NetPanic: The Attack Surface You Can't Syscall

Tianshuo Han[1,2], Zong Cao[3], Zhen Dong[4], Xiapu Luo[5], Zhenyu Song[1,2], Jian Liu[1,2]
[1]*Institute of Information Engineering, Chinese Academy of Sciences*
[2]*School of Cyber Security, University of Chinese Academy of Sciences*
[3]*Imperial Global Singapore* [4]*Fudan University* [5]*The Hong Kong Polytechnic University*
{*hantianshuo, songzhenyu, liujian6*}*@iie.ac.cn*
z.cao@imperial.ac.uk zhendong@fudan.edu.cn csxluo@comp.polyu.edu.hk

***Abstract*—The Linux kernel network stack exposes a critical remote attack surface, yet kernel security research has historically focused on the local attack surface, especially the post-breach local privilege escalation attacks. This has left a direct, pre-authentication attack vector dangerously overlooked. To address this gap, we present the first systematic, fuzzing-based audit of this attack surface.**

Our work first identifies the unique challenges that render existing fuzzers ineffective for this task: Extreme Input Complexity and the challenge of Dual-channel Input Coordination. To overcome them, we present NetPanic, a novel fuzzer built on a Fuzzer-in-the-Middle architecture. This design inherently solves the coordination challenge by orchestrating real communication between two network stack instances. Simultaneously, it provides a stream of valid packets that serves as a high-quality baseline for our execution-guided structure-mutation strategy, which addresses the input complexity challenge.

Our evaluation of NetPanic on the latest Linux kernel yielded significant results. It discovered 15 new, remotely triggerable vulnerabilities, none of which could be found or reproduced by the state-of-the-art kernel fuzzer Syzkaller. In direct comparison, NetPanic demonstrated vastly superior performance, achieving over 100 times the execution throughput and an average code coverage improvement of over 400%. This performance gap, combined with a targeted ablation study, provides a dual validation: it confirms the correctness of our insight in identifying the unique challenges and proves the effectiveness of our solutions in addressing them. Our work provides concrete evidence that the kernel's remote attack surface is a potent and immediate threat, and we provide the community with the first effective methodology and a practical tool to begin securing it.

1. Introduction

The Linux kernel is a cornerstone of modern computing, ubiquitous across a vast ecosystem of servers, desktops, and embedded systems. Its security is therefore paramount, as a single failure can cascade into system-wide outages. The network stack is one of its most critical components, implementing foundational protocols like TCP/IP and application-layer services such as NFS and KSMBD that enable all modern network communication.

Given this criticality, securing the kernel is a primary focus of the research community. However, these efforts have historically concentrated on the *local attack surface* to discover [1], [2], [3], [4], [5], [6] and exploit [7], [8], [9], [10], [11], [12] local-only vulnerabilities. The prevailing threat model assumes an attacker has already gained low-privileged access and seeks to exploit a vulnerability for privilege escalation. This paper, however, addresses a more direct threat that has been dangerously overlooked: the *remote attack surface*. This attack surface is characterized by **Remotely Triggerable Vulnerabilities** that can be triggered by sending crafted packets externally, requiring no prior local system access. As we analyze in detail in Section 3, such vulnerabilities pose a fundamentally higher security risk than local-only bugs, with potential outcomes ranging from kernel panics to full remote code execution.

The network stack is the primary embodiment of this remote attack surface, yet it has remained largely unaudited by systematic fuzzing. Although numerous kernel fuzzers and network protocol fuzzers exist, none can systematically target this surface: kernel fuzzers like kAFL [13] lack the ability to inject external packets entirely; Syzkaller [4], despite offering a limited external packet injection extension [14], is fundamentally constrained by the challenges we identify below; and protocol fuzzers like AFLNet [15], Nyx-Net [16], and TCP-Fuzz [17] target user-space protocol implementations and cannot instrument or reach kernel-level code paths (Section 2). Our central insight is that this gap stems from two unique challenges that render existing fuzzing approaches ineffective: the **Extreme Input Complexity** of multi-layer packets and the **Dual-Channel Input Coordination** required between system calls and network traffic. In Section 4, we analyze why these challenges cause existing approaches to fail, necessitating a new fuzzing schema.

To solve these problems, we present a novel framework based on a Fuzzer-in-the-Middle [18] design. The key to our approach is to leverage the kernel network stack itself as an expert, stateful packet generator. This design inherently solves the *Dual-Channel Input Coordination Challenge* by orchestrating a real communication session between two net-

work stack instances. This stream of valid packets, in turn, provides the baseline for a novel structure-aware mutation strategy that strikes a delicate balance between structural integrity and targeted malformation, thus addressing the *Extreme Input Complexity Challenge*. We have implemented this methodology, detailed in Section 5, in a prototype fuzzer named NetPanic, which we claim is the first fuzzer capable of systematically and effectively fuzzing the remote attack surface in the Linux kernel network stack.

Our evaluation of NetPanic on the latest Linux kernel demonstrates its profound effectiveness. We discovered **15** new, remotely triggerable vulnerabilities, all considered high-risk according to kernel security guidelines [19]. To demystify the reason for this success, we compared NetPanic against a tuned Syzkaller baseline. The results show that NetPanic achieves over **100** times the execution throughput and an average code coverage improvement of over **400%**, while Syzkaller found or reproduced none of the vulnerabilities. This performance gap, combined with our ablation study, provides a dual validation: it not only confirms the correctness of our insight in identifying the unique challenges, but also proves the effectiveness of our methodology in addressing every one of them.

In summary, our main contributions are as follows:

- We identify and analyze the unique challenges of fuzzing the Linux kernel network stack, explaining why existing fuzzing approaches are ineffective or infeasible for this target.
- We design a novel fuzzing framework based on Fuzzer-in-the-Middle design that leverages the kernel as a generator, complemented by a powerful structure-aware mutation strategy to solve the identified unique challenges.
- We present NetPanic, a prototype implementation of our methodology, and demonstrate its significant performance gains over the state-of-the-art through comprehensive evaluation.
- We discover and responsibly disclose 15 critical, remotely triggerable vulnerabilities, providing concrete evidence that this overlooked attack surface poses a severe and immediate threat and offering the first effective methodology to begin securing it.

Upon acceptance, we will release the full codebase of NetPanic under the open-source policy discussed in Section 10, to facilitate further research and security auditing of this critical attack surface.

2. Related Work

Fuzzing the kernel network stack intersects two established research areas: general-purpose kernel fuzzing and network protocol fuzzing. This section reviews the state of the art in these domains to provide the necessary context for understanding the unique challenges that we introduce in Section 4.

Kernel Fuzzing. The primary input vector for kernel fuzzing is the system call interface, which allows user-space programs to request services from the kernel. The state-of-the-art kernel fuzzer is Syzkaller [4], which has defined the dominant methodology in this field. Its ecosystem is built upon critical kernel-side infrastructure that it helped pioneer, including KCOV [20] for coverage feedback and sanitizers like KASAN [21] for memory error detection. Its effectiveness stems from Syzlang [22], a descriptive language whose core principle is to model the dependencies between system calls by specifying how the output of one call is used as an argument to another. As a general-purpose fuzzer, Syzkaller can test numerous kernel subsystems, including the network stack. To do so, it wraps external packet injection into a pseudo-syscall. The structure of the network packets is defined in Syzlang, and during execution, the fuzzer generates these packets from scratch based on this description before injecting them via a TAP [23] device. We use this capability of Syzkaller as the primary baseline in our evaluation.

Over the last decade, Syzkaller's success has been demonstrated by the thousands of bugs it has discovered [24]. Consequently, a significant amount of research has been built upon it, either enhancing its general-purpose capabilities [5], [6], [25], [26], [27], [28] or adapting it for specific subsystems [29]. Other kernel fuzzing approaches exist [30], [30], [31], [31], [32], [32], [33], [34], but they typically specialize in other areas, such as the filesystem or device drivers. kAFL [13] uses hardware-assisted feedback for OS kernel fuzzing but, like the others, lacks the ability to inject external packets into the network stack and therefore cannot reach the relevant code paths. Despite this extensive body of work, the specific challenge of fuzzing the remote attack surface of the kernel network stack remains a significant and under-explored area in the literature.

Network Protocol Fuzzing. Fuzzing the kernel network stack is also related to network protocol fuzzing. However, the vast majority of protocol fuzzers [35] target single, application-layer protocol implementations running in user-space. The typical methodology involves a generation- or mutation-based fuzzer acting as one of the communication endpoints, sending malformed packets to a client or server. Research in this domain largely focuses on challenges such as protocol state modeling and complex data generation, with notable examples like AFLNet [15] for stateful servers and StateAFL [36] for automatic state inference.

Crucially, none of these tools can systematically reach the kernel network stack. AFLNet's traffic is routed through the `SOCKET` layer, a high-level abstraction for local user-space programs that does not penetrate into the kernel protocol implementations where the remote attack surface resides. Nyx-Net [16] replaces kernel networking entirely with a shared-memory channel to accelerate user-space fuzzing, intentionally bypassing the kernel stack. TCP-Fuzz [17] targets user-space TCP implementations (e.g., TLDK) and uses the Linux kernel stack only as a behavioral reference; it cannot instrument or collect coverage from kernel-level code. While sharing some high-level concepts, protocol fuzzing and kernel fuzzing are fundamentally different domains. User-space tools are technically incompatible with

the kernel due to different mechanisms for instrumentation and coverage collection. Furthermore, the kernel's distinct execution model means that many insights from user-space protocol fuzzing are not directly applicable.

In addition to automated fuzzing frameworks, manual packet crafting tools like Scapy [37] and testing environments like Mininet [38] exist but are not designed for systematic, large-scale fuzzing.

In summary, despite the maturity of both fields, no existing tool can systematically fuzz the kernel network stack for remotely triggerable vulnerabilities: kernel fuzzers lack external packet injection, and protocol fuzzers operate entirely in user-space. To our knowledge, Syzkaller's external packet injection extension [14] is the only tool with any ability to reach this surface, yet it remains fundamentally limited by the challenges we identify in the next section. This persistent gap confirms that the remote attack surface of the kernel network stack has long been overlooked, and NetPanic provides the first effective methodology to address it.

3. Linux Kernel Attack Surfaces and Problem Scope

In this section, we provide a detailed analysis of the different kernel attack surfaces to establish the context and motivation for our work.

3.1. The Overlooked Remote Attack Surface

The vast majority of kernel security research and exploitation has traditionally focused on the local attack surface. The prevailing threat model assumes an adversary has already gained a foothold on the target system, typically as a low-privileged user, with the primary goal of achieving privilege escalation. This is accomplished by invoking system calls with carefully crafted arguments to trigger a kernel vulnerability and gain root-level control. While critical, this model presumes the attacker has already overcome the significant hurdle of initial system access.

In contrast, a remote attack targets vulnerabilities that can be triggered by an adversary over the network, without any prior access to the system. The kernel network stack serves as the primary gateway for such attacks [39]. Those vulnerabilities can lead to severe consequences, ranging from system-wide denial of service to full remote code execution. Remote attacks are significantly more dangerous than local-only attacks due to two key characteristics.

Firstly, remote attacks have **substantially lower prerequisites**. A local attack requires an attacker to first compromise an application or user account to gain a shell, which is a non-trivial initial step. A remote attack bypasses this entirely; the only requirement is network connectivity to the target. This dramatically expands the pool of potential attackers and the number of systems at risk, as such attacks can be launched from anywhere on the internet.

Secondly, remotely triggerable vulnerabilities often present **greater impact and versatility**, manifesting in several key aspects: (1) *direct denial of service*: where for many local vulnerabilities simply triggering the bug is only the first step in a complex exploit chain [7], whereas for a remote vulnerability the act of triggering it with a single malformed packet can be sufficient to crash the entire kernel, resulting in an immediate and effective system-level denial of service attack, (2) *potential for remote code execution*: where a remote vulnerability with a powerful primitive can be leveraged to achieve remote code execution, arguably one of the most severe types of security breaches as it allows an attacker to remotely gain full control of a system, and (3) *superset of threat*: where a vulnerability that can be triggered remotely can often also be triggered by a local user while the reverse is rarely true, making remote vulnerabilities inherently more versatile and therefore more dangerous.

3.2. Threat Model and Problem Scope

Threat Model. We consider a remote attacker whose only prerequisite is network connectivity to a target system. The attacker has no prior access to the system, such as a local user account or shell, and is therefore unable to execute local system calls. The adversary's sole method of interaction with the target kernel is by sending network packets. The attacker's objective is to discover and trigger vulnerabilities within the kernel network stack, leading to outcomes such as a system-level denial of service via a kernel crash, performance degradation, or creating a primitive for remote code execution. Our model encompasses both unauthenticated attackers and those who may possess valid credentials for a specific network service (e.g., an NFS [40] share), as both interact with the kernel through the same remote vector.

Problem Scope. The primary scope of this work is the *discovery* of any vulnerability that can be triggered remotely by network packets and results in a security risk. This includes, but is not limited to, bugs that cause a kernel crash, a specific service to hang, or severe performance degradation. The development of post-trigger exploitation techniques to achieve more advanced goals, such as reliable remote code execution, is considered outside the scope of this paper.

Our *primary focus* is on the hardware-independent kernel network stack, as vulnerabilities found within it have the broadest possible impact across the Linux ecosystem. However, our methodology of injecting packets inevitably exercises code paths in adjacent subsystems, such as Network Interface Card (NIC) drivers [41]. Consequently, any remotely triggerable vulnerability discovered through our fuzzing process, regardless of the specific subsystem in which it resides, will be considered a valid finding within the scope of this work.

4. Unique Challenges in Fuzzing the Linux Kernel Network Stack

To find remotely triggerable vulnerabilities, a fuzzer's basic approach is to inject malformed network packets into

the target and monitor for anomalous behavior. However, when the target is the Linux kernel network stack, this seemingly simple task presents formidable and unique challenges that render existing fuzzing paradigms ineffective. This section analyzes two fundamental challenges and explains why a new, specialized fuzzing architecture is necessary.

4.1. Challenge 1: Extreme Input Complexity

The first challenge stems from the extreme complexity of the input the network stack consumes. Unlike protocol implementations in user-space that process single-layer data streams, the network stack's fundamental unit of input is a complete, multi-layered network packet. For example, to fuzz the kernel's NFS implementation, one cannot simply inject an NFS data segment; the kernel would discard such an input at an early processing stage because it lacks the required Ethernet, IP, and TCP headers [42], [43]. This reality forces any fuzzer to operate *in vivo* by constructing fully encapsulated packets. This multi-layer requirement creates two non-trivial requirements for any fuzzer.

First is the **Construction Requirement**. A fuzzer must be able to generate structurally valid packets, often spanning four or more protocols, each with its own intricate header formats and options. Second is the **Mutation Requirement**. A fuzzer cannot simply apply random mutations to a packet's byte stream. The complex inter-dependencies between fields—such as checksums that depend on payloads, or length fields that must match actual data sizes—mean that naive mutations will almost always produce a packet that fails the basic sanity check. This necessitates a mutation strategy that strikes a delicate balance. If a mutation is too aggressive, the resulting packet will fail these trivial checks and never reach deep code paths. Conversely, if a mutation is too conservative, the generated inputs will be too similar to valid traffic and are unlikely to trigger the corner-case handling logic where vulnerabilities are more likely to reside. Therefore, a fuzzer must employ an intelligent strategy that preserves enough structural validity to pass basic sanity checks while introducing targeted malformation to explore deeper error-handling logics.

This dual requirement of complex construction and intelligent mutation proves to be an insurmountable obstacle for existing fuzzing paradigms. Protocol fuzzers are ill-suited for this task. They are overwhelmingly designed to target single, application-layer protocols implemented in user-space, such as an FTP [44] or SSH [45] server, and are not designed to handle the multi-layer packet construction required to reach a target like NFS inside the kernel. State-of-the-art kernel fuzzers like Syzkaller also fall short. Its description language, Syzlang, is designed for the relatively simple structure of system call arguments and is not expressive enough to effectively model the deeply nested, interdependent fields of a multi-layer network packet. This makes both the initial construction and subsequent intelligent mutation of packets an intractable problem within its existing framework.

4.2. Challenge 2: Dual-Channel Input Coordination

The second fundamental challenge is that the network stack operates on two distinct yet tightly coupled input channels: network packets arriving from the hardware and network-related system calls invoked from user space. The kernel's processing of an incoming network packet is critically dependent on the state that was previously established by these system calls. For instance, the kernel will summarily drop an incoming `TCP-SYN` packet [43] unless a user-space process has already invoked the `listen()` [46] system call to create and prepare a listening socket. This creates a critical requirement: an effective fuzzer must be able to precisely coordinate its network packet injections with corresponding system call invocations to navigate stateful protocol logics.

This requirement for cross-channel synchronization is a blind spot for existing approaches. Protocol fuzzers are entirely unequipped for this task. They operate exclusively on the network packet channel and are completely unaware of and unable to control the system call state of the kernel. Even sophisticated kernel fuzzers like Syzkaller cannot coordinate these channels effectively. While it can generate both syscalls and packets, its architecture lacks a mechanism to enforce the tight, logical, and temporal coupling required to ensure a system call like `listen()` has prepared the kernel state before the corresponding `SYN` packet is injected. It can generate both inputs, but it cannot guarantee they happen in the correct, state-aware sequence needed to make progress.

5. Method

This section details the complete methodology of NetPanic. We first present our core Fuzzer-in-the-Middle [18] based design philosophy (Section 5.1) and the system architecture that realizes it (Section 5.2). We then describe the core components of the fuzzing process: our structure-aware mutation strategy (Section 5.3) and the algorithms for fuzzing loop management (Section 5.4).

5.1. Design Philosophy: Fuzzer in the Middle

The core of our methodology is a **Fuzzer-in-the-Middle (FitM)** architecture. In this design, the fuzzer will position itself between two communicating network stacks that are driven by corresponding user-space programs to establish a communication session. The fuzzer intercepts every packet, applies targeted mutations, and forwards the modified packet to its destination. This architecture is not merely an implementation choice; it is a direct response to the *Extreme Input Complexity* and *Dual-Channel Coordination* challenges identified in Section 4.

First, it elegantly resolves the Extreme Input Complexity Challenge by completely separating packet generation from mutation. To satisfy the *Construction Requirement*, we delegate the entire task of packet generation to the

kernel itself. By executing standard user-space programs, we compel the kernel to act as a native, stateful packet generator, producing a stream of perfectly valid packets. This frees the fuzzer from the intractable burden of protocol data generation and state modeling. This stream of valid packets then serves as a high-quality baseline for satisfying the *Mutation Requirement*. By starting with a valid packet, we can apply precisely calibrated malformations to strike the delicate balance needed to bypass initial sanity checks. This approach allows our mutations to reach deep and state-dependent code paths, a topic we detail in Section 5.3.

Second, the FitM design inherently solves the *Dual-Channel Coordination Challenge*. The user-space client and server programs act as natural state orchestrators. They automatically manage the session state between two network stacks by invoking system calls (e.g. `socket()`, `listen()`, `connect()`) in the correct, logical sequence. The fuzzer does not need to model the protocol's state machine; the required synchronization between system calls and packet arrivals emerges as a natural property of the architecture. This enables the exploration of stateful protocol interactions that are only reachable through the correct synchronization of these two input channels.

As a bonus, the symmetry of this design allows systematic testing of both client-side and server-side protocol implementations seamlessly. By controlling which side of the communication receives malformed packets, we can shift the fuzzing focus without any architectural changes, ensuring comprehensive testing of a network protocol implementation.

5.2. System Architecture and Execution Flow

As illustrated in Figure 1, NetPanic employs a manager-agent architecture. The *manager* contains the core fuzzing logic and runs on the host, while the disposable *agent* serves as the *in-situ* execution engine inside the virtual machine. This design is a practical necessity, proven effective SOTA kernel fuzzer Syzkaller [47], as it isolates the fuzzer intelligence from the instability of the target kernel, which may crash or hang at any time.

The fundamental unit of work in this architecture is a seed, which is composed of a **Session Drivers** and a set of **Packet Modifier**. The Session Drivers are a pair of user-space programs that establish a complete and valid communication session between two network stacks when executed simultaneously. This design directly fulfills the *Construction Requirement* by delegating complex packet generation to the network stack itself. Simultaneously, because the programs naturally orchestrate the required system calls and network traffic, it inherently solves the *Dual-Channel Coordination Challenge*. Crucially, this architecture decouples the complex task of packet generation from the task of packet corruption.

This decoupling enables our solution to the *Mutation Requirement*: the set of Packet Modifier. Each Packet Modifier in this set is a precise, deterministic instruction to introduce a deterministic malformation to a specific single packet in the session. The Packet Modifier is elaboratedly designed to introduce adequate malformation to a packet while preserving its basic structural integrity, ensuring the resulting malformed packet is valid enough to bypass the basic sanity checks. This approach allows the fuzzer to focus its effort on corrupting small, targeted parts of a packet, freeing it from understanding the cumbersome structure of a complicated multi-layer packet.

The entire fuzzing process operates by iteratively executing and evolving these seeds. A complete fuzzing iteration is a feedback-driven loop orchestrated by the manager. First, the manager selects a seed from its corpus, applies a mutation to its Packet Modifier set, and sends the new seed to the agent. Upon receiving the seed, the agent executes the Session Drivers to initiate the session. It intercepts each resulting session packet in-flight, applies the Packet Modifier to the specific ones of them before forwarding them to their destination network stack. This process continues until the session finishes.

Throughout this execution, the agent gathers a comprehensive observation, including code coverage, the session trace, logs, and seed execution time (detailed in Section 5.4). Once the session is complete, this observation is returned to the manager. The manager analyzes the observation to check for new coverage (feedback) or crashes (objective) and saves the captured session trace as metadata with the seed. This metadata will serve as the structural blueprint for its mutation in the future.

5.3. Execution-Guided Structure-Aware Mutation

The design of our mutation strategy directly addresses the *Mutation Requirement* outlined in Section 4. The goal is to strike a delicate balance between packet validity and malformation, ensuring the malformed packets are valid enough to pass the basic sanity checks yet malformed enough to trigger deep, corner-case handling logic. To achieve this, we first formalize our mutation model (Section 5.3.1). We then detail the comprehensive suite of mutation operators derived from this model (Section 5.3.2) and the intelligent heuristics that guide their application (Section 5.3.3).

5.3.1. Mutation Model. At the core of our mutation strategy is the decoupling of session generation from packet mutation. This principle is embodied in our definition of a seed. Formally, a seed is a pair:

$$Seed = (D, \mathcal{M}) \tag{1}$$

$$\mathcal{M} = [M_1, \ldots] \tag{2}$$

Here, D is the Session Drivers, which deterministically generates a session containing a valid stream of packets. $\mathcal{M}$ is a set of Packet Modifiers containing a certain number of Packet Modifier M, each representing a precise instruction for how to introduce malformation to a specific packet in the session.

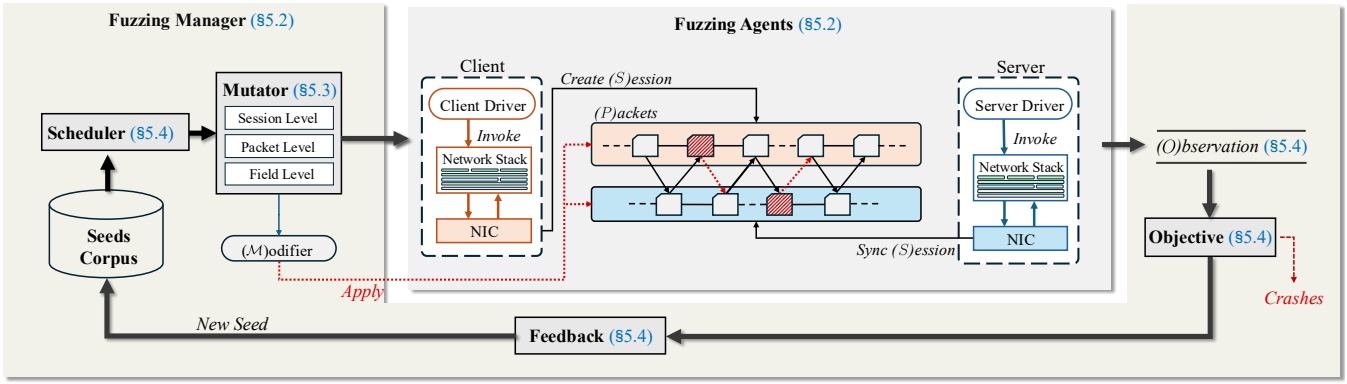


Figure 1: NetPanic System overview. The Fuzzing Manager orchestrates the fuzzing campaign on the host machine, while the agent executes seeds and collects observations within the VM. During the execution of a seed, the agent will initialize a communication session between two network stacks and intercept each packet in it. A deterministic malformation is introduced into the specific ones of them as guided by the seed, before they are forwarded to the destination network stack.

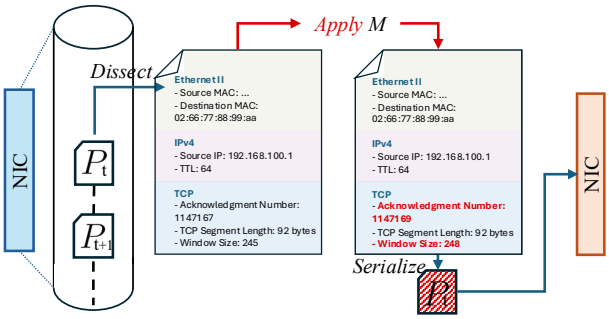


Figure 2: This diagram illustrates how a Packet Modifier works to introduce deterministic maflormedness to a network packet.

During execution, these two components work synchronously. When executed, D produces a clean, valid session S, which consists of an ordered sequence of k packets:

$$S = [P_1, \ldots, P_k] \tag{3}$$

As packets are generated in real-time, the agent intercepts them and checks if any Packet Modifier in $\mathcal{M}$ matches the current packet's index. If a match is found, the modifier is applied to introduce a malformation before the packet is forwarded to its destination. This interception and modification process continues for every packet exchanged between the two network stacks until the session concludes. Applying the entire set $\mathcal{M}$ to the base session S thus yields a systematically mutated session S' for fuzzing:

$$S \xrightarrow{\mathcal{M}} S' \tag{4}$$

The application of a single Packet Modifier M is a multi-step, structure-aware process, as illustrated in Figure 2. A Packet Modifier is a declarative instruction defined as a tuple:

$$M = \langle idx, op, meta \rangle \tag{5}$$

where $idx \in [1, k]$ specifies the target packet in the base session S, op is a mutation operator (detailed in Section 5.3.2), and $meta$ contains any operator-specific parameters. When a packet with a matching idx is intercepted, it is first dissected according to its protocol specifications. The op and $meta$ are then applied to this dissected data structure to introduce a deterministic malformation. Finally, the modified structure is serialized back into a byte stream and injected into the target network stack.

This model enables an efficient, evolutionary process that operates on the seed itself, guided by the session trace collected from the previous execution:

$$Seed(D, \mathcal{M}) \xrightarrow{S} Seed'(D, \mathcal{M}') \tag{6}$$

The manager dissects the captured trace S to inform its choices when adding, deleting, or altering Packet Modifier in $\mathcal{M}$ to create $\mathcal{M}'$. If the resulting $Seed'$ discovers interesting behavior, its resulting session trace S' is saved metadata for its future evolution.

5.3.2. Mutation Operators. The op element of a Packet Modifier is drawn from a comprehensive suite of operators designed to introduce a deterministic malformation at three distinct layers of abstraction.

(1) *Session Level* operators manipulate the sequence and timing of packets within a session without altering their content, including dropping a packet entirely (`DropPacket`) or delaying its injection (`DelayPacket`).

(2) *Packet Level* operators modify the high-level protocol composition of a single packet by replacing an entire protocol header with another (`ReplaceProtocol`), adding a new protocol layer (`AddProtocol`), or removing one (`RMProtocol`).

(3) *Field Level* operators target specific fields within a protocol header by setting a field to a boundary value (`SetToExtremeL/S`), a random value (`SetToRandom`), or applying small arithmetic changes (`ArithmeticPlus/Minus`), as well as manipulating

TABLE 1: Mutation operators.

Level	OP	Meta	Detail
Session Level	DropPacket	<>	drop the packet in the session
	DelayPacket	<*Duration*>	delay the packet in the session
Packet Level	ReplaceProtocol	<*src_prot,dst_prot,random*>	replace the protocol header with another in the packet
	AddProtocol	<*prot,random*>	add the protocol header in the packet
	RMProtocol	<*prot,random*>	remove the protocol header from the packet
Field Level	SetToExtremeL	<*prot,field,index*>	set the field to its extreme lower value
	SetToExtremeS	<*prot,field,index*>	set the field to its extreme upper value
	SetToRandom	<*prot,field,index,random*>	set the field to a random value
	ArithmeticPlus	<*prot, field,index,random*>	increment the field by a small value
	ArithmeticMinus	<*prot,field,index,random*>	decrement the field by a small value
	AddOptionalHeader	<*prot,field,random*>	add an optional header to the packet
	RMOptionalHeader	<*prot,field*>	remove an optional header from the packet

[1] This table summarizes the mutation operators used in our mutation.

optional protocol features (`AddOptionalField`, `RMOptionalField`), such as TCP optional fields and IPv6 [48] optional headers.

Each operator is coupled with the necessary *meta* parameters to define its precise behavior. For example, a Session-Level operator `DelayPacket` requires a delay duration to specify the exact duration of the delay. A complete list of all operators and their parameters is provided in Table 1.

5.3.3. Mutation Heuristics. We draw two heuristics from general fuzzing principles and our practical experience to focus the mutation process on the most productive paths.

Sparsity Heuristic. The Packet Modifier list $|\mathcal{M}|$ is kept small for every seed. Network sessions are often fragile, and severe malformation in a single packet can cause its underlying session to terminate prematurely. By applying mutations sparsely, we increase the probability that the session remains coherent to reach deeper protocol states.

Exclusivity Heuristic. Higher-level mutation operators preclude the lower-level ones within the same mutation stage. Modifying higher-level *op* is expected to alter the packet's structure, rendering lower-level *op* based on the original session trace S invalid. This heuristic ensures the logical consistency when mutating Packet Modifier set $\mathcal{M}$,thus increasing mutation efficiency.

5.4. Fuzzing Loop Management

To guide the fuzzing process, we employed several fuzzing components [49] that have been specifically adapted for kernel network stack fuzzing. This section details our customization to observation, feedback, objective, and scheduling.

Observation. At the end of each execution, the manager gathers a comprehensive observation [49], formalized as a tuple:

$$O = \langle C, S, L, T \rangle \tag{7}$$

where C is code coverage, S is the session trace containing all packets exchanged in the session, L contains kernel and user-space logs, and T is the execution time. It is worth noting that this observation tuple O will be stored as metadata if the seed is deemed interesting by our feedback function.

Feedback. The feedback function [49] uses the observation tuple O to identify interesting seeds for inclusion in the corpus. A seed is considered interesting if it increases code coverage (C), produces a new non-fatal log message in L, or generates a session with a previously unseen packet count ($|S|$).

Objective. The objective function [49] identifies seeds that can trigger vulnerabilities. Given an observation tuple O, the objective function will first detect kernel crash signatures in the log L to identify kernel panic. It will also check the execution time T and session trace S to identify possible system or server hang.

Scheduler. To maximize efficiency, our scheduler [49] prioritizes seeds from the corpus based on two rules. First, a *derivation depth* metric gives priority to seeds that have undergone many successful rounds of mutation from an original, valid session. The intuition is that highly evolved seeds generate complex, malformed sessions that can probe deep corner-case logic within the network stack, making them more productive at discovering vulnerabilities. Second, a standard *power schedule* de-prioritizes seeds that consistently execute slowly or time out, preventing the fuzzer from wasting cycles on unproductive seeds.

6. Implementation

We implemented our methodology to in prototype named NetPanic. It is written primarily in Rust [50] (approx. 14,000 LoC), with C (approx. 5,100 LoC) for low-level kernel interactions. NetPanic consists of a manager process running on the host and an agent process that executes seeds within a QEMU virtual machine. The manager contains the core fuzzing logic and is built on the *LibAFL* framework [49]. We leverage its infrastructure for corpus management and implement our fuzzing logic by replacing its fuzzer components with our customized ones.

Upon execution, the *agent* will initialize the seed execution environment within the VM and establish a com-

munication channel with the *manager*. It uses the Linux kernel *namespace* [51] feature to create two isolated network stacks. Each stack is assigned a *TAP* virtual device serving as its network gateway, which allows the *agent* to intercept and inject packets from it. According to a recent commit, packets written to a *TAP* device will be processed synchronously [52], bypassing the dereference mechanisms in the network stack [53]. This allows us to use *KCOV* on the `write()` syscall to track the exact code paths executed. This allows the coverage collection with *KCOV* by tracking the `write()` system call of the *TAP* device.

We leverage the power of network traffic analyzer `Wireshark` [54] as our packet dissection engine. The ability to perform in-flight packet dissection for structure-aware mutation is made practical by a recent commit in `Wireshark` that allows it to parse single packets from standard input with minimal overhead [55].

7. Evaluation

This section evaluates the performance of NetPanic to fuzz the Linux kernel network stack, structured around four research questions.

- **RQ1: Vulnerability Discovery.** Does NetPanic effectively discover remotely triggerable vulnerabilities in the Linux kernel network stack?
- **RQ2: Fuzzing Effectiveness.** How does NetPanic compare to existing approaches in terms of code coverage across various network protocol layers?
- **RQ3: Fuzzing Efficiency.** How efficient is NetPanic in terms of packet injection throughput and meaningful interaction compared to existing kernel fuzzers?
- **RQ4: Design Contribution.** What is the specific contribution of our design choices to the overall effectiveness of NetPanic?

We begin by detailing the experimental setup in Section 7.1. To answer RQ1, we report the vulnerabilities discovered by NetPanic in Section 7.2. Next, to address RQ2 and RQ3, we benchmark NetPanic against Syzkaller, a state-of-the-art kernel fuzzer, comparing code coverage (Section 7.3) and execution efficiency (Section 7.4). Finally, we conduct an ablation study in Section 7.5 to isolate the contribution of each design component, addressing RQ4.

7.1. General Experimental Setup

We select Syzkaller, the state-of-the-art kernel fuzzer, as our baseline to benchmark NetPanic's performance. While being a general-purpose kernel fuzzer, it is the only publicly available, state-of-the-art tool with the capability to inject packets into the kernel network stack, making it the most relevant and robust baseline for comparison (see Section 2 for a detailed discussion of related fuzzers and why they are not directly comparable).

All experiments were conducted on a host machine equipped with an Intel Core i7-14700K CPU and 32GB of RAM. The target kernel for all experiments is Linux kernel version 6.14, a recent stable release compiled with its default configuration under our hardware platform. We enabled *KCOV* for coverage collection and *KASAN* for memory corruption detection.

7.2. Vulnerability Discovery

Throughout the development of NetPanic, it is configured to target a diverse set of protocol implementations within the Linux kernel network stack, and NetPanic continuously discovers novel vulnerabilities in them. By the time of paper submission, NetPanic has identified **15** novel, remotely triggerable vulnerabilities.

Table 2 provides a comprehensive list of them. Those vulnerabilities span multiple protocols, including TCP, SMB, NFS, and SUNRPC, and all expose severe security risks. Some of those vulnerabilities have been fixed by the Linux community with CVE ID assigned, while others are undergoing responsible disclosure.

The results provide a decisive answer to **RQ1**. To contextualize the findings, we performed a direct comparison against a dedicated Syzkaller instance tuned to focus exclusively on the network stack. In a fuzzing campaign over 3 months, Syzkaller failed to discover any novel remotely triggerable vulnerabilities, and it failed to reproduce any vulnerability identified by NetPanic. Google Syzbot continuously fuzzes the Linux kernel with massive computing resources, yet it also reported no new remotely triggerable vulnerabilities during this period. This powerful outcome affirms that NetPanic represents a significant advancement in discovering security risks of the remote attack surface in the Linux kernel network stack.

7.3. Effectiveness

Having demonstrated that NetPanic is highly effective in finding novel remotely triggerable vulnerabilities, we now quantitatively analyze the reasons for its success. This section addresses **RQ2** by evaluating the effectiveness of NetPanic through a detailed code coverage analysis against our baseline.

7.3.1. Evaluation Setup. Fuzzing the kernel network stack through packet injection makes it infeasible to target individual protocols in isolation, as we have established in Section 4.1. To create a representative evaluation, we selected three widely-used protocol combinations as our target: Eth-IPv4-TCP, Eth-IPv6-UDP, and Eth-IPv4-TCP-NFS. This selection provides a solid basis for comparison by covering protocols at the network, transport, and application layers, and includes both connection-oriented and connectionless transport models.

To establish a rigorous baseline, we carefully tuned Syzkaller to maximize its performance in fuzzing those selected protocol combinations. Our tuning represents our best-effort attempt, involving the restriction of its system call selection to only those that inject packets for our chosen

TABLE 2: Remotely Triggerable Vulnerabilities Discovered by NetPanic.

#	Attack Result	Mem Corruption	Protocol	Root Cause	Status	Disclosure
1	Server Hang	N/A	SMB	ksmbd_kthread_fn	Fixed	CVE-2025-38089
2	Kernel Panic	NPD	SUNRPC	nfsd	Fixed	CVE-2025-38089
3	Kernel Panic	UAF	SUNRPC	cache_clean	Fixed	CVE-2025-40212
4	Kernel Panic	Mem Drain	NFSv4	sunrpc_cache_lookup_rcu	Fixed	CVE-2025-40210
5	Kernel Panic	UAF	SMBv3	fib_get_table	Fixed	CVE-2026-23220
6	Kernel Panic	NPD	SMBv3	smb3_get_tree	Fixed	Embargoed
7	Kernel Panic	GP	SMBv3	cifs_smb3_do_mount	Fixed	Embargoed
8	Kernel Panic	UAF	SMBv3	svc_process_common	Fixed	Embargoed
9	Kernel Panic	UAF	NFSv3	nfs_localio_disable_client	Fixing	Pending
10	System Hang	N/A	SUNRPC	91da337e5d506f	Fixing	Pending
11	Perf Degrad	N/A	SMBv2	crypt_message	Fixing	Pending
12	Kernel Panic	UAF	SMBv2	944aacb68baf76	Fixing	Pending
13	System Hang	N/A	SMBv2	110fee6b9bb58a	Fixing	Pending
14	Kernel Panic	UAF	SMBv2	smb_grant_oplock	Fixing	Pending
15	System Hang	N/A	TCP	inet_frag_rbtree_purge	Fixing	Pending

[1] Each entry represents a unique bug confirmed with a stable reproducer.
[2] *Attack Result* abbreviations: *System/Server Hang* (system or server thread unresponsive), and *Perf Degrad* (severe performance degradation).
[3] *Memory Corruption* abbreviations: *NPD* (Null-Pointer Dereference), *UAF* (Use-After-Free), and *GP* (General Protection Fault).

protocol combinations, as well as the provisioning of initial seeds for these targets.

Besides, a consideration for a fair comparison stems from the fundamental architectural differences between the fuzzers. Syzkaller's architecture inherently positions it as an external client, capable of fuzzing and collecting coverage only for the server-side implementation of each protocol. In contrast, the symmetric design of NetPanic allows it to fuzz both client and server sides simultaneously. To ensure a fair comparison, we configured NetPanic to collect coverage data only from the server-side network stack, mirroring Syzkaller's perspective. Thus, it is important to note that the coverage results presented for NetPanic in this section represent approximately **half** of its total fuzzing capability.

We measured code coverage at the basic block level using KCOV, tracing execution paths originating from the `write()` system call that injects packets into the TAP device. Raw coverage data was mapped to source code lines using `addr2line` [56] and then aggregated by protocol layer for a fine-grained analysis. For each protocol combination, both NetPanic and the tuned Syzkaller baseline were configured to run with one single thread for 24 hours. Each experiment was repeated three times to ensure the results were consistent and to mitigate the effects of randomness.

7.3.2. Results. The results of our 24-hour fuzzing experiments are presented in Figure 3. As shown in Figure 3f, both fuzzers achieved negligible coverage in the Ethernet layer. This is expected because Layer 2 logic resides almost entirely within the corresponding NIC driver, which is the TAP virtual NIC device in this case. For this reason, we deliberately omit Layer 2 coverage from our analysis.

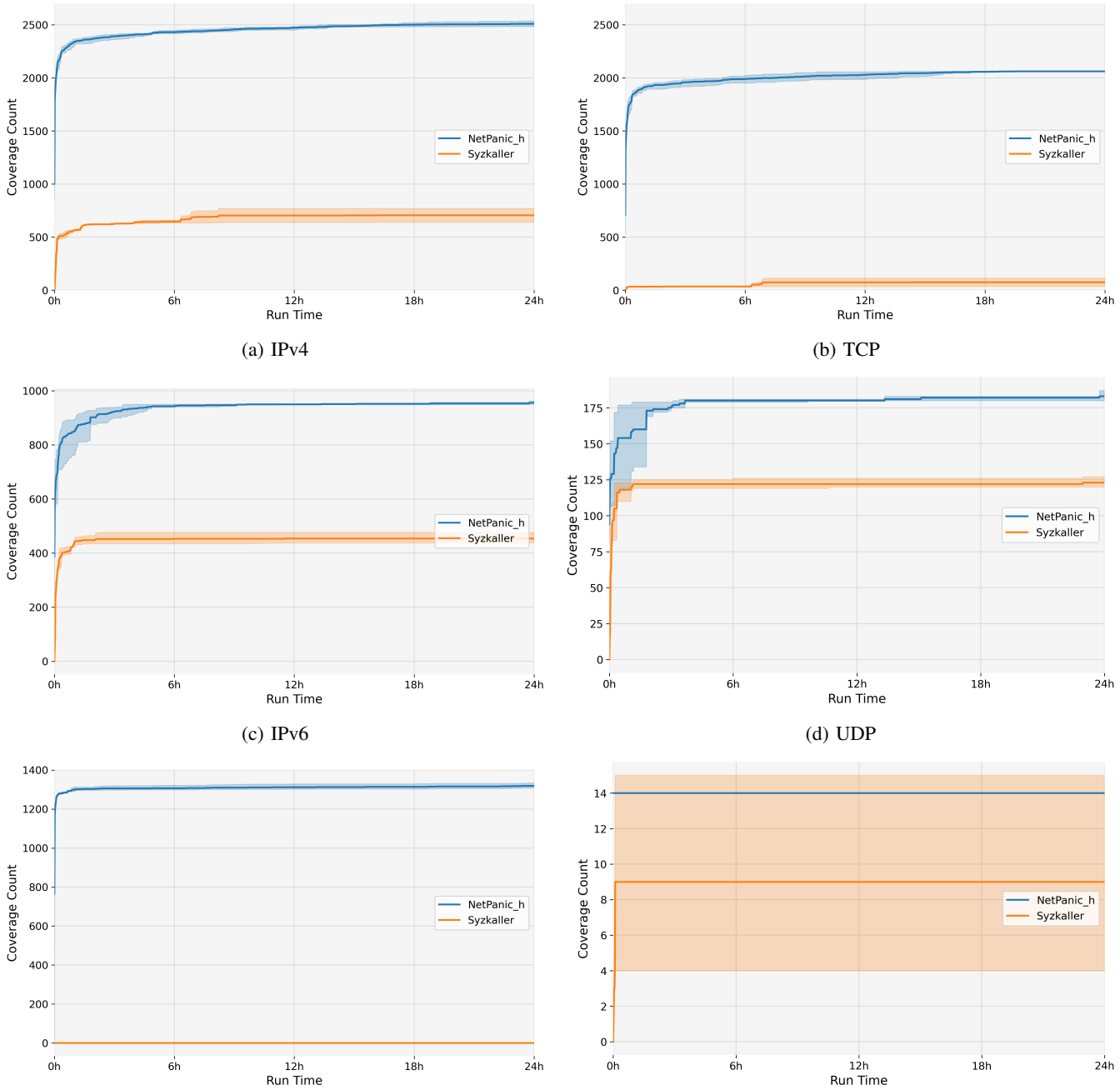
The data reveals a clear and consistent outcome across all experiments: NetPanic significantly outperforms the tuned Syzkaller baseline across every protocol layer. Quantitatively, NetPanic achieves an average coverage improvement of approximately 400%, accounting for all protocols.

The performance gap is most pronounced in the stateful TCP (Figure 3b) and application-layer NFS (Figure 3e) protocols. For TCP, Syzkaller failed to establish a single successful connection throughout the 24-hour experiment and achieved very limited coverage in this layer. For NFS, Syzkaller achieved zero coverage.

By analyzing the execution log of Syzkaller, we confirmed that its dramatic failure in TCP stems from its inability to manage stateful connections. Despite being provided with a valid seed, it failed to establish any valid TCP connection. This inability prevented it from exploring the majority of TCP protocol logics and had a cascading effect on NFS. The complete failure in NFS has two causes. The direct cause is that Syzkaller lacks the necessary protocol specifications to generate valid NFS packets. More fundamentally, even if such specifications were provided, its inability to establish and maintain the required underlying TCP connection would still render it ineffective at fuzzing NFS.

7.3.3. Investigation. The analysis of these results reveals that the performance gap between NetPanic and the baseline is not incremental but fundamental, stemming directly from our architectural solutions to the unique challenges of network stack fuzzing.

First, the broad coverage advantage of NetPanic across all protocols demonstrates its success in fulfilling the *Construction Requirement* (Section 4.1). Syzkaller's reliance on manually written Syzlang templates proves fundamentally intractable for modeling the vast and evolving specifications of the network stack. Despite being a mature, community-driven project with years of development, its network protocol templates remain far from comprehensive. This inadequacy is a direct result of the intractability of the manual approach and leads to the inferior coverage we observed. In contrast, NetPanic elegantly addresses this problem by reusing the network stack itself as a generator to produce valid, context-aware packets, enabling it to seamlessly explore deep code paths without any protocol-specific



(a) IPv4 (b) TCP

(c) IPv6 (d) UDP

(e) NFS (f) Ethernet

Figure 3: Code coverage comparison across protocol layers.

modeling.

Second, NetPanic's overwhelming coverage advantage in stateful protocols highlights its success in addressing the *Dual-Channel Coordination* Challenge (Section 4.2). This is starkly illustrated by Syzkaller's complete failure to establish a single valid TCP connection. While it can generate the correct system calls and packets, it failed to synchronize them correctly. This failure prevents it from exploring the vast majority of the TCP protocol state space and, by extension, any application-layer protocol like NFS that depends on it. By contrast, NetPanic's Fuzzer-in-the-Middle design inherently coordinates these two input channels, allowing it to naturally establish and maintain stateful connections and explore the entire connection lifecycle.

In conclusion, the superior coverage of NetPanic is not coincidental; it is a direct consequence of our architectural

solutions to the unique challenges of network stack fuzzing. These results provide a clear, affirmative answer to **RQ2**, demonstrating that our methodology is significantly more effective at fuzzing the attack surface of the network stack, which in turn explains its superior vulnerability discovery performance (RQ1).

7.4. Efficiency

In this section, we evaluate the execution efficiency of NetPanic to address **RQ3**.

7.4.1. Evaluation Setup. We employ the same experimental setup as in Section 7.3, with two execution efficiency metrics:
Packet Injection Rate is defined as the number of packets injected into the network stack per second; a higher rate directly translates to more opportunities to exercise code paths and discover vulnerabilities within a given timeframe.
Packet Reply Rate is defined as the number of reply packets received from the network stack per second, serving as a crucial proxy for input quality. Replies, particularly in connection-oriented protocols, indicate that injected packets are valid enough to be successfully parsed, processed, and responded to by the network stack, signifying meaningful interaction.

For NetPanic, we directly measure the packets sent to and received from the network stack. For Syzkaller, we instrumented its source code to count the successful executions of the pseudo-system calls responsible for packet injection and reception. Metrics were sampled every minute over a 24-hour period, and each experiment was repeated three times.

7.4.2. Results. The results, presented in Figure 4, reveal a stark difference in both the quantity and quality of inputs generated by the two fuzzers.

For the TCP and UDP scenarios, NetPanic's average packet injection rate is **three orders of magnitude higher** than Syzkaller's peak performance. Besides, NetPanic maintains a consistently high injection rate throughout the 24-hour period, while Syzkaller's injection rate collapses to near zero shortly after starting. For the NFS protocol, Syzkaller's injection rate is zero because of a lack of support.

Regarding packet reply rate, NetPanic achieves a high and consistent reply rate across all tests. In contrast, Syzkaller achieves a zero reply rate across all experiments. Syzkaller cannot complete the TCP handshake; its UDP packet injection is designed for a one-shot without listening for replies, and it has no support for NFS.

7.4.3. Investigation. First, this stark difference in injection rate and stability is not merely a performance artifact; it highlights a fundamental architectural advantage of NetPanic. Syzkaller's performance collapses because its general-purpose, coverage-guided scheduler is ill-suited for this task. Its packet injection syscalls quickly exhaust the low-hanging coverage, causing the scheduler to deprioritize them as unproductive. This creates a vicious cycle: as fewer packets are injected, the fuzzer has fewer opportunities to reach deeper states and discover new coverage, leading to a complete stall. Conversely, NetPanic's design ensures a consistently high and stable throughput, applying continuous testing pressure that is not dependent on the immediate coverage returns of simple inputs.

Second, the packet reply rate difference highlights the dramatic gap in input quality. NetPanic's high reply rate confirms that its packets are valid enough to successfully engage in the network stack's implementation of protocol logic. This demonstrates that NetPanic's injections are not only faster but are also far more effective at driving meaningful, stateful interactions. Syzkaller's zero reply rate across all experiments is due to its architectural limitations.

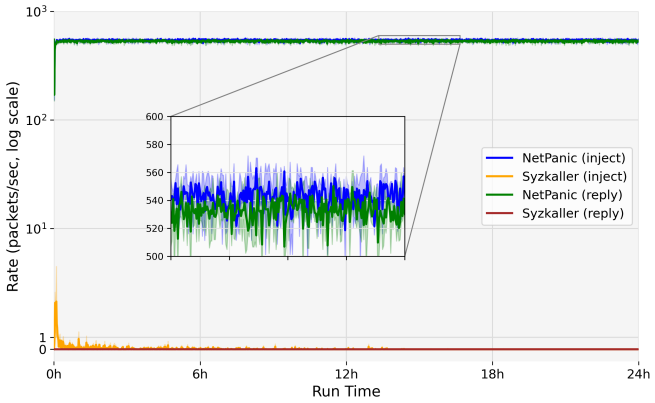
In summary, NetPanic is not only quantitatively faster but also qualitatively more effective than the baseline. These gains stem directly from our architectural solutions to the unique challenges. The ability to reuse protocol implementations in the network stack itself as a generator ensures a stable supply of high-quality network packets. And those packets are perfectly orchestrated to effectively engage the stateful protocol logic, which is validated by the high packet reply rate. This combination of high injection throughput and meaningful interaction decisively answers **RQ3** and serves as the foundation for its superior performance in coverage gain and vulnerability discovery reported earlier.

7.5. Ablation Study

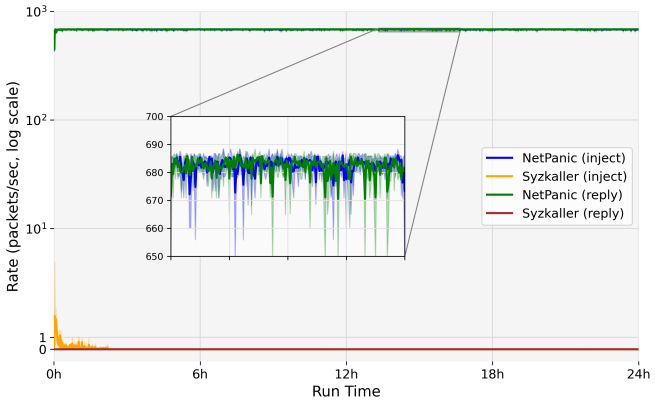
To answer **RQ4** and quantify the contribution of our design choices, we conducted an ablation study. This study is designed to directly validate that our solutions fulfill the *Mutation Requirement* (Section 4.1) and effectively address the *Dual-Channel Coordination* Challenge (Section 4.2).

7.5.1. Ablation Methodology. We designed an experiment with our full fuzzer and three ablated configurations, each targeting the Eth-IPv4-TCP protocol combination for 24 hours:

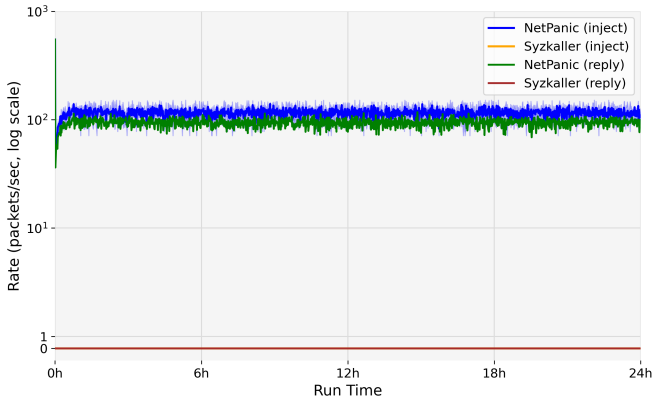
- **Group 1 (NetPanic Full System):** This is our complete, unmodified NetPanic fuzzer, serving as the point of comparison.
- **Group 2 (Valid-Only Traffic):** This group operates with all mutation operators disabled. It establishes a baseline for coverage achieved by sending only perfectly valid, protocol-compliant packets.
- **Group 3 (No Syscall Coordination):** This group operates with an empty Server Driver. This prevents the invocation of system calls that prepare the network stack for incoming TCP packets, isolating the effect of packet injection alone.
- **Group 4 (Random Mutation):** This group replaces our structure-aware mutator with a naive, byte-level random mutation engine, similar to AFL havoc mutator [57]. This configuration represents a structure-



(a) IPv4-TCP

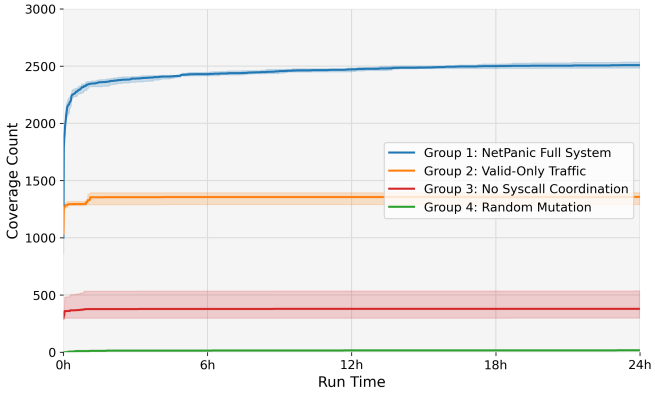


(b) IPv6-UDP

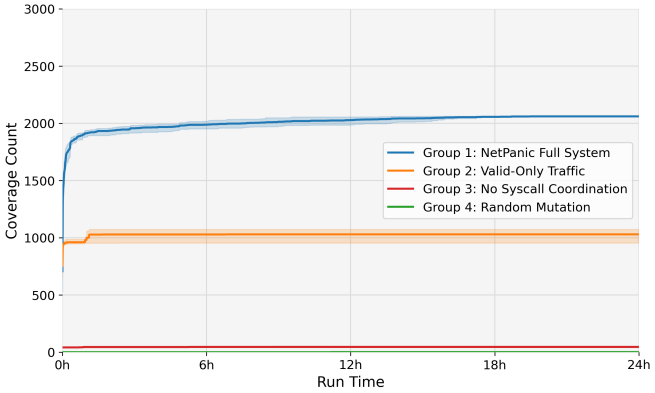


(c) IPv4-TCP-NFS

Figure 4: packet injection and reply rate comparison across protocol combinations.



(a) Coverage in IPv4 layer



(b) Coverage in TCP layer

Figure 5: Ablation study coverage comparison.

unaware approach that prioritizes malformation over validity.

We measured effectiveness using the same per-layer code coverage metric from Section 7.3, repeating each experiment three times to ensure consistent results.

7.5.2. Results. The results are presented in Figure 5. Group 2 (Valid-Only Traffic) achieves a respectable baseline coverage in both protocol layers by successfully navigating the standard protocol logic, though its coverage is significantly lower compared to that in our full system Group 1. Group 3

(Random Mutation) achieves almost zero coverage in both the IPv4 and TCP layers. Group 4 (No Syscall Coordination) achieves moderate coverage in the IPv4 layer, but virtually zero coverage in the TCP layer. In contrast, Group 1 (NetPanic- Full System) achieves the highest coverage across both layers by a significant margin.

7.5.3. Investigation. The coverage gap between Group 1 (NetPanic Full System) and Group 2 (Valid-Only Traffic) directly quantifies the contribution of our mutation design. Group 2 establishes a baseline coverage by exercising the trivial, standard processing logic of the network stacks. However, this valid-only traffic is inherently unable to reach the vast error-handling and corner-case code paths where vulnerabilities are most often found. The significant coverage gap between Group 2 and our full system represents the direct gain from our mutation design, targeting precisely the corner-case logics that are most productive for vulnerability discovery.

While the results from Group 2 highlight the need for mutation, the catastrophic failure of Group 4 (Random Mutation) underscores that its intelligence is paramount. The naive havoc mutations produced packets so structurally unsound that they were immediately discarded by the network stack's initial sanity checks, resulting in near-zero coverage. This result, together with the antecedent analysis of Group 2, validates that our mutation strategy design effectively fulfills the *Mutation Requirement* (Section 4.1). It strikes the necessary balance, introducing malformations that are potent enough to explore corner-case logics while preserving the structural validity required to penetrate deep into the network stack.

The coverage gap between our full system and Group 3 (No Syscall Coordination)perfectly illustrates its ability in addressing the *Dual-Channel Coordination* Challenge (Section 4.2). The moderate coverage in the IPv4 layer occurs because the network stack will process individual IP packets to some extent in its connectionless logic. However, without the corresponding system calls to create a listening socket in the recipient network stack, it summarily drops all incoming TCP packets, preventing any exploration of the valid TCP state space.

In summary, this ablation study, combined with our earlier results, provides clear, empirical validation for our entire design philosophy, addressing **RQ4**. The superior performance of NetPanic against the ablated configurations and the Syzkaller baseline provides comprehensive evidence that it successfully addresses the unique challenges identified in Section 4.

8. Discussion

In this section, we discuss the broader implications of our findings, the limitations of our current approach, and promising avenues for future research.

Implications for Kernel Security. The discovery of numerous severe, remotely triggerable vulnerabilities indicates that the remote attack surface of the Linux kernel is a significant and under-examined threat. For years, the kernel security academic and industry [58] community has overwhelmingly focused on the local attack surface, particularly local privilege escalation attacks and defenses. Our findings should serve as a call to action for both the research and industry communities, providing compelling evidence that the remote attack surface poses a critical risk that has been largely overlooked. Identifying these vulnerabilities is the first concrete step in a comprehensive security audit of this surface. We hope this work serves as a foundation that inspires future research into the discovery, exploitation, and defense of vulnerabilities in this attack surface.

Adaptation to Other Kernels. The design and architecture of NetPanic operate on the high-level abstraction of a communication session, making it inherently OS-agnostic. While the manager's fuzzing logic is portable, the agent is implementation-specific. By rewriting the agent, our framework can be seamlessly adapted to fuzz the network stacks of other operating systems. We have already begun this process by porting NetPanic to target the Windows kernel. This adaptation is feasible due to the availability of equivalent technologies for network stack isolation and packet interception, analogous to Linux's namespaces [59] and TAP devices [60]. We are confident that this approach will prove effective on other platforms.

Expanding the Mutation Space. Our current mutation strategy focuses exclusively on one of the kernel's two input channels: network packets. As we established in Section 4.2, system calls represent a second, tightly-coupled input channel. A promising direction for future work is to extend our mutation engine to also mutate the parameters of network-related system calls invoked by the Session Drivers. This would enable the exploration of complex corner cases that arise from the interaction of system call inputs and malformed packet inputs, potentially uncovering deeper and more subtle vulnerabilities.

Implementation-Level Optimization. The current NetPanic prototype was designed to validate our methodology, with performance optimization being a secondary concern. While it already demonstrates a three-order-of-magnitude efficiency improvement over the baseline, its performance is far from its full potential. Specifically, the execution efficacy of NetPanic is constrained by static timeouts under the current implementation. The results shown in Section 7.4 adopt a hard-coded set of timeout configurations based on our experience, and we do not claim they represent the maximal or average execution efficacy across all protocols. We are currently implementing a dynamic mechanism to allow NetPanic to adjust these timeouts automatically based on runtime observations, maximizing throughput. Furthermore, we are actively adding support for multi-threaded parallelism and optimizing the low-level IPC between the manager and agent to scale the fuzzing effort and minimize overhead.

Improving Deflaking Heuristics. Flaky test cases are a notorious problem in kernel fuzzing. NetPanic currently adopts the same pragmatic deflaking strategy [61] used by Syzkaller, which involves re-executing a potentially inter-

esting seed multiple times to ensure its behavior is stable. While this approach has proven effective in practice, it is entirely empirical and has not been rigorously evaluated. Future work could involve developing more sophisticated, observation-based heuristics to improve the accuracy and efficiency of deflaking. As this is a common challenge, broader research in this area would benefit the entire kernel fuzzing community, including NetPanic.

9. Conclusion

In this work, we addressed the security of a critical but long-overlooked area: the remote attack surface of the Linux kernel network stack. We began by identifying the unique challenges that make fuzzing this target intractable for existing tools. To solve these challenges, we presented NetPanic, the first novel fuzzing framework designed to systematically and effectively audit this attack surface for remotely triggerable vulnerabilities in kernel network stack.

Our comprehensive evaluation provided clear evidence that this success is a direct consequence of our design and architecture. The performance comparison against Syzkaller, combined with our ablation study, empirically validates that our design and architecture are an effective solution to the unique challenges of kernel network stack fuzzing. This provides a dual validation: it not only confirms the correctness of our insight in identifying the specific challenges in kernel network stack fuzzing, but also proves the effectiveness of our solutions for addressing them.

The practical impact of this work is underscored by the discovery of 15 new, remotely triggerable vulnerabilities in the Linux kernel. This finding highlights the severe and immediate risk posed by the kernel's remote attack surface and serves as a call to action for the research and industry communities to devote urgent attention to this area. As the first concrete step in a comprehensive security audit of this surface, NetPanic provides an effective tool and a methodological foundation to begin securing it.

10. Ethics considerations

The primary ethical consideration for a tool like NetPanic is the responsible handling of the vulnerabilities it discovers. Given that our evaluation has proven its ability to find new, remotely triggerable vulnerabilities—all of which are considered high or critical risk—we have adhered to a strict policy of responsible disclosure. Throughout our evaluation, all identified vulnerabilities were promptly reported to the Linux kernel security team, following their standard disclosure guidelines [19]. Our commitment extends beyond simple reporting; we have actively cooperated with kernel maintainers to facilitate fixes. This cooperation included providing detailed crash reports, stable reproducers, assistance with bisecting to identify root causes, and verification of proposed patches.

Recognizing that NetPanic is a potent fuzzer with significant potential to find more bugs, an immediate public release could introduce unpredictable security risks. Therefore, we will provide a private demonstration to the Linux kernel security team and will only open-source the full codebase after receiving their approval. As we adapt NetPanic to other operating systems, we are committed to extending this ethical framework. We will engage in the same process of active cooperation and responsible disclosure with the respective security teams or vendors and will seek their consent before any potential release of the tool.

Acknowledgements

We are grateful to all the anonymous reviewers for their insightful comments, which have greatly improved our paper. The research is supported in part by the National Natural Science Foundation of China (No. 62202465), the National Key Research and Development Program of China (No. 2021YFB2910109), the Beijing Key Laboratory of Network Security Protection Technology (No. 2022YFB3103900), and the Outstanding Talent Scheme (Category B) - Qihang Zhou (E3YY141116).

References

[1] H. Zhang, J. Kim, C. Yuan, Z. Qian, and T. Kim, "Statically discover cross-entry use-after-free vulnerabilities in the linux kernel," in *Network and Distributed System Security (NDSS) Symposium*, 2025.

[2] K. Hu, Q. Chen, Z. Lu, W. Zhang, B. Chen, Y. Lu, H. Jiang, B. Sun, X. Peng, and W. Zhao, "A survey of fuzzing open-source operating systems," *arXiv preprint arXiv:2502.13163*, 2025.

[3] Y. Sun, Y. Kang, C. Wu, K. Lu, J. Wang, X. Li, Y. Hu, J. Ren, Y. Lai, M. Xie *et al.*, "Syzparam: Introducing runtime parameters into kernel driver fuzzing," *arXiv preprint arXiv:2501.10002*, 2025.

[4] V. Dmitry, "syzkaller: An unsupervised coverage-guided kernel fuzzer," https://github.com/google/syzkaller, 2016.

[5] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1630–1644.

[6] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3262–3278.

[7] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 71–88.

[8] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "Pspray: Timing {Side-Channel} based linux kernel heap exploitation technique," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6825–6842.

[9] Z. Lin, Y. Wu, and X. Xing, "Dirtycred: Escalating privilege in linux kernel," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1963–1976.

[10] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "{FUZE}: Towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 781–797.

[11] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting user-controlled data to burn away linux kernel protections," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 3093–3107.

[12] L. Maar, L. Giner, D. Gruss, and S. Mangard, "When good kernel defenses go bad: Reliable and stable kernel exploits via defense-amplified tlb side-channel leaks," in *34rd USENIX Security Symposium: USENIX Security 2024*. USENIX Association, 2025.

[13] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "{kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 167–182.

[14] A. Konovalov, "External network fuzzing for linux kernel," https://github.com/google/syzkaller/blob/master/docs/linux/external-fuzzing-network.md, 2023.

[15] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[16] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proceedings of the seventeenth european conference on computer systems*, 2022, pp. 166–180.

[17] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 489–502.

[18] D. Maier, O. Bittner, M. Munier, and J. Beier, "FitM: Binary-only coverage-guided fuzzing for stateful network protocols," in *Workshop on Binary Analysis Research (BAR)*, 2022.

[19] Linux Kernel Community, "Linux kernel security bugs," https://docs.kernel.org/process/security-bugs.html, 2024.

[20] S. Kagstrom, "Kcov: code coverage for fuzzing," https://docs.kernel.org/dev-tools/kcov.html, 2025.

[21] D. V. Andrey Konovalov, Alexander Potapenko, "Kernel address sanitizer (kasan)," https://docs.kernel.org/dev-tools/kasan.html, 2025.

[22] D. Vyukov, "Syzlang: the eclarative description of syscall interfaces to manipulate programs by syzkaller," https://github.com/google/syzkaller/blob/master/docs/syscall-descriptions.md, 2025.

[23] "Universal tun/tap device driver," https://docs.kernel.org/networking/tuntap.html, 2025.

[24] D. Vyukov, "Syzbot: the system continuously fuzzes main linux kernel branches and automatically reports found bugs to kernel mailing lists," https://syzkaller.appspot.com/upstream, 2025.

[25] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "Grebe: Unveiling exploitation potential for linux kernel bugs," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2078–2095.

[26] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "Syzgen: Automated generation of syscall specification of closed-source macos drivers," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 749–763.

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

[28] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "HEALER: Relation learning guided kernel fuzzing," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021, pp. 344–358.

[29] C. Liu, S. Gong, and P. Fonseca, "Kit: Testing os-level virtualization for functional interference bugs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 427–441.

[30] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 147–161.

[31] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 818–834.

[32] T. Mu, H. Zhang, J. Wang, and H. Li, "Colafuze: Coverage-guided and layout-aware fuzzing for android drivers," *IEICE TRANSACTIONS on Information and Systems*, vol. 104, no. 11, pp. 1902–1912, 2021.

[33] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRACE: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[34] S. Huster, M. Hollick, and J. Classen, "To boldly go where no fuzzer has gone before: Finding bugs in linux'wireless stacks through virtio devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4629–4645.

[35] X. Zhang, C. Zhang, X. Li, Z. Du, B. Mao, Y. Li, Y. Zheng, Y. Li, L. Pan, Y. Liu *et al.*, "A survey of protocol fuzzing," *ACM Computing Surveys*, vol. 57, no. 2, pp. 1–36, 2024.

[36] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 191, 2022.

[37] P. Biondi, "Scapy: Packet crafting for Python," https://scapy.net, 2003.

[38] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *9th ACM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[39] "Network subsystem in the linux kernel," https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html, 2025.

[40] "Mannual of nfs in the linux kernel," https://man7.org/linux/man-pages/man5/nfs.5.html, 2025.

[41] "Device drivers for ethernet and ethernet-based virtual function devices." https://docs.kernel.org/networking/device_drivers/ethernet/index.html, 2025.

[42] "the internet protocol," https://datatracker.ietf.org/doc/html/rfc791, 1981.

[43] "Specification of the transmission control protocol (tcp)," https://datatracker.ietf.org/doc/html/rfc793, 1981.

[44] "the file transfer protocol," https://datatracker.ietf.org/doc/html/rfc765, 1985.

[45] "Specification of the secure shell (ssh) transport layer protocol," https://datatracker.ietf.org/doc/html/rfc4253, 2006.

[46] "Mannual of socket and its related system calls in the linux," https://man7.org/linux/man-pages/man2/socket.2.html, 2025.

[47] V. Dmitry, "The internal architecture of syzkaller," https://github.com/google/syzkaller/issues/4639, 2024.

[48] "Specification of the internet protocol, version 6 (ipv6)," https://datatracker.ietf.org/doc/html/rfc8200, 2017.

[49] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[50] "The rust programing language," https://datatracker.ietf.org/doc/html/rfc4253, 2006.

[51] "The namespace feature of the linux kernel," https://man7.org/linux/man-pages/man7/namespaces.7.html, 2025.

[52] A. Konovalov, "The recent patch to tap device that allows bypassing of the dereference machanisms in the network stacks," commit id=d4aea20d889e05575bb331a3dadf176176f7d631, 2025.

[53] C. Benvenuti, *Understanding Linux network internals*. " O'Reilly Media, Inc.", 2006.

[54] J. Thacker, "Wireshark the a network traffic analyzer," https://gitlab.com/wireshark, 2025.

[55] J. T. Anders Broman, "low overhead in packet dissection," https://gitlab.com/wireshark/wireshark/-/commit/f0712606a3d014a915e585997f624640b326b9c0, 2025.

[56] "The address mapping tools that convert addresses into file names and line numbers," https://man7.org/linux/man-pages/man1/addr2line.1.html, 2006.

[57] M. Zalewski, "American Fuzzy Lop," https://lcamtuf.coredump.cx/afl/, 2014.

[58] Google, "Google kctf." https://google.github.io/kctf/, 2025.

[59] "Windows equivalent of the linux networking namespace: the network compartments." https://learn.microsoft.com/en-us/uwp/api/windows.networking?view=winrt-26100, 2014.

[60] OpenVPN, "Windows equivalent of the linux tap device: Windows tap device driver." https://github.com/OpenVPN/tap-windows, 2014.

[61] V. Dmitry, "Deflaking stretagy of syzkaller," https://github.com/google/syzkaller/issues/4639, 2024.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper introduces a new "fuzzer-in-the-middle" approach that orchestrates fuzzing of two OS kernel network stacks. It employs a guided packet mutation mechanism and coordinates system call invocations between the two sides to facilitate fuzzing. The authors demonstrate the effectiveness of the proposed approach in identifying 15 new remote triggerable vulnerabilities in Linux and show that it outperforms the state-of-the-art, Syzkaller.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

1) The paper creates a new tool to enable future science. The paper introduces a new framework to the community that enables efficient fuzzing of the remote network stack with high coverage, facilitating future research in identifying remotely triggerable vulnerabilities.
2) The paper provides a valuable step forward in an established field. The authors identify a key limitation in the network stack fuzzing framework: the lack of coordination between packet mutation and system call usage, and introduce a new "fuzzer-in-the-middle" approach to address it.