCP/IP-backed RPC systems. *Data* represents the execution time proportions used to encode and decode the data for *Syzkaller* and to transfer the data to and from the stub structures for Horus.

during transferring data is negligible in comparison. Thus we demonstrate that Horus's transfer mechanisms deliver significant improvements in comparison to either using RPC or VMI to directly transfer the serialized or deserialized blobs.



(a) *Syzkaller*'s component-wise contributions.

(b) *Moonshine*'s component-wise contributions

Fig. 8. Breakdown of the individual components in Horus for both *Syzkaller* and *Moonshine*. "Syz+Hor"/"Moo+Hor" indicates Syzkaller/Moonshine using Horus, while "Syz+Direct"/"Moo+Direct" indicates Syzkaller/Moonshine transferring the RPC data blobs directly through VMI, and "Syzkaller"/"Moonshine" indicates using vanilla Syzkaller/Moonshine.

Thus, for *Syzkaller* and *Moonshine*, we can answer **RQ2** by demonstrating that Horus, through reducing the overhead for data transfers, helps state-of-the-art kernel fuzzers achieve better execution throughput.

6.3.2 kAFL. We also measured the execution throughput statistics of kAFL with Horus compared with vanilla kAFL. The statistics are given as follows: the average execution throughput of kAFL with Horus are on average

102.3 executions per second, while that of vanilla kAFL is 100.2 executions per second. The statistical testing, as shown in Table 4, presents a value greater than 0.05, therefore their performance are within error margins of each other. Unfortunately, HORUS does not exhibit an impressive improvement over kAFL with Nyx. We believe that the reason still lies within the results in the previous evaluation, where kAFL's inputs transferred to the guest instance are mainly linear buffers, therefore does not fully demonstrate HORUS's transfer techniques' capabilities. In addition, while Redqueen's transfer speed are increased, it is only used during processing new seeds, which is sparsely scattered within the 12-hour fuzzing period, whereas for the most commonly used data transfer scenario, which is transferring the input buffer, HORUS does not show significant improvement. This result is within expectations, as HORUS mainly targets the transfer of structured data, while kAFL's data transfer is, as aforementioned, linear data buffers, thus not demonstrating HORUS's full potential, as reflected in *Syzkaller* and *Moonshine*'s statistics.

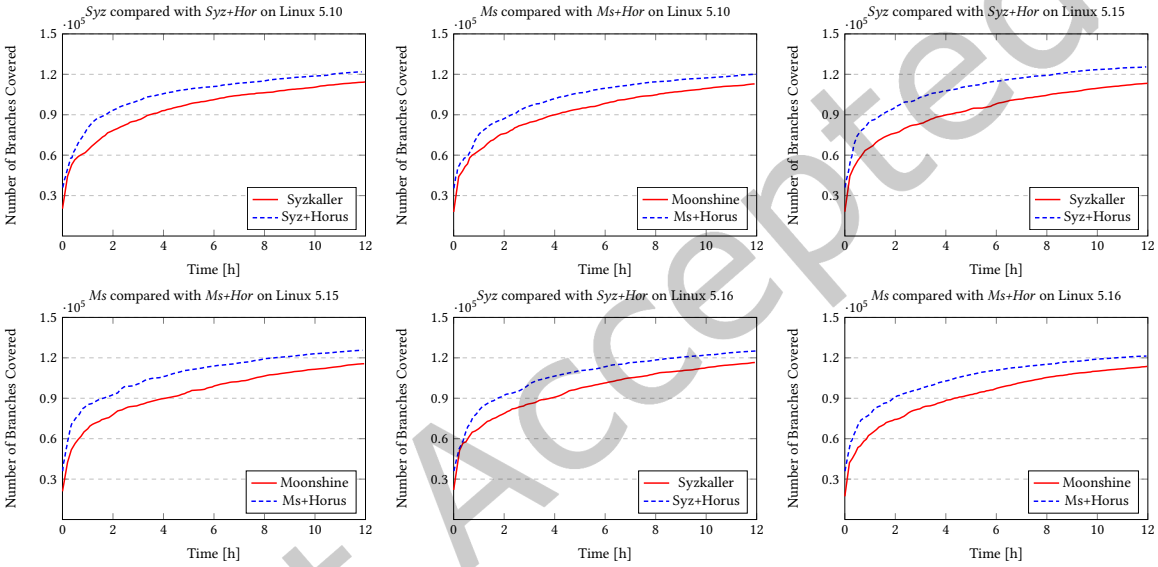


Fig. 9. Branch coverage statistics of *Syzkaller*, *Syzkaller*+HORUS, *Moonshine*, *Moonshine*+*Syzkaller* for the respective kernel versions over a duration of 12 hours.

6.4 Coverage Statistics

We address **RQ3** through examining their respective coverage growths and determining whether using HORUS allows fuzzers to achieve the same coverage in a shorter amount of time.

6.4.1 Syzkaller and Moonshine. We wish to evaluate whether the execution throughput speedup delivered through HORUS can assist *Syzkaller* and *Moonshine* in achieving better code coverage. Hence, we conduct fuzzing campaigns for *Syzkaller*, *Moonshine*, *Syzkaller*+HORUS and *Moonshine*+HORUS over a period of 12 hours on the three kernel versions. Each fuzzer's campaign is repeated 5 times. We sample their coverage statistics every 10 seconds during their respective campaigns and take the average values. The overall results are presented in Table 3, and we show the plots of coverage over the duration of their respective fuzzing campaigns in Figure 9.

As listed in Table 4, we performed statistical testing on the results, and found that the kernels 5.10 on *Syzkaller*, 5.10 on *Moonshine* and 5.16 on *Moonshine* have a p value of over 0.05, while the other three do not. Upon further

Table 3. Coverage statistics of *Syzkaller*, *Syzkaller*+Horus, *Moonshine* and *Moonshine*+Horus on the respective Linux kernel versions over a duration of 12 hours.

Fuzzer	5.10	5.15	5.16	Average
Syzkaller	1.14E+05	1.13E+05	1.17E+05	1.15E+05
Syz+Horus	1.20E+05	1.26E+05	1.22E+05	1.23E+05
Improvement	**+5.3%**	**+11.5%**	**+4.3%**	**+6.9%**
Speedup	**+1.3×**	**+1.9×**	**+1.6×**	**+1.6×**
Moonshine	1.13E+05	1.16E+05	1.14E+05	1.14E+05
Ms+Horus	1.22E+05	1.26E+05	1.22E+05	1.23E+05
Improvement	**+8.0%**	**+8.6%**	**+7.0%**	**+8.2%**
Speedup	**+1.5×**	**+1.7×**	**+1.5×**	**+1.6×**

examination, we find that the *p-values* are borderline. As the acceleration effects tends to lean towards fuzzing segments that contain frequent data transfers, we deduce that the effects can be more observable at the beginning half of the fuzzing run. Therefore for these entries, we calculated their respective p-values when considering their coverage statistics for the first 6-hours of the entire run. Thus, all six data sets yield p values of less than 0.05, demonstrating a statistically significant improvement over vanilla *Syzkaller* and *Moonshine*.

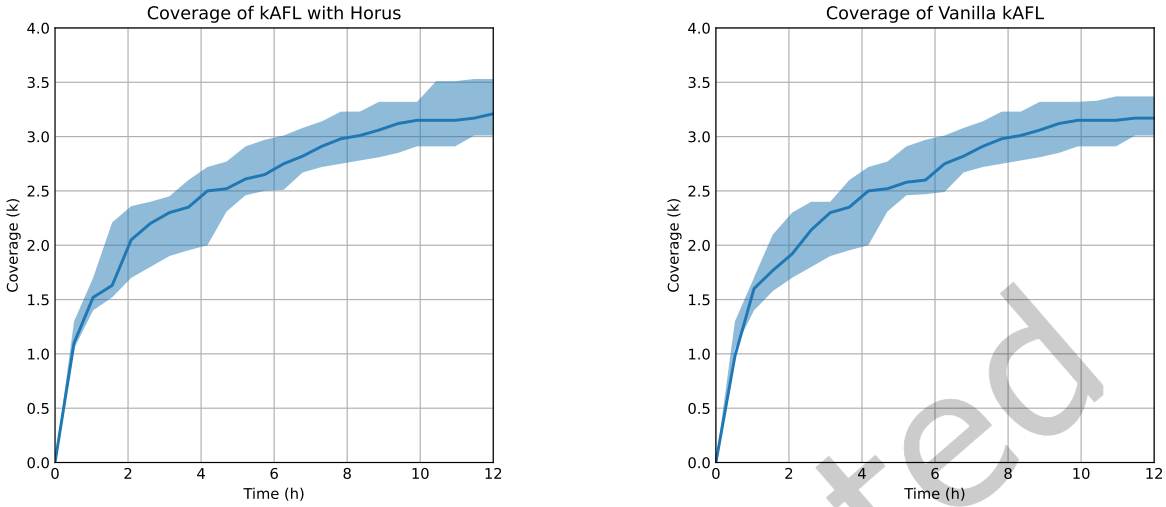
The statistics show that: for *Syzkaller*, Horus assists the fuzzer in increasing coverage statistics for the Linux kernel versions 5.16, 5.15 and 5.10 by 7.22%, 11.71% and 8.39%, respectively, with an average of 9.09%, at the end of 12 hours; for *Moonshine*, Horus increases the fuzzer's coverage statistics by 7.72%, 8.93% and 7.24%, with an average of 7.97%, respectively. However, comparing coverage statistics at the end of the campaign does not reveal the entire picture. As shown in the plot, Horus accelerates *Syzkaller* and *Moonshine*'s coverage statistics significantly before the 4-hour-mark. Furthermore, *Syzkaller*+Horus and *Moonshine*+Horus were able to achieve the same coverage as *Syzkaller* and *Moonshine* did over 12 hours significantly faster, leading to a speedup of 1.6× and 1.6× for *Syzkaller* and *Moonshine*, respectively. Intuitively, kernel fuzzers benefit from Horus's design more when they frequently find test cases that trigger new kernel behavior, thus transferring newly found inputs more often to the manager. When new interesting test cases become rare, the coverage statistics of kernel fuzzers will converge to similar points with and without Horus. We believe that Horus's mechanisms will benefit fuzzers more when they are capable of generating increasingly high quality inputs, thus being capable of exploring the kernel's state space faster and reaping the benefits of using Horus even greater.

Therefore, for *Syzkaller* and *Moonshine*, we can answer **RQ3** that Horus is able to increase the speed at which kernel fuzzers explore kernel state, allowing those fuzzers to cover kernel code more efficiently.

6.4.2 kAFL. We also examined the coverage statistics of using Horus on kAFL when compared to vanilla kAFL. The results are shown in Figure 10. As expected, due to no significant execution throughput increases observable when adapting Horus to kAFL, we do not see a significant improvement of the coverage statistics over a duration of 12 hours, which is evidenced through our statistical testing, as shown in Table 4, where the datasets yielded a p value of greater than 0.05. This is indeed as expected, as the previous two evaluations shows that Horus does not deliver a statistically significant speedup when compared to vanilla kAFL. We will delve into the relevant details in Section 7.

6.5 Bug Detection

Though Horus does not aim to improve kernel fuzzer's bug detection capabilites, we are interested in evaluating the effects of increasing execution throughput on the number of bugs found under a given time constraint. Thus,



(a) Coverage growth of kAFL with HORUS.

(b) Coverage growth of kAFL with Nyx.

Fig. 10. Coverage statistics of kAFL with Horus and with Nyx on Linux kernel 5.15-4 over a duration of 12 hours. The blue shaded areas represent the range of the values at each sample point. The blue solid line is the average coverage statistics.

Table 4. Mann-Whitney U-Test *p-values* for each statistical comparison.

Evaluation Target	**Fuzzer**	**Configuration**	***p-value***	**Significance**	**Comment**
Data transfer speeds	Syzkaller	-	0.007	✓	-
	Moonshine	-	0.006	✓	-
	kAFL	Buffer Transfer	> 0.05	×	Apparent from results
		Redqueen	0.042	✓	-
Execution throughput	Syzkaller	-	0.009	✓	-
	Moonshine	-	0.011	✓	-
	kAFL	-	> 0.05	×	Apparent from results
Coverage	Syzkaller	Kernel 5.10	0.052	×	Borderline, see below
		Kernel 5.15	0.023	✓	-
		Kernel 5.16	0.046	✓	-
	Moonshine	Kernel 5.10	0.061	×	Borderline, see below
		Kernel 5.15	0.042	✓	-
		Kernel 5.16	0.057	×	Borderline, see below
	kAFL	Kernel 5.15-4	> 0.05	×	Apparent from results
	Syzkaller-6h	Kernel 5.10	0.036	✓	-
	Moonshine-6h	Kernel 5.10	0.047	✓	-
		Kernel 5.16	0.021	✓	-

we collected the bug report data generated by *Syzkaller*, *Moonshine*, *Syzkaller*+HORUS, and *Moonshine*+HORUS over their respective fuzzing campaigns. We manually analyzed the bug reports to remove false-positives and

duplicates. The results are as follows. *Syzkaller* and *Syzkaller*+Horus found 8 unique bugs in total, of which *Syzkaller* found 5 and *Syzkaller*+Horus found all 8. *Moonshine* and *Moonshine*+Horus found 11 unique bugs in total, of which *Moonshine* found 7 and *Moonshine*+Horus found 10.

Thus, we conclude that Horus can help *Syzkaller* and *Moonshine* find more kernel bugs under a given time constraint, which adequately answers **RQ4**. We also performed kernel fuzzing using Horus for an extended period of time. Of the bugs found, we submitted reports and received confirmation for 5 previously unknown bugs, as listed in Table 5.

Table 5. Previously unknown unique bugs found by the fuzzers used during evaluation.

Operation	Bug Type	Status
svm_vm_copy_asid_from	deadlock	*Confirmed*
devkmsg_read	deadlock	*Confirmed*
add_transaction_credits	assert error	*Confirmed*
kvm_mmu_uninit_tdp_mmu	assert error	*Fixed*
usbdev_release	assert error	*Confirmed*

7 DISCUSSION

7.1 Performance on kAFL

Horus is less effective when adapted to kAFL, compared to that of *Syzkaller* and *Moonshine*. Our analysis and reasoning points us to the following reasons. First, in comparison to fuzzers that perform *non-trivial*, highly-structure data transfers between their respective host and guest instances, such as *Moonshine*, HEALER, TARDIS [Shen et al. 2022], etc., the data that is transferred between the host and guest instances in kAFL are linear data buffers that are filled into specific kernel buffers, which does not fully match Horus's design goal to facilitate efficient data transfers of highly-structured data between host and guest instances in kernel fuzzing scenarios. Second, unlike *Syzkaller* and *Moonshine*, kAFL utilizes a host-guest shared memory buffer to transmit relevant data, which is a functionality provided by its Nyx backend, while *Syzkaller* and *Moonshine* rely on slower RPC calls. Nevertheless, Horus still provides a performance uplift in the Redqueen component, where it transfers structural data, thus demonstrating Horus's effectiveness. Furthermore, many kernel fuzzers are also designed with data transfer requirements similar to that of *Syzkaller*, and thus can utilize Horus's data transfer methods to improve overall fuzzing efficiencies.

7.2 Applicability to Other Kernel Fuzzers

Our design, as shown in Section 4, is based on *Syzkaller* due to the requirement of tight integration imposed by the specific *fixed stub structures* required, as well as the interception of the original data transfer procedures. However, this method is widely applicable to kernel fuzzers that use the manager-fuzzer model, which requires constant data transfers between the two components to synchronize test cases and kernel coverage while having high transfer overheads. To adopt to another kernel fuzzer, one should first identify the structures to pass through direct memory accesses and the relevant transfer procedures to intercept. Then they can implement *fixed stub structures* and the relevant memory layout descriptions. Finally, the they can reuse the QEMU patch and facilitate transfer operations to that of Horus.

7.3 Guest Physical Memory Consistency

Currently, our design does not support page swapping in the guest VM. Swapping occurs when the kernel has insufficient physical memory pages to map newly requested memory and must swap some infrequently used pages to disk. This results in invalidating the *fixed stub structures*' physical memory descriptions, leading to incorrect data transfers by using Horus. Currently, we have disabled page swapping in our VMs to avoid this scenario.

Furthermore, the *fixed stub structures* are allocated with constant capacities. However, as the fuzzing campaign progresses, the system call sequences generated may become more complex, thus there may be a point after which the data that needs to be transferred using Horus can no longer fit within the stub structures' buffers. To mitigate this, we can reallocate the stub structures when their capacities are too small and re-register the memory layout on demand to the manager process.

7.4 Implications on Kernel Fuzzer Design

There are further design implications for kernel fuzzers when utilizing direct memory access to the memory space of guest virtual machine instances. In order for kernel fuzzers to efficient, they need to 1) retain as little code in the emulated guest machines as possible, 2) reduce inter-process communications as much as possible, and 3) leverage efficient IPC primitives when such communication is inevitable. Using *Syzkaller* as an example, it places the fuzzer in the guest virtual machine, as it frequently passes to-be-executed programs to the executor for system call sequence invocation, while sending new inputs to the manager instance occurs relatively sparse, thus allowing the fuzzer and executor to pass information through a shared memory region while leveraging RPC for communication between the fuzzer and manager instances. However, there is the possibility of moving the fuzzer's functionality into the host machine, thus allowing the fuzzer instances to run at native speed rather than the reduced emulated speed due to the emulation overhead. Utilizing the insights of Horus, the fuzzer can communicate directly with the executor via shared memory, while removing RPC entirely with other efficient IPC primitives such as pipes. By making such adjustments, kernel fuzzers such as *Syzkaller* can conform to the aforementioned three principles and yield more performance during fuzzing. As this is a research problem on its own, i.e. distributing the workload to maximize fuzzing efficiency, we believe that this is another topic that should be conducted as future work.

8 CONCLUSION

In this paper, we identify that data transfer overheads in kernel fuzzers affect their effectiveness in covering kernel code and detecting potential vulnerabilities. We propose Horus, a kernel fuzzing data transfer mechanism that mitigates the synchronization overheads present in kernel fuzzers such as *Syzkaller* and *Moonshine* by circumventing their original data transfer mechanisms over remote procedure calls. In doing so, our approach provides a more efficient solution towards efficiently transferring highly-structured data between host to the guest instances. Specifically, Horus exposes the guest fuzzer's memory space to host manager and sets up fixed stub structures in the fuzzer instances during their initialization processes. Horus then registers the stub structures with the manager to allow for efficient transfers. When conducting transfers, Horus's stubs in both the manager and fuzzer instances use the stub structures to pass highly-structured and non-trivial data consistently and efficiently. Our evaluation shows that Horus improves data transfer speeds by 84.5% and 85.8% , as well as execution throughput by 31.07% and 30.62% for *Syzkaller* and *Moonshine*, respectively. In addition, Horus allows *Syzkaller* and *Moonshine* to achieve a speedup of 1.6× and 1.6× and increases their coverage statistics by 6.9% and 8.2% over a period of 12 hours, respectively. On kAFL, Horus decreases the data transfer latency of its Redqueen component by 19.4%. To facilitate open research, we have open-sourced Horus on Github (https://github.com/Wingtecher-OSLab/Horus).

ACKNOWLEDGMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003).

REFERENCES

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.* 2, 1 (feb 1984), 39–59. https://doi.org/10.1145/2080.357392

Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002.

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

Pradyumna Dash. 2013. *Getting Started with Oracle VM VirtualBox*. Packt Publishing.

Alex Deucher. 2021. CVE-2021-42327. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42327.

Marco Elver. 2019. The Kernel Concurrency Sanitizer. https://patchwork.kernel.org/project/linux-kbuild/patch/20191017141305.146193-2-elver@google.com/.

S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696.

Stefano Garzarella. 2020. vsock. https://devconfcz2020a.sched.com/event/YOwb/vsock-vm-host-socket-with-minimal-configuration.

Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. *SIGPLAN Not.* 43, 6 (June 2008), 206–215. https://doi.org/10.1145/1379022.1375607

Google. 2021a. CVE-2021-1048. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-1048.

Google. 2021b. pprof. https://github.com/google/pprof.

Ben Hutchings. 2021. CVE-2021-42008. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42008.

Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *NDSS*.

Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 147–161. https://doi.org/10.1145/3341301.3359662

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

Patrik Lantz. 2021. CVE-2021-44733. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44733.

lcamtuf. 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. 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*. ACM, 809–814.

Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.

Darek Mihoka, Stanislav Shwartsman, and Intel Corp. 2008. Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure. In *Emerging Technologies for Information Systems, Computing, and Management*.

Bart Miller. 1988. CS 736: Project Assignment Lists. http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf.

Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. Moonshine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 729–743.

Sagar Patni, Jobin George, Pratik Lahoti, and Jibi Abraham. 2015. A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*. 6–9. https://doi.org/10.1109/NGCT.2015.7375072

Andrey Ryabinin. 2014. The Kernel Address Sanitizer. https://lwn.net/Articles/611410/.

Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. 2597–2614.

Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 166–180. https://doi.org/10.1145/3492321.3519591

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. 2021. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 83 (sep 2021), 22 pages. https://doi.org/10.1145/3477014

Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. 2022. Tardis: Coverage-Guided Embedded Operating System Fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022), 1–1. https://doi.org/10.1109/TCAD.2022.3198910

Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. 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*. 986–995.

Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. {KSG}: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 351–366.

Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

Dmitry Vyukov. 2016. syzkaller. https://github.com/google/syzkaller.

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

Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. {RIFF}: Reduced Instruction Footprint for {Coverage-Guided} Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 147–159.

Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

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