TRON: Fuzzing Linux Network Stack via Protocol-System Call Payload Synthesis

Qiang Zhang*,Yifei Chu†, Yuheng Shen†, Jianzhong Liu§, Heyuan Shi‡, Yu Jiang† and Wanli Chang*
* College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
† KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
§ School of Computer Science and Technology, Shandong University, Qingdao, China
‡ School of Electronic Information, Central South University, Changsha, China

***Abstract*—The Linux kernel network stack is a critical component of modern operating systems, widely deployed across platforms and often exposed to untrusted inputs. Its complex and stateful nature makes it a frequent target of security vulnerabilities, particularly those triggered by subtle protocol interactions. While existing fuzzers like syzkaller have demonstrated strong capabilities in discovering kernel bugs, they face challenges in exercising deep protocol logic due to the lack of coordinated inputs and protocol awareness. In this paper, we present TRON, a tool designed for fuzzing the Linux kernel network stack. By synthesizing syscall–packet input sequences based on protocol structure and incorporating runtime feedback, TRON enables the exploration of protocol-dependent state transitions and deep execution paths. Our approach addresses the fundamental challenges in dual-input fuzzing by integrating protocol knowledge with execution feedback. We evaluate TRON on four recent Linux kernel versions and compare it against syzkaller and kernelGPT. The results show that TRON improves branch coverage by 22.9% and 12.1% over syzkaller and kernelGPT, respectively, and discovers 25 previously unknown bugs, 7 of which have been fixed. These results demonstrate the effectiveness of protocol–system call input synthesis in enhancing network stack fuzzing and uncovering hard-to-reach bugs in kernel protocol implementations.**

***Index Terms*—Vulnerability Detection, Kernel Fuzzing, Network Stack**

I. INTRODUCTION

The linux kernel network stack is a critical subsystem that facilitates communication between local and remote entities in modern operating systems. Deployed across a wide range of platforms and frequently exposed to untrusted or adversarial inputs, this subsystem has become a persistent source of severe security vulnerabilities. Due to its privileged execution context and exposure to external traffic, it has become a persistent source of severe security vulnerabilities.

Software testing plays an important role in ensuring the security and reliability of modern operating systems [1]–[3]. Among various testing methodologies, fuzzing [4], [5] has emerged as a particularly effective approach for discovering software bugs and security vulnerabilities. The core idea of fuzzing is to automatically generate a large volume of inputs and observe program behavior for signs of failure, such as crashes or hangs. Its scalability and automation make it well-suited for exploring complex and stateful code paths, especially in low-level system components like the kernel. In particular, syzkaller [6] remains one of the most advanced kernel fuzzers to date. It has been instrumental in discovering more than 6,000 bugs [7] in the Linux kernel, demonstrating the practical impact of fuzzing on large-scale kernel security.

Network stack exhibits complex and stateful semantics. Their behavior is shaped by inputs from both the control plane and the data plane. The control plane, driven by system calls, is responsible for configuring socket states, managing protocol parameters, and initiating communication sessions. The data plane, on the other hand, involves the processing of incoming and outgoing packets, which trigger protocol-specific logic such as handshakes, retransmissions, and state transitions. These two input channels jointly determine the internal state of the protocol stack. Only through carefully crafted sequences that coordinate both system calls and packets can specific protocol states and deep execution paths be reached. This dual-input nature introduces additional complexity to the testing process and poses significant challenges for uncovering subtle and state-dependent vulnerabilities.

Current efforts on fuzzing network stack have made important progress, but several limitations remain. Some techniques consider both control-plane and data-plane inputs, such as TCP-Fuzz [8], which generates coordinated socket operations and network packets. However, its primary focus is on testing user-space implementations, and its evaluation of kernel-space stacks is limited to differential testing without access to internal runtime states. As a result, it cannot effectively leverage kernel-level protocol-state information to guide the fuzzing process. Other approaches, such as VirtFuzz [9], operate directly on kernel network stack but restrict their input space to packet-level data, lacking the ability to explore system-call-driven control flows. Additionally, several classical fuzzers, including AFLNet [10] and Peach [11] are not designed for kernel-space testing and cannot exercise the linux kernel's network stack at all. To address these limitations, we identify two core challenges in fuzzing linux kernel network stack.

The first challenge is to generate semantically valid and context-aware sequences of system calls and packets. Unlike other kernel subsystems, the network stack relies on both inputs to drive its internal state transitions. System calls

✉Wanli Chang is the corresponding author. This research is supported in part by the National Key R&D Program of China (2023YFB4503704).

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00266

configure and manage network resources, while packets trigger protocol-level processing. These two input channels are tightly coupled, and their interactions must follow specific temporal and logical relationships to trigger meaningful behaviors. Capturing such dependencies is essential for reaching deep protocol states and exposing hidden logic flaws. Naively combining inputs often results in early rejection or shallow coverage. To address this, input generation must be guided by protocol-specific knowledge, such as valid message patterns, state transition conditions, and field-level constraints. This knowledge enables the synthesis of input sequences that conform to protocol semantics while preserving generality across different protocol implementations.

The second challenge is to leverage runtime feedback to guide the fuzzing process. The network stack can be naturally modeled as a stateful system, where the execution of system calls and network packets jointly influences internal protocol behaviors. Runtime feedback is used to assess the impact of syscall–packet inputs on execution paths. Such inputs may trigger complex protocol mechanisms, including connection handshakes, retransmissions, and timeout handling, which typically manifest only under specific protocol states. By monitoring runtime code coverage metrics, along with observing execution phenomena such as system call return codes or protocol-level responses, the fuzzer can better prioritize inputs that lead to semantically rich behaviors and adapt mutations accordingly, thereby enhancing the depth and precision of protocol exploration.

To address the aforementioned challenges, we propose TRON, a protocol-aware and feedback-guided fuzzer designed specifically for kernel network stack. This process is further supported by a pattern dictionary, which captures reusable input structures extracted from prior executions or specifications. Second, to improve exploration depth and efficiency, TRON integrates runtime feedback primarily based on code coverage. The mutation process is further informed by execution observations—such as system call outcomes and protocol-induced responses—that reflect the semantic effects of syscall–packet interactions. This feedback guides a coordinated mutation strategy that dynamically adapts input generation based on observed execution behavior. By jointly considering semantic structure and runtime feedback, TRON enables effective exploration of deep protocol states and hard-to-reach execution paths within the kernel network stack.

We evaluated TRON on four recent Linux kernel versions, including 6.12, 6.13, 6.14, and 6.15. Our evaluation shows that TRON discovered 25 previously unknown bugs in the linux kernel network stack. Furthermore, in terms of code coverage, TRON improves over syzkaller and kernelGPT by up to 22.9% and 12.1%, respectively. These results highlight the effectiveness of TRON, which combines system call–packet payload synthesis with runtime feedback to explore protocol logic, especially in subsystems with tightly coupled data and control paths. Our contributions are summarized as follows:

- We propose to synthesise semantically meaningful system call–packet payloads and mutate them based on runtime code coverage feedback, enabling deep exploration of protocol behaviors.
- We implemented TRON, a fuzzer that constructs syscall–packet payloads aligned with protocol interaction semantics, enabling effective testing of Linux kernel network stack components.
- We evaluated TRON on four mainline versions of the linux kernel, uncovering 25 previously unknown bugs, 7 of which have been fixed. We also open source TRON [1] and make it available for practice.

II. BACKGROUND

A. Linux Network Stack

The Linux network stack is a core kernel subsystem that enables communication between applications and network devices via standardized protocol layers [12]. It supports all networking functions in Linux-based systems, from basic packet I/O to complex protocol handling. Given its central role in managing inter-process and inter-host communication, its correctness is critical. Bugs in network stack can cause serious consequences such as remote code execution, denial-of-service, particularly in systems exposed to external networks, such as servers and embedded devices [13].

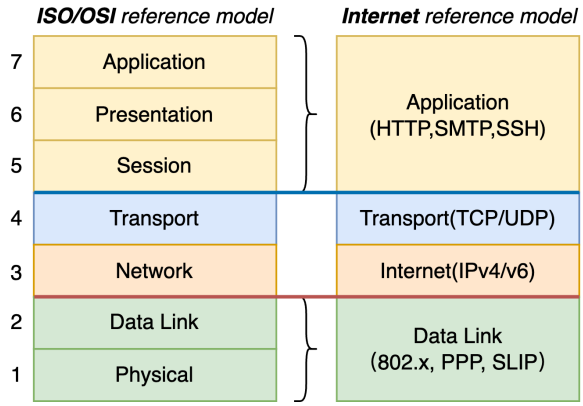


Fig. 1: Layered architectures of the ISO/OSI reference model and the Internet reference model, both used to describe network communication functions.

Figure 1 presents the layered architectures of the ISO/OSI and Internet protocol models, both widely used to describe network communication functions. Each layer provides services to the one above, forming a modular architecture. Due to its layered design, complex protocol semantics, and exposure to untrusted inputs (e.g., remote packets), the network stack has a large attack surface. As a result, it remains a frequent target for both academic research and real-world exploits. Ensuring its robustness requires systematic testing to uncover subtle flaws and validate behavior under diverse conditions. With Linux widely deployed in cloud, mobile, and embedded platforms, securing this subsystem is more urgent than ever.

[1] TRON is available at: https://github.com/rushwow/TRON.

B. Kernel Fuzzing

Fuzzing is an automated testing technique that detects software flaws by feeding invalid or unexpected inputs and monitoring execution behavior. It is widely used in system-level software testing, including compilers [14]–[16], databases [17]–[19], and operating system kernels [20]–[24].

Figure 2 illustrates the overall workflow of kernel fuzzing. The process begins with compiling the kernel into an instrumented image, booted in a virtual machine. Test cases are generated from a corpus and executed, with runtime feedback such as code coverage and crashes used to guide future input mutations. This feedback loop enables systematic exploration of deep and edge-case kernel behaviors.

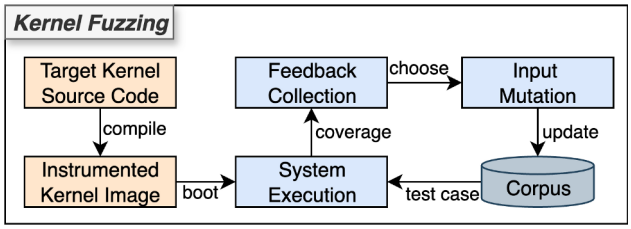


Fig. 2: Overall workflow of kernel fuzzing, where test cases are generated from the target kernel source, executed in an instrumented kernel image, and iteratively mutated based on runtime feedback, including code coverage, to enhance exploration and vulnerability discovery.

Syzkaller is a state-of-the-art kernel fuzzer that uses Syzlang [25], a domain-specific language, to describe system calls and kernel resources. Based on these specifications, it generates valid programs and mutates them during testing. It is designed to target deep kernel logic via system call sequences and collect runtime feedback to drive input evolution. It has been particularly effective in finding bugs in areas such as memory management, IPC, and networking. Given the complexity and input exposure of the Linux network stack, kernel fuzzing plays a crucial role in ensuring its security. Tools like syzkaller automate the generation and mutation of system call sequences to reach diverse execution paths. As of 2017, Syzkaller has reported over 6,000 kernel bugs across multiple subsystems, demonstrating its practical impact.

III. Motivation

The Linux kernel network stack is inherently complex, involving tightly integrated components such as transport protocols, routing logic, and filtering mechanisms. Its runtime behavior is influenced not by static configurations alone, but by a combination of dynamic interactions from both control-plane and data-plane inputs. Control-plane operations configure socket-level states and protocol parameters, while data-plane packets invoke protocol handlers and trigger internal transitions. These dual input streams jointly shape the protocol stack's state, which in turn determines how subsequent inputs are processed. As a result, the execution paths within the stack evolve dynamically and may expose vulnerabilities only under specific input interleavings or protocol states. This tight coupling between internal state and input behavior presents fundamental challenges for vulnerability discovery.

However, existing kernel fuzzing techniques largely treat the system call interface as the only input surface. Most current approaches, such as syzkaller and its variants, mutate and execute syscall sequences in isolation, without considering how packet-level inputs interact with those syscalls to jointly affect protocol state. This single-channel view therefore misses a substantial class of semantic vulnerabilities that arise in the cross-layer transitions between the control and data plane. In particular, when packet inputs arrive at runtime and modify protocol fields, they can drive the kernel into protocol states that are not reachable through syscalls alone, thereby enabling exploration of deeper protocol logic.

At the same time, conventional protocol fuzzing typically focuses on driving message exchanges at the application layer. By contrast, our work emphasizes the kernel network stack, where packet handling is closely tied to syscall-established context and kernel-maintained state. Some vulnerabilities only become observable when system calls first establish particular socket conditions and subsequently injected packets trigger protocol transitions under those exact conditions. For example, certain error-handling branches or uncommon state transitions may depend on precise interactions between these two input channels. These observations motivate a dual-end fuzzing approach that jointly considers syscalls and packets to more effectively exercise cross-layer behaviors within the kernel protocol stack.

To illustrate the limitation and underscore the critical role of interactions between system calls and packet inputs in shaping protocol state, we use CVE-2025-38192 as a motivating example. As shown in Figure 3, the vulnerability is triggered by a specific sequence of system call and packet inputs. Initially, system calls such as socket() and sendto() set up the communication parameters, and the packet follows the expected IPv6 processing path. However, under certain conditions, when a specially crafted packet is injected after the ip_local_out() call but before ip6_rcv_core(), the vulnerability may manifest. Specifically, the packet's type field is altered from IPv6 to IPv4, which causes the kernel, having initially processed the packet under the IPv6 context, to mistakenly handle it using the IPv4 handler. This mismatch in protocol state transitions leads to an inconsistent internal state and eventually triggers memory corruption within ip6_rcv_core().

The vulnerability is caused by the integration of system calls and packet inputs that lead to an unexpected transition between protocol handling logic. When the packet's type field is modified between ip_local_out() and ip6_rcv_core(), the system call sequence initially sets the packet's protocol context to IPv6. Upon packet injection, however, the modified packet is misinterpreted under the IPv4 handler, causing a state transition that cannot be triggered by syscalls alone. This interaction between the system call and packet input is what leads to the bug and demonstrates the need for fuzzing

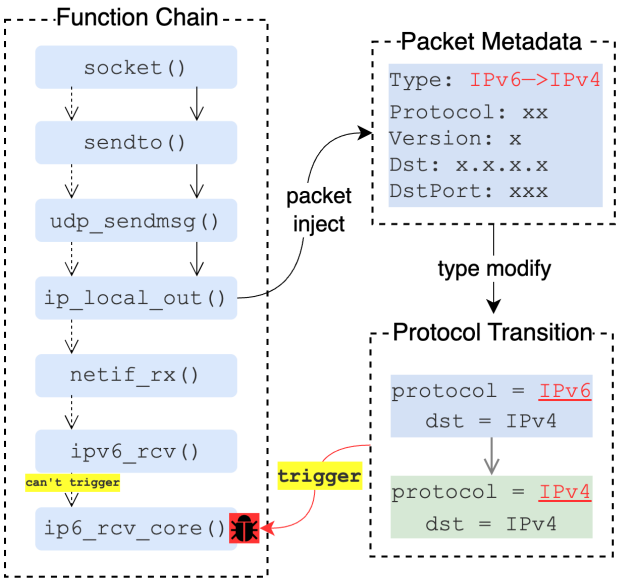


Fig. 3: Illustration of CVE-2025-38192. This vulnerability is triggered by a synthesized sequence combining a system call that injects a packet and a subsequent type modification at the packet level. The interplay causes the packet to be misinterpreted by ip6_rcv_core(), leading to a null pointer dereference. The case highlights how jointly crafted syscall–packet inputs can drive unexpected protocol logic and expose deep-state bugs in the network stack.

methods that jointly consider system calls and packet inputs.

This case exemplifies a class of vulnerabilities caused by the joint effect of system call and packet interactions on protocol state transitions. Unlike traditional bugs that can be triggered by malformed packets or syscalls in isolation, this vulnerability underscores the importance of considering both inputs together during fuzzing. Effective fuzzing must generate semantically valid sequences of both system calls and packet inputs that align with the expected protocol behavior. This requires understanding the dependencies between control and data plane inputs to expose deep protocol logic that cannot be reached by syscalls or packets alone.

Challenge 1: Synthesizing valid syscall-packet payload. Triggering protocol-specific vulnerabilities requires syscall and packet inputs to jointly satisfy protocol semantics. Inputs must be crafted to align with expected state transitions and interleaved in meaningful ways. Naive or isolated mutations often violate protocol constraints and are rejected before reaching deep logic. The challenge lies in generating inputs that combine both channels coherently to activate vulnerable paths within the network stack.

Challenge 2: Leveraging runtime feedback to guide fuzzing. The network stack exhibits dynamic behavior shaped jointly by syscall-packet payloads. While syscalls configure socket and protocol states, packets trigger protocol-specific handlers that may lead to deep and conditional logic. Observing runtime feedback from syscall–packet interactions helps uncover protocol execution paths that are otherwise difficult to reach, thereby improving the effectiveness of network stack fuzzing. By monitoring execution effects induced by these interactions, fuzzers can adapt input generation strategies to more effectively explore conditional logic, cross-layer transitions, and subtle protocol-state changes that are not observable through static analysis alone.

IV. Design

We propose TRON, a kernel fuzzer that synthesises semantically meaningful syscall-packet sequences, guided by runtime feedback in fuzzing execution, to explore the deep state space of network stacks. As shown in Figure 4, TRON follows a structured fuzzing workflow that consists of three main stages. First, it extracts protocol-specific knowledge and analyzes syscall relations to synthesize valid and context-aware payloads. Second, the synthesized payloads are executed in a fuzzing environment, where TRON collects runtime feedback that includes code coverage and protocol states. Finally, based on the collected feedback, TRON performs mutations to generate new test cases. These three processes form a closed-loop fuzzing workflow that continuously drives exploration toward complex and state-sensitive behavior in the kernel network stack. Unlike stateful protocol fuzzers that mutate message sequences in user space, TRON injects packets at syscall-marked positions inside the kernel so that messages are processed under the precise syscall-established context (e.g., bound or connected sockets, device/offload modes). In real kernels, routing, netfilter hooks, loopback short-circuiting, and driver-specific paths can alter how packets traverse the stack; certain paths are reachable only when a particular syscall prefix has established an in-kernel state. By coordinating syscall sequences and packet construction within a unified execution, TRON systematically exercises control–data coupled paths that are difficult to reach via remote injection alone, complementing protocol fuzzers rather than replacing them.

A. Semantics-Aware Payload Synthesis

To synthesize semantically meaningful syscall-packet payloads, TRON first extracts protocol-specific knowledge to construct a pattern dictionary. Then, TRON performs relation analysis to examine syscall sequences and identify valid insertion points where packets can be injected without violating the protocol logic. By combining these two sources of guidance, TRON synthesizes payloads that respect the protocol semantics and are more likely to trigger deep state-sensitive behavior in the kernel network stack.

1) Protocol Knowledge Extraction. To enable the network stack fuzzing, TRON extracts protocol-specific knowledge into a structured pattern dictionary, which encodes key semantic elements of the target protocol and provides essential guidance for payload synthesis and mutation.

First, the dictionary defines protocol-specific message formats, including field order, legal ranges, and optional fields.

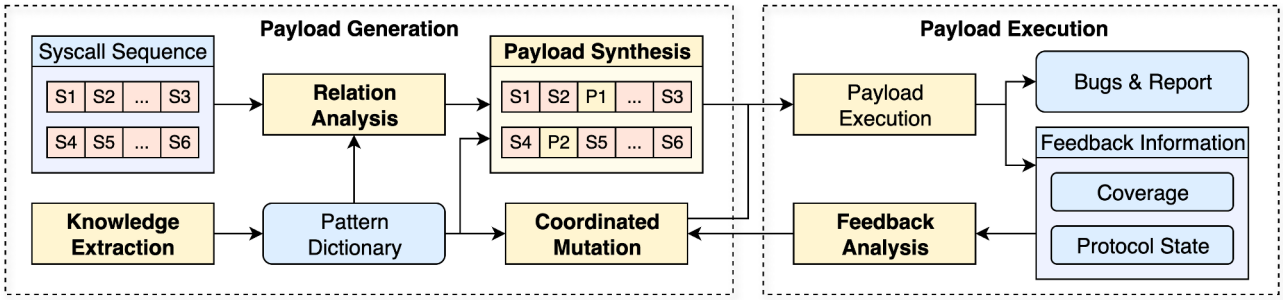


Fig. 4: Overview of TRON: TRON first captures syscall sequences and performs relation analysis to identify appropriate insertion points for data packets. In parallel, it extracts protocol knowledge to construct a pattern dictionary. Guided by the collected relations and protocol patterns, TRON synthesizes syscall-packet sequences for payload execution. During execution, TRON collects bug reports as well as feedback information. This feedback is analyzed to mutate for further fuzzing.

For example, in TCP, it describes the layout of ports, sequence numbers, and control flags. These definitions allow TRON to construct structurally valid packets that conform to protocol syntax. Second, the dictionary defines injection constraints for each message type, such as the required socket state or the surrounding syscall context. These constraints are used to filter out semantically invalid injection attempts during payload synthesis. Third, it includes a static transition priority table derived from the protocol's abstract state machine. Each protocol state is assigned a priority value equal to the number of distinct states directly reachable from it via a single transition. which reflects the local branching potential of each state and is later used to guide mutation by favoring transitions that may lead to richer state exploration.

To extract these components, TRON uses a semi-automated pipeline. Message formats are derived from RFCs and curated packet traces, aligned using annotated templates that define field names, types, ranges, and dependencies. Injection constraints are inferred by correlating syscall context with successful packet delivery observed in execution traces, identifying the minimal conditions required for valid injection. The protocol state machine is extracted from specifications using pattern-based scripts. From this, TRON builds a transition priority table by counting the number of directly reachable states from each node. All elements are compiled into a reusable dictionary that can be extended with new protocol templates and models.

2) Relation Analysis. Given a syscall sequence, TRON performs relation analysis to identify valid positions where packet injection is structurally and semantically permissible. This process ensures that inserted packets respect the logical flow of socket setup and comply with protocol-specific constraints. TRON begins by grouping syscalls by file descriptor (*FD*). Each *FD* corresponds to a socket instance, and all related calls are organized into a per-socket action trace. These traces capture the sequence of operations such as `socket()`, `bind()`, `connect()`, `listen()`, `accept()`, and data-transfer calls. By analyzing the order and semantics of these operations, TRON reconstructs the logical evolution of each socket in the control plane.

To determine valid injection points, TRON evaluates protocol-specific constraints defined in the pattern dictionary. Each constraint specifies the minimal syscall context under which a packet can be injected, formulated as symbolic predicates over the syscall trace prefix. For example, a TCP data packet may require that the socket has completed a connection setup and has not been subsequently closed, i.e., `connected(fd)` is true while `closed(fd)` is false. Although the constraint logic varies by protocol, evaluation is performed generically over the trace.

For each syscall trace, TRON performs a forward scan. At each position, it evaluates the relevant constraints to determine whether any known packet type is admissible. If so, the position is marked as a valid injection point. These annotated locations represent candidate positions for subsequent packet insertion. This analysis is performed offline and cached per seed input. The result is a preprocessed syscall sequence with marked injection points, which will later guide the construction of semantically valid syscall-packet payloads.

3) Payload Synthesis. With marked injection points identified through relation analysis, TRON proceeds to generate concrete packets and insert them into the corresponding syscall sequences. This process results in unified syscall-packet payloads that exercise both control and data plane logic in the network stack. An overview is shown in Figure 5.

TRON first selects an appropriate packet template from the pattern dictionary. Each template specifies the layout of protocol fields (e.g., flags, ports, sequence numbers), value constraints, and optional extensions. Guided by the protocol knowledge embedded in the template, TRON instantiates each field to produce a valid yet diverse packet. Header fields are populated with legal values, and optional fields such as flags or TCP options are varied to increase behavioral coverage. If the protocol allows, corresponding payload data is also generated.

The generated packet is then inserted into the syscall trace at the corresponding marked position. As illustrated in Figure 5, the TCP syscall sequence starting with socket() and followed by bind(), connect(), and read() contains a position identified

during relation analysis TRON generates a TCP packet using the selected template and inserts it at that point, producing a combined syscall-packet sequence. This hybrid input simultaneously exercises control-plane transitions and data-plane logic within the kernel, increasing the likelihood of reaching deep or vulnerability-prone code paths.

Throughout this process, TRON ensures that the generated packets comply with the protocol-specific message formats, as defined in the pattern dictionary. By decoupling injection point selection from packet construction, the system preserves semantic correctness while enabling flexible and diverse test generation. In practice, the pattern dictionary is built by parsing RFC specifications into annotated templates (field order, types, legal ranges, and dependencies) and aligning them with curated packet traces to validate frequent options and concrete values. Minimal injection conditions are expressed as symbolic predicates over the syscall trace prefix, e.g., `connected(fd)`, `bound(fd)`, `listening(fd)`, `not_closed(fd)`. During relation analysis, TRON scans each per-socket syscall sequence and evaluates these predicates to mark admissible injection points for a given template. This explicit representation documents the minimal syscall context required for valid packet delivery while keeping payload generation diverse (RFC-derived legal values, trace-derived seeds, and boundary values), and ensures semantic correctness for the combined syscall–packet input.

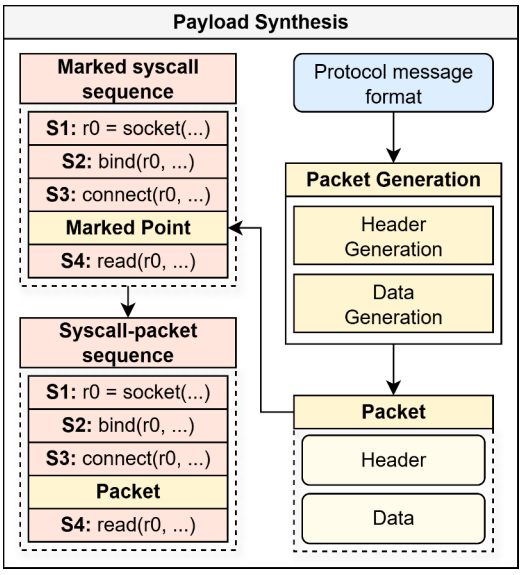


Fig. 5: Overview of the payload synthesis process in TRON, where a selected packet template is instantiated and inserted into a syscall trace, generating a unified syscall-packet sequence that exercises both control and data plane logic in the network stack.

B. Runtime Feedback Collection and Analysis

After generating syscall-packet payloads, TRON executes them within an instrumented kernel environment and collects runtime feedback to guide further fuzzing. This feedback captures both potential vulnerabilities and runtime signals that characterize execution depth and behavioral diversity. In particular, TRON monitors code coverage and protocol state transitions to identify inputs that exercise meaningful paths through the network stack. Coverage and protocol feedback are jointly used to prioritize future mutations. While KCOV ensures systematic expansion into unexplored code regions, protocol state transitions reflect semantic advancement, such as TCP moving from SYN_SENT to ESTABLISHED. Each observed transition is mapped to the corresponding syscall-packet input, and its contribution to exploration depth is quantified through transition scores.

1) Payload Execution and Runtime Feedback. To avoid host-guest synchronization overhead during payload execution, TRON employs an in-guest testing model in which the guest transmits packets to itself. To ensure the injected packets traverse the complete network stack rather than being short-circuited by the loopback interface, the destination is set to the guest's routable IP address instead of using loopback address.

Syscall–packet payloads are executed within an instrumented virtualized environment. A guest-resident executor is responsible for executing the system call sequence. Upon encountering a marked injection point, it constructs the corresponding packet according to the pattern dictionary and transmits it to the kernel network stack. This unified execution context ensures that each packet is injected in a temporally and contextually valid state, maintaining semantic consistency between the control and data planes.

During execution, TRON monitors the system for crashes, warnings, and hangs to detect potential vulnerabilities. It simultaneously collects runtime feedback, including code coverage and state transitions. Code coverage is captured using KCOV, which tracks basic blocks and control-flow edges exercised during syscall and packet processing. Protocol state transitions are observed by tracing kernel-level state variables. For stateful protocols like TCP, TRON uses in-kernel probes (e.g., kprobes) to monitor fields such as `sk_state`.

2) Feedback Analysis and Mutation. After each syscall-packet sequence is executed, TRON analyzes the collected runtime feedback to determine its contribution to behavioral diversity and potential for uncovering deeper kernel logic. Feedback is categorized along two complementary dimensions, coverage and protocol state transitions.

For coverage analysis, inputs that execute previously unvisited basic blocks are considered valuable and incorporated into the fuzzing corpus, enabling systematic expansion into unexplored code regions. Protocol feedback provides complementary information to traditional coverage by reflecting semantic progress within the protocol state machine. During execution, TRON monitors protocol state transitions, for example from `SYN_SENT` to `ESTABLISHED`, and associates each observed transition with the syscall-packet sequence that triggered it. These transitions are represented as directed edges in the protocol state graph. A dynamic transition map is maintained to record the mapping between transitions and their

Algorithm 1: Feedback-Guided Mutation

Input: Dynamic transition map $\mathcal{T}$, Pattern dictionary $\mathcal{D}$
Output: Mutated syscall-packet sequences

1 **Function** *RankInputs*($\mathcal{T}, \mathcal{D}$)**:**
2 $\mathcal{Q} \leftarrow []$;
3 **for each** *input sp in* $\mathcal{T}$ **do**
4 $T \leftarrow \mathcal{T}[sp]$;
5 $score \leftarrow \sum_{t \in T} \Big(w_s \cdot \mathcal{D}.\textsc{Priority}(t) + w_f \cdot \log(1 + \textsc{Count}(t)) + w_c \cdot \Delta_{\text{cov}}(t) \Big)$;
6 $\mathcal{Q}.\textsc{Insert}(sp, score)$
7 **return** $\textsc{SortDescending}(\mathcal{Q})$
8 **Function** *Mutate*($\mathcal{Q}, \mathcal{D}$)**:**
9 $\mathcal{M} \leftarrow \emptyset$;
10 **for each** *input* $sp \in \mathcal{Q}$ **do**
 `// Mutate syscall sequence`
11 **for each** *syscall* $c \in sp.s$ **do**
12 **if** $\textsc{Random}() < p_{arg}$ **then**
13 $c.\texttt{arg} \leftarrow \textsc{MutateArg}(c)$
14 **if** $\textsc{Random}() < p_{insert}$ **then**
15 $sp.s \leftarrow \textsc{InsertSyscall}(sp.s)$
16 **if** $\textsc{Random}() < p_{reorder}$ **then**
17 $sp.s \leftarrow \textsc{ReorderSyscalls}(sp.s)$
18 $template \leftarrow \mathcal{D}.\textsc{GetTemplate}(sp.p.\texttt{type})$;
19 **for each** *field* $f \in template$ **do**
20 **if** $\textsc{Random}() < p_{field}$ **then**
21 $sp.p.f \leftarrow \textsc{MutateField}(f, \mathcal{D})$
22 **if** $\textsc{Random}() < p_{payload}$ **then**
23 $sp.p.\texttt{data} \leftarrow \textsc{MutatePayload}(sp.p.\texttt{data})$
24 execute sp and collect KCOV edges and protocol state transitions;
25 update $\mathcal{T}$ with new transitions and Δ_{cov};
26 **if** *every* N *executions* **then**
27 **for each** *operator* $x \in \{arg, insert, reorder, field, payload\}$ **do**
28 $p_x \leftarrow \textsc{Clip}(p_x \cdot (1 + \alpha(s_x - \bar{s})), p_{\min}, p_{\max})$
29 $\mathcal{M} \leftarrow \mathcal{M} \cup sp$
30 **return** $\mathcal{M}$

corresponding input sequences.

Algorithm 1 illustrates TRON's feedback-guided mutation procedure. The process begins by ranking existing syscall-packet inputs based on their ability to trigger protocol transitions and expand coverage (lines 1–7). Each input is mapped in the dynamic transition map $\mathcal{T}$ to a set of observed transitions. For each transition t, a score is calculated by combining three factors: (1) structural importance $w_s \cdot \mathcal{D}.\textsc{Priority}(t)$, (2) observation frequency $w_f \cdot \log(1 + \textsc{Count}(t))$, and (3) coverage gain $w_c \cdot \Delta_{\text{cov}}(t)$ obtained from KCOV. These scores are aggregated to produce a prioritized queue $\mathcal{Q}$ that favors inputs with higher semantic impact.

TRON then mutates the ranked inputs (lines 8–27). At the syscall level, arguments are mutated with probability p_{arg}, new syscalls may be inserted (p_{insert}), and orderings shuffled (p_{reorder}). At the packet level, fields are mutated with probability p_{field} using domain-specific constraints, and payload data is varied independently (p_{payload}). After mutation, the sequence is executed, and runtime feedback—coverage edges and protocol transitions—is collected and fed back into $\mathcal{T}$. To adaptively guide fuzzing, mutation probabilities are updated every N=256 executions according to each operator's relative success rate in producing new coverage or transitions, with clipping bounds $p_{\min}$=0.01 and $p_{\max}$=0.90.

By integrating structural scoring, runtime feedback, and adaptive operator selection, this procedure ensures that mutation is not only coverage-driven but also guided by protocol semantics. This enables the system to prioritize inputs that meaningfully advance both control-flow exploration and state-machine traversal, resulting in deeper and semantically valid fuzzing of the kernel network stack.

V. Implementation

We implemented TRON in Python and Golang by extending syzkaller to support network stack fuzzing through the synthesis of syscall and packet payloads. We developed a static analyzer in Python to identify socket usage contexts from syzkaller-generated syscall traces. Injection points are validated using protocol constraints extracted from RFC-derived templates [26], which specify required socket states and expected packet structures. Furthermore, we extended syzkaller's executor to support protocol-aware packet injection using PF_PACKET sockets [27], enabling raw packets to traverse the kernel's network stack for comprehensive protocol fuzzing. We used the `scapy` [28] library to generate protocol-compliant packets and synchronized their injection with syscall execution using in-guest markers. To monitor runtime behavior, we leveraged lightweight kprobes [29] attached to socket-related functions (e.g., tracking `sk_state`), and used KCOV to collect edge coverage.

TRON has been deployed to UnionTech's UOS CI/CD testing pipeline. In production, it continuously generates and executes syscall–packet test cases against development kernels, enabling long-term, automated discovery of deep protocol logic bugs under real-world conditions.

VI. Evaluation

A. Experiment Setup

Experiments were conducted on a server with 128 CPU cores and 32 GiB RAM, running Linux as the host OS. We selected Linux kernel versions v6.12, v6.13, v6.14, and v6.15 for evaluation, with v6.15 being the latest mainline release at the time. All kernels were compiled with the same configuration, enabling KCOV [30] for coverage collection and KASAN [31] for memory safety checks. KCSAN [32] was additionally enabled in control settings to detect data races.

We compared TRON with syzkaller [6] and kernelGPT [33], using a unified syzlang syscall specification across tools. Each experiment was executed in an isolated QEMU [34] virtual machine with 2 CPU cores and 2 GiB memory. To reduce variance from nondeterministic fuzzing behavior, each configuration was repeated five times for 24 hours. Reported results are averaged across all runs.

TABLE I: TRON has discovered 25 previously unknown bugs. The third column shows the kernel module in the Linux kernel that contains the bug, and the rest of the columns illustrate the bug index, kernel version, bug location, bug type, and bug description of the bugs, respectively.

Index	Version	Module	Bug Location	Bug Type	Bug Description
1	6.12	net/netlabel/netlabel_kapi.c	netlbl_conn_setattr()	null-ptr-deref	uninitialized socket option leads to null pointer dereference
2	6.12	net/ipv6/calipso.c	calipso_sock_setattr()	null-ptr-deref	null socket option dereference triggers IPv6 label handling failure
3	6.12	net/core/neighbour.c	neigh_timer_handler()	logic error	improper neighbor timer handling causes logic error
4	6.13	net/core/rtnetlink.c	rtnl_newlink()	deadlock	unsynchronized network configuration operations lead to deadlock
5	6.13	net/core/sock.c	__sk_free()	logic error	socket resource deallocation logic triggers error
6	6.14	net/netfilter/core.c	nf_hook_slow()	logic error	improper logic in nf_hook_slow leads to packet misprocessing
7	6.14	net/wireless/core.c	cfg80211_dev_free()	use-after-free	dangling pointer access in cfg80211_dev_free triggers memory error
8	6.14	net/mac80211/tx.c	ieee80211_beacon_update_cntdwn()	logic error	invalid beacon countdown state triggers logic error
9	6.14	net/ipv4/ipmr.c	ipmr_rules_exit()	use-after-free	improper multicast table cleanup triggers use-after-free error
10	6.14	net/wireless/nl80211.c	nl80211_set_wiphy()	use-after-free	invalid pointer access during wiphy netns transition
11	6.14	/net/ipv4/tcp_ipv4.c	tcp_v4_rcv()	deadlock	synchronized TCP processing causes deadlock
12	6.14	net/ipv4/tcp_input.c	tcp_conn_request()	null-ptr-deref	null socket reference in TCP connection request triggers memory error
13	6.14	/drivers/net/ethernet/intel/e1000/e1000_main.c	e1000_close()	deadlock	improper lock ordering causes deadlock
14	6.14	net/ipv4/ipmr.c	ipmr_net_exit_batch()	logic error	improper multicast routing cleanup triggers logic error
15	6.14	net/ipv4/udp.c	udp_sendmsg()	logic error	improper skb allocation during UDP send leads to logic error
16	6.15	drivers/net/ethernet/intel/e1000/e1000_hw.c	e1000_read_phy_reg()	deadlock	synchronized net device operation leads to deadlock
17	6.15	net/mac80211/scan.c	ieee80211_inform_bss()	use-after-free	invalid reuse of freed BSS entry triggers memory error
18	6.15	net/unix/af_unix.c	unix_dgram_sendmsg()	logic error	improper socket message routing triggers logic error
19	6.15	net/ipv4/igmp.c	igmp_mc_init()	logic error	protocol initialization inconsistency induces logic error
20	6.15	net/ipv6/af_inet6.c	inet6_release()	logic error	improper socket teardown logic causes logic error
21	6.15	net/ipv6/ip6_input.c	ip6_mc_input()	use-after-free	invalid buffer access leads to use-after-free in ip6_mc_input
22	6.15	net/mac80211/rx.c	ieee80211_rx_napi()	logic error	packet handling in ieee80211_rx_napi triggers logic error
23	6.15	net/unix/af_unix.c	unix_create1()	double free	duplicate deallocation in unix_create1 triggers memory error
24	6.15	net/xfrm/xfrm_user.c	xfrm_send_state_notify()	logic error	inconsistent state notification logic triggers logic error
25	6.15	net/packet/af_packet.c	run_filter()	logic error	faulty packet filtering in net receive operation causes logic error

B. Bug Detection Capability

To assess the effectiveness of TRON in detecting kernel vulnerabilities, we deployed TRON, Syzkaller, and KernelGPT to fuzz the four latest mainline Linux kernel versions during our experiment period.

TRON uncovered 25 previously unknown vulnerabilities, 7 of which were fixed. Table I lists the affected kernel version, module, bug location, type, and a brief description. These bugs span multiple subsystems in the Linux network stack, including protocol layers (e.g., IPv4, IPv6, TCP, UDP, IGMP), network security modules (e.g., NetLabel, CALIPSO, XFRM), and network drivers (e.g., Intel e1000, mac80211) responsible for lower-layer data path handling. This variety illustrates TRON's ability to explore intricate paths within the network stack, covering protocol states, socket-layer behaviors, and coordination between high-level protocol logic and low-level driver execution.

TRON's effectiveness arises from its joint synthesis of system calls and protocol-level packets. By combining control-plane syscalls with data-plane inputs, it enables valid protocol transitions and drives the stack into internal states often missed by syscall mutation alone.

C. Case Study

Bug Case 1. Listing 1 illustrates a deadlock bug (Bug #13 in Table I) in Linux kernel v6.14's network interface control path, caused by circular locking on `rtnl_mutex` shared by a userspace-triggered shutdown and a driver-internal reset.

The shutdown sequence begins with a netlink system call invoking `rtnl_setlink()` (Line 1), which acquires `rtnl_mutex` and eventually calls `e1000_close()` (Line 13), leading to `e1000_down()` (Line 21). This function waits for any ongoing reset via `cancel_work_sync()`.

Meanwhile, a crafted packet can induce an abnormal link state and asynchronously trigger `e1000_reset_task()` (Line 28), which attempts to reacquire `rtnl_mutex`. When both paths overlap, the mutex is held by the shutdown path while the reset task blocks, resulting in a deadlock. Listing 1 shows this interleaving: the shutdown path holds `rtnl_mutex` and waits for the reset, while the reset task waits for the same lock.

Crucially, this deadlock is only exposed when packet input and system calls are exercised jointly. Packet input activates asynchronous driver logic, such as resets, that system calls alone cannot reach. Since packet processing occurs in interrupt context, it accesses internal paths unreachable from process-context system calls. Our synthesis of packet and syscall inputs enables exploration of such cross-layer interactions,

revealing concurrency bugs. If triggered, this deadlock can stall administrative tasks like interface reconfiguration, leading to persistent disruption in multi-threaded environments.

```
static int rtnl_setlink(struct sk_buff *skb, ...)
{
  // holds rtnl_mutex during shutdown
  rtnl_nets_lock(&rtnl_nets);
       ...
  dev = __dev_get_by_index(net, ifm->ifi_index);
  if (dev)
    do_setlink(...);
  // handles netlink-triggered interface change
  rtnl_nets_unlock(&rtnl_nets);
}

int e1000_close(struct net_device *netdev)
{
  set_bit(__E1000_DOWN, &adapter->flags);
  // called under rtnl_mutex
  e1000_down(adapter);
  return 0;
}

void e1000_down(struct e1000_adapter *adapter)
{
  netif_tx_disable(adapter->netdev);
  // waits for reset task via cancel_work_sync
  e1000_down_and_stop(adapter);
}

static void e1000_reset_task(struct work_struct
    *work)
{
  // reset triggered by coordinated packet input
  rtnl_lock();
  e1000_reinit_locked(adapter);
  // potential deadlock if rtnl_mutex held
  rtnl_unlock();
}
```

Listing 1: A deadlock in `e1000_close()` due to circular locking on `rtnl_mutex`. The shutdown path is triggered by a netlink system call, while the reset task is scheduled in response to crafted packet input. Their interleaving causes both paths to block on the same mutex, revealing a deadlock.

Bug Case 2. Listing 2 shows the patch for Bug #2 in Table I, which fixes a null pointer dereference in the CALIPSO module of Linux kernel v6.12, caused by improper TCP state handling during socket attribute assignment.

The issue lies in `calipso_sock_setattr()`, which assigns CALIPSO labels by retrieving the `struct sock` via `sk_to_full_sk(req_to_sk(req))`. When TCP SYN Cookies are enabled—used to mitigate SYN floods—the connection remains in `SYN_RECV` without creating a full listener socket, making `reqsk->rsk_listener` NULL. Without checking for NULL, dereferencing `sk` leads to a crash. According to TCP semantics, sockets are fully initialized only in the `ESTABLISHED` state. Invoking CALIPSO logic before that may violate socket lifecycle assumptions and cause unsafe memory access.

This bug was triggered in TRON via coordinated packet and syscall inputs that drove the connection into the `SYN_RECV` state. In TCP, transitions such as `LISTEN` to `SYN_RECV` are packet-driven and cannot be reached by syscalls alone. TRON achieves this by injecting crafted packets at precise syscall execution points to guide TCP state transitions. This exposes latent bugs dependent on protocol-specific intermediate states. In this case, CALIPSO logic was activated while the socket structure was still uninitialized—an intermediate state usually bypassed in pure syscall flows. If exploited, the bug could cause a kernel crash or denial-of-service due to a null pointer dereference during early TCP handling.

The patch addresses this bug by adding two null pointer checks in the function `calipso_req_setattr()` and `calipso_req_delattr()`, ensuring safe socket access under SYN Cookie conditions.

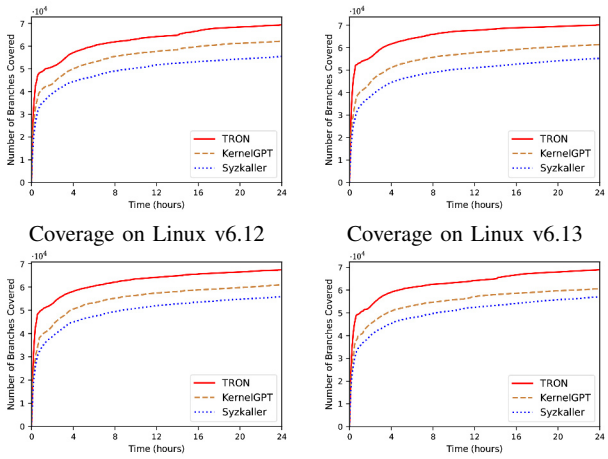
```
--- a/net/ipv6/calipso.c
+++ b/net/ipv6/calipso.c
@@ -1207,6 +1207,10 @@ static int
    ↪  calipso_req_setattr(struct request_sock *req,
          struct ipv6_opt_hdr *old, *new;
          struct sock *sk =
          ↪  sk_to_full_sk(req_to_sk(req));

+         /* sk is NULL for SYN+ACK w/ SYN Cookie */
+         if (!sk)
+                 return -ENOMEM;
+
          if (req_inet->ipv6_opt &&
          ↪  req_inet->ipv6_opt->hopopt)
                  old = req_inet->ipv6_opt->hopopt;
          else
@@ -1247,6 +1251,10 @@ static void
    ↪  calipso_req_delattr(struct request_sock *req)
          struct ipv6_txoptions *txopts;
          struct sock *sk =
          ↪  sk_to_full_sk(req_to_sk(req));

+         /* sk is NULL for SYN+ACK w/ SYN Cookie */
+         if (!sk)
+                 return;
+
          if (!req_inet->ipv6_opt ||
          ↪  !req_inet->ipv6_opt->hopopt)
                  return;
```

Listing 2: The patch introduces null pointer checks to avoid accessing uninitialized socket structures under incomplete TCP connection states.

D. Coverage Comparison

To evaluate the effectiveness of TRON in exercising deeper paths within the Linux network stack, we measured branch coverage on four mainline kernel versions: v6.12, v6.13, v6.14, and v6.15. These versions were selected to represent a sequence of stable upstream releases with continuous network stack evolution, and to ensure comparability with recent

fuzzing studies. Each fuzzing campaign ran for 24 hours and was repeated five times to mitigate statistical variance. The average coverage across all runs is reported. We compared Syzkaller, KernelGPT, and our TRON, respectively. All were evaluated on the same kernel builds, targeting only network-related modules, with coverage collected via KCOV instrumentation sampled every 60 seconds.



Coverage on Linux v6.12 Coverage on Linux v6.13

Coverage on Linux v6.14 Coverage on Linux v6.15

Fig. 6: Branch coverage of TRON, Syzkaller, and KernelGPT on Linux versions 6.12, 6.13, 6.14, and 6.15. During 24 hours of testing, TRON consistently achieved the highest coverage across all versions.

Figure 6 shows the branch coverage trends of all three fuzzers over the 24-hour experiment. In the early phase, all tools show steady growth. As time progresses, TRON continues to increase coverage, while syzkaller and KernelGPT begin to plateau. This improvement comes from TRON's ability to synthesize protocol payloads and system call sequences together, allowing it to trigger protocol states that require specific control–data interactions. Many paths in the Linux network stack are guarded by protocol state machines and are only executed after precise sequences of socket calls and packets. For example, entering the `SYN_RECV` state in TCP requires a valid socket setup followed by a crafted SYN packet. Syzkaller cannot model these transitions due to its random syscall strategy, and KernelGPT, while generating syscall sequences using language models, lacks protocol-level semantics. In contrast, TRON coordinates syscall logic with packet inputs, reaching deeper, stateful paths across the stack by bridging control and data planes.

Table II presents the detailed branch coverage achieved by each tool across the four kernel versions. Compared to syzkaller, TRON improves branch coverage by 24.9% on v6.12, 22.3% on v6.13, 23.7% on v6.14, and 20.8% on v6.15. Against kernelGPT, TRON achieves 11.5%, 10.1%, 13.4%, and 13.7% improvements on the same respective versions. Overall, TRON achieves 22.9% and 12.1% higher coverage than syzkaller and kernelGPT, respectively. These results highlight

TABLE II: Coverage statistics of TRON, Syzkaller, and KernelGPT on Linux over 24 hours. TRON achieves higher coverage across all tested kernel versions.

Version	TRON	Syzkaller	KernelGPT
v6.12	**69337.6**	55481.2(+24.9%)	62172.4(+11.5%)
v6.13	**67517.4**	55229.4(+22.3%)	61356.8(+10.1%)
v6.14	**69121.8**	55856.2(+23.7%)	60971.4(+13.4%)
v6.15	**68940.4**	57036.8(+20.8%)	60629.6(+13.7%)
Overall	**68699.8**	**55900.9(+22.9%)**	**61282.5(+12.1%)**

TRON's capability to reach deep, state-dependent execution paths in the network subsystem by jointly synthesizing system call sequences and protocol-valid packet inputs. This coordinated input strategy allows TRON to activate code guarded by internal protocol state transitions that are typically unreachable through system call fuzzing alone.

In addition to kernel-oriented fuzzers, we further analysed protocol fuzzers such as AFLNet. It focuses on generating and mutating sequences of protocol messages, making it effective at driving user-space protocol implementations through application-level state machines. However, its execution model interacts with the kernel primarily via socket interfaces, which restricts its ability to trigger deeper code paths tied to kernel-maintained protocol states. In contrast, analysis of TRON's design and execution logs shows that it directly exercises the kernel network stack by combining system call operations with semantically constructed packet inputs. This dual-end process allows syscall invocations to first establish kernel states—such as configuring TCP options or transitioning through connection phases—before injecting corresponding packets. As a result, TRON systematically reaches code regions in the kernel protocol stack that remain less accessible to protocol-only fuzzers, for example error-handling routines that are executed only when a specific sequence of system calls sets up an uncommon internal state prior to packet processing.

VII. Related Work

A. Fuzz Testing

Fuzz testing [4], [35], [36] has become a widely used technique for uncovering software vulnerabilities, particularly in large-scale and security-sensitive systems like kernels and protocol stacks. In the kernel domain, syscall-level fuzzing has been especially effective due to the structured and privileged nature of kernel interfaces. Syzkaller [6] is a representative kernel fuzzer that systematically mutates system call arguments and sequences using a coverage-guided approach and a domain-specific language to model syscall specifications. It has led to the discovery of numerous bugs across various kernel components.

To improve the semantic quality of generated syscall sequences, several recent efforts incorporate learning-based or trace-driven techniques. Healer [37] learns influence relations

between system calls and leverages these relations to guide input generation and mutation. KernelGPT [33] uses large language models to automatically synthesize syscall specifications based on knowledge distilled from kernel code and documentation. In addition, MoonShine [38] distills high-quality syscall sequences from real-world execution traces using lightweight static analysis to preserve syscall dependencies, and AFL [39] leverages lightweight instrumentation to mutate inputs and track coverage in both user-space and kernel fuzzing scenarios. These approaches enhance syscall diversity and code coverage, but they primarily focus on the control plane and do not support protocol-driven input construction or packet-level interaction.

In contrast, our work targets the kernel network stack by synthesizing system call sequences together with packet payloads. By integrating both control-plane and data-plane inputs, our approach enables the fuzzer to drive complex protocol state transitions that are unreachable through system calls alone. This coordinated input strategy allows for deeper exploration of stateful behaviors within the network stack and improves the ability to trigger subtle vulnerabilities across protocol layers.

B. Network Stack Testing

Testing of the kernel network stack needs to account for both protocol state transitions and system-level interactions that evolve over time. Unlike syscall fuzzing, which targets static interfaces, network stack testing requires the generation of temporally and semantically valid packet sequences that adhere to protocol logic.

Peach [11] conducts generation-based fuzzing using manually defined grammars, enabling precise input control but limiting scalability. AFLNet [10] adopts a greybox approach, replaying and mutating recorded protocol messages with feedback from server response codes to guide state exploration. BLEEM [40] advances black-box testing by tracking state space evolution across packet sequences and using interaction feedback to steer fuzzing. TCP-Fuzz [8] combines syscall and packet inputs to improve test effectiveness, using transition coverage and differential analysis to uncover semantic bugs in TCP implementations.

Our work differs by synthesizing syscall sequences and protocol-compliant packets. This unified input strategy enables triggering of state-dependent kernel paths that require control–data-plane interaction, exposing vulnerabilities beyond the reach of prior approaches.

VIII. Lesson Learned

System Call–Packet Semantic Alignment is Critical. We found that effective fuzzing of the Linux network stack requires system calls and packet payloads to follow protocol-consistent semantics. For instance, sending data before completing a TCP connection handshake led to the input being silently dropped, preventing deeper code paths from executing. When syscall sequences and packets were generated separately, they often violated protocol expectations. To address this, we synthesized syscall–packet pairs that respected the logical order and dependencies defined by common protocol usage. Ensuring this semantic alignment significantly improved coverage of protocol-specific logic and avoided invalid inputs that would otherwise be ignored.

Valid Socket State is Key to Protocol Execution. In protocol–system call payload synthesis, we found that the behavior of the kernel network stack is closely tied to the internal state of sockets during testing. If system calls or packet injections are performed on sockets that are not in the expected state—for example, before a connection is fully established or after the socket has been closed—the kernel may silently skip the associated protocol logic. These cases typically do not produce errors or logs, but can result in inputs that fail to explore meaningful paths. To improve the accuracy of such fuzzing efforts, future frameworks may benefit from more explicit modeling or tracking of socket state transitions to better align generated inputs with protocol expectations.

IX. Threats to Validity

In evaluating the effectiveness of our approach, we recognize several limitations that may influence the generality of the results. Our method synthesizes system call–packet input combinations based on protocol logic and syscall–packet interactions. However, some protocol behaviors rely on implicit kernel-side state transitions—such as socket progression, timer-triggered responses, or internal acknowledgment handling—that are not always activated by structurally valid inputs. These gaps may prevent certain kernel paths from being exercised, even when inputs follow the expected format. To mitigate this risk, we use execution feedback to filter out inputs that fail to trigger meaningful behavior. Still, due to the lack of observable state indicators in some socket interactions, it remains challenging to distinguish valid-but-ineffective inputs from those that meaningfully reach protocol logic. Enhancing visibility into socket state changes—such as monitoring key ioctl calls or netlink events—may offer a practical way to reduce this ambiguity.

X. Conclusion

In this paper, we present TRON, a tool designed to improve the effectiveness of fuzzing the Linux kernel network stack. TRON synthesizes syscall–packet payloads by combining protocol structure with runtime feedback, enabling it to explore deeper and more complex execution paths within the network stack. We evaluated experiments across four mainline Linux kernel versions, comparing TRON with Syzkaller and KernelGPT. Experimental results show that TRON improves branch coverage by 22.9% and 12.1%, respectively, and uncovers 25 previously unknown bugs, 7 of which have been fixed. These results demonstrate the ability of TRON to explore network stack code paths, thereby increasing the likelihood of exposing previously unknown vulnerabilities.

REFERENCES

[1] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018. [Online]. Available: https://doi.org/10.1145/3182657

[2] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency use-after-free bugs in linux device drivers," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 255–268.

[3] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 811–824. [Online]. Available: https://doi.org/10.1145/3460120.3484798

[4] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[5] L. McDonald, M. I. U. Haq, and A. Barkworth, "Survey of software fuzzing techniques."

[6] D. Vyukov and A. Konovalov, "Syzkaller: an unsupervised coverage-guided kernel fuzzer," 2015, https://github.com/google/syzkaller.

[7] ——, "Syzbot," 2015, https://syzkaller.appspot.com/upstream.

[8] 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)*. USENIX Association, Jul. 2021, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou

[9] 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)*, 2024, pp. 4629–4645.

[10] 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)*, 2020, pp. 460–465.

[11] M. Eddington. (2023) Peach fuzzer platform. [Online; accessed 07-October-2023]. [Online]. Available: http://www.peachfuzzer.com/products/peach-platform/

[12] S. Seth and M. A. Venkatesulu, *TCP/IP Architecture, Design and Implementation in Linux*. Wiley-IEEE Computer Society Pr, 2008.

[13] A. Quach, Z. Wang, and Z. Qian, "Investigation of the 2016 linux tcp stack vulnerability at scale," in *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '17 Abstracts. New York, NY, USA: Association for Computing Machinery, 2017, p. 8. [Online]. Available: https://doi.org/10.1145/3078505.3078510

[14] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpiński, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1017–1032. [Online]. Available: https://doi.org/10.1145/3453483.3454092

[15] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," *SIGPLAN Not.*, vol. 46, no. 6, p. 283–294, Jun. 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993532

[16] C. Zhou, B. Qian, G. Go, Q. Zhang, S. Li, and Y. Jiang, "Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3689757

[17] D. R. Slutz, "Massive stochastic testing of sql," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB '98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, p. 618–622.

[18] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of DBMS with coverage-based guidance," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 4309–4326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/liang

[19] J. Liang, Y. Chen, Z. Wu, J. Fu, M. Wang, Y. Jiang, X. Huang, T. Chen, J. Wang, and J. Li, "Sequence-oriented dbms fuzzing," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 668–681.

[20] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069

[21] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-Aware Task Generation for RTOS Fuzzing," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3477014

[22] Y. Shen, Y. Xu, H. Sun, J. Liu, Z. Xu, A. Cui, H. Shi, and Y. Jiang, "Tardis: Coverage-guided embedded operating system fuzzing," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 41, no. 11, p. 4563–4574, nov 2022. [Online]. Available: https://doi.org/10.1109/TCAD.2022.3198910

[23] J. Liu, Y. Shen, Y. Xu, H. Sun, and Y. Jiang, "Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures," *ACM Transactions on Software Engineering and Methodology*, 2023.

[24] J. Liu, Y. Shen, Y. Xu, H. Sun, H. Shi, and Y. Jiang, "Effectively sanitizing embedded operating systems," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658490

[25] D. Vyukov and A. Konovalov, "Syzlang: System Call Description Language," 2015, https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[26] C. J. Kale and T. J. Socolofsky, "TCP/IP tutorial," RFC 1180, Jan. 1991. [Online]. Available: https://www.rfc-editor.org/info/rfc1180

[27] "Linux packet socket interface (packet(7))," https://manpages.ubuntu.com/manpages/jammy/en/man7/packet.7.html, accessed: 2025-07-26.

[28] P. Biondi, "Scapy," in *CanSecWest Conference*, 2007, accessed: 2025-07-26. [Online]. Available: https://scapy.net/

[29] R. Krishnakumar, "Kernel korner: kprobes-a kernel debugger," *Linux J.*, vol. 2005, no. 133, p. 11, May 2005.

[30] SimonKagstrom, "Kcov," 2013, https://github.com/SimonKagstrom/kcov.

[31] Google, "Kernel address sanitizer," 2013, https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[32] ——, "Kernel concurrency sanitizer," 2013, https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[33] C. Yang, Z. Zhao, and L. Zhang, "Kernelgpt: Enhanced kernel fuzzing via large language models," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 560–573. [Online]. Available: https://doi.org/10.1145/3676641.3716022

[34] F. Bellard, "QEMU, a fast and portable dynamic translator," in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[35] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

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

[37] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, *HEALER: Relation Learning Guided Kernel Fuzzing*. New York, NY, USA: Association for Computing Machinery, 2021, p. 344–358. [Online]. Available: https://doi.org/10.1145/3477132.3483547

[38] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 729–743. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[39] lcamtuf, "American fuzzy lop," 2013, https://lcamtuf.coredump.cx/afl/.

[40] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: packet sequence oriented fuzzing for protocol implementations," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.