NLSaber: Enhancing Netlink Family Fuzzing via Automated Syscall Description Generation

Lin Ma[1], Xingwei Lin[1], Ziming Zhang[2], and Yajin Zhou[1(✉)]

[1] Zhejiang University, Hangzhou, China
{linma,xwlin.roy,yajin_zhou}@zju.edu.cn
[2] Independent Researcher
ezrakiez@gmail.com

Abstract. Recently, security researchers have uncovered a significant number of high-severity vulnerabilities in Netlink families, posing serious threats to overall kernel security. Despite these risks, there are no automated methods available to effectively detect bugs in Netlink families. For example, Syzkaller—a state-of-the-art, general-purpose kernel fuzzer—fails to achieve effective fuzzing for Netlink families because it depends on manually written descriptions that are often incomplete or inaccurate. To address this gap, we present NLSaber, the first specialized tool designed for enhancing Netlink family fuzzing. NLSaber uses static taint analysis to construct parse graphs that model the message parsing process, and then automatically generates complete and accurate fuzzing descriptions based on these graphs. In our evaluation on Linux 6.1.70, NLSaber identified 76 target families, encompassing 865 operations. The generated fuzzing descriptions were significantly more complete (supporting 43% more families) and more accurate (93% vs. 33% accuracy) compared to existing descriptions. Using these generated descriptions, our enhanced fuzzer improved code coverage by 9.1% over Syzkaller in families supported by both tools (and by 40.8% when including Syzkaller-unsupported families). Additionally, NLSaber uncovered 19 previously unknown vulnerabilities, all reported and confirmed, with 12 CVEs assigned.

Keywords: System Security · Linux Kernel · Netlink · Fuzzing.

1 Introduction

Netlink is a socket-based communication interface that facilitates inter-process communication (IPC) between the kernel and userspace processes. Introduced in Linux 2.2, Netlink offers greater flexibility than the traditional ioctl interface, supporting higher throughput and a richer feature set [37]. Initially, Netlink was designed for exchanging networking-related information. It has since been adopted by other kernel subsystems [24] and integrated into operating systems such as FreeBSD [27].

The widespread adoption of Netlink has improved efficiency but also introduced significant security risks. Recently, security researchers have uncovered

numerous high-severity vulnerabilities [29,9,28,15,4] in Netlink families, most of which can lead to privilege escalation, threatening the security of the Linux kernel and millions of devices. Thus, automatically detecting Netlink vulnerabilities before they can be exploited in the wild remains a pressing challenge and an open research problem.

Fuzzing is a promising approach for automatically detecting bugs in OS kernels. For example, Syzkaller [35], a state-of-the-art kernel fuzzer developed by Google, has uncovered thousands of Linux kernel bugs since 2017. However, we observed that Syzkaller fails to achieve effective Netlink family fuzzing. A key limitation is that the syscall descriptions it relies on, which enable structure-aware fuzzing, are incomplete or inaccurate (see Sect. 2). Consequently, Syzkaller often fails to perform as expected, leaving many code paths inadequately tested.

The problem is particularly severe for Netlink families because their inputs, known as Netlink messages, are highly complex. These messages use specialized attribute encoding and require a specific parsing process to extract usable payloads. Notably, Netlink messages have an average nesting-depth [3] five times greater than that of `ioctl` arguments used in device drivers according to the existing descriptions. This complexity makes manual description writing both time-consuming and error-prone, while also rendering existing automated description generation approaches for device drivers[8,33,22,6] ineffective.

To address this gap, we propose a dedicated model called the **parse graph** for systematically analyzing the Netlink message parsing process. This graph, constructed using static taint analysis, captures essential parsing details for recovering the complex Netlink messages. By traversing and translating the parse graph, we generate complete and accurate syscall descriptions, thereby enhancing the effectiveness of Netlink family fuzzing. We implemented the prototype system **NLSaber** and evaluated it on Linux 6.1.70. Cross-validation shows that the generated descriptions are more complete (supporting 43% more families) and accurate (93% vs. 33%) than existing descriptions. Additionally, we assessed the system's effectiveness through comprehensive fuzzing experiments. The enhanced fuzzing improved code coverage by more than 9.1% for common families (and by 40.8% when including Syzkaller-unsupported families). NLSaber uncovered 19 previously unknown vulnerabilities, all have been reported and confirmed, with 12 CVEs assigned.

Contributions In summary, our main contributions are in the following.

- We found that the state-of-the-art fuzzer Syzkaller is ineffective in Netlink family fuzzing due to its incomplete and inaccurate descriptions. Existing device-driver-targeted solutions are difficult to adapt for Netlink due to their inability to analyze the message parsing process.
- We propose a solution that models the Netlink message parsing process using parse graphs constructed with static taint analysis. These graphs capture essential parsing details, enabling the generation of complete and accurate syscall descriptions to enhance Netlink family fuzzing.

[3] Syzkaller types are organized in directed-acyclic graph form. We call the depth of the expanded tree from one type as the nesting-depth for this type.

– We present a prototypical implementation named NLSaber and evaluate its effectiveness on Linux 6.1.70. The evaluation shows that our generated descriptions are more complete and accurate than existing ones. Using these improved descriptions, the enhanced fuzzing improved code coverage by over 9.1% (and by 40.8% when including Syzkaller-unsupported families). Moreover, NLSaber uncovered 19 previously unknown vulnerabilities, with 12 CVEs assigned.

2 Technical Background and Motivation

2.1 Netlink Family and Message Parsing

Netlink facilitates communication between the kernel and userspace processes [25]. Similar to driver interfaces (e.g., char and block device drivers), Netlink provides multiple interfaces, such as `NETLINK_ROUTE`, the foundation of Linux kernel components for packet routing and device hook frameworks, and `NETLINK_NETFILTER`, the foundation for packet filtering and iptables. Each Netlink family registers callback functions (called operations) to handle Netlink messages [4]. These messages, sent from userspace via network syscalls like `send` and `sendmsg`, consist of a fixed-format metadata header followed by a stream of attributes containing userspace payloads in TLV (Type, Length, Value) format [20]. Figure 1 presents an operation snippet for configuring a virtual network interface, along with its Netlink policies and message structure. The header comprises a 16-byte message header (`nlmsghdr`) and a family-specific header (`ifinfomsg`). Each subsequent attribute begins with 4-byte metadata (`nlattr`), followed by the actual payload. Notably, attributes can be nested, meaning a payload may contain another TLV-formatted attribute stream to carry complex data.

To parse the Netlink message and attributes to get the accessible payload, the `rtnl_setlink` function takes a parameter `nlh` (of type struct `nlmsghdr *`), which points to the message header. Then, `nlmsg_data` (L2) retrieves the family-specific header, assigning it to the pointer `ifm`. Next, `nlmsg_parse_deprecated` (L3) parses `nlh` into an array of attributes `tb`. Attribute-type enums, such as `IFLA_IFNAME` and `IFLA_NEW_IFINDEX`, are then used as indices to get specific attribute pointers from the array `tb` (L11,12). Functions like `nla_strscpy` (L11) and `nla_get_s32` (L12) are used to extract the payload from these attributes. For nested attributes, such as `tb[IFLA_XDP]`, the function `nla_parse_nested_deprecated` (L14) is called to parse the nested payload into another attribute array `xdp`. Importantly, during parsing, Netlink families may define validation rules through **Netlink policies**. For example, the `ifla_policy` in Fig. 1 specifies the following constraints: (1) `IFLA_IFNAME`'s payload must be a string shorter than `IFNAMSIZ`; (2) `IFLA_XDP` must be a nested attribute; and (3) `IFLA_NEW_IFINDEX`'s payload must be a signed integer with a minimum value of 1. The function `nlmsg_parse_deprecated` verifies these constraints. If any validation fails, it returns an error and terminates the parsing process. Such validations

[4] Netlink family also transmits messages to userspace via unicast or broadcast. This study focuses on the case where userspace is the sender.

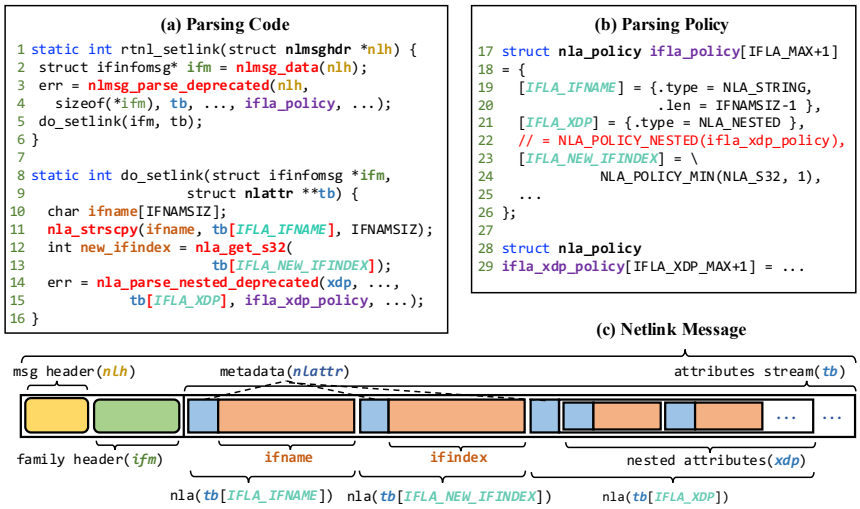

Fig. 1. Simplified code snippet for network link configuration, including Netlink policies and message structure.

are flexible yet robust, making them widely used in the Netlink family code for payloads verification.

2.2 Motivation

Recently, security researchers have uncovered numerous high-severity vulnerabilities in Netlink families [29,9,28]. Notably, over half of the kCTF [15] submissions exploit this attack surface to achieve privilege escalation. Despite this, no specialized tool exists for detecting these vulnerabilities. Even Syzkaller, the state-of-the-art advanced general-purpose kernel fuzzer, fails to achieve effective fuzzing of Netlink families due to *incomplete* and *inaccurate* descriptions. In the following, we elaborate on these two issues, which motivate our work to generate *complete* and *accurate* descriptions to enhance Netlink family fuzzing.

Issue I. Current descriptions are incomplete. Our analysis on Syzkaller's descriptions reveals the following: (1) Since 2020, Syzkaller has added descriptions for nine new Netlink families, yet eight [5] of them have already existed in the kernel for over two years. (2) Five critical vulnerabilities [6] were reported in kCTF 2023, but their corresponding descriptions were not updated in Syzkaller until over a year later (September 2024). Furthermore, our evaluation shows that over 23 Netlink families still lack descriptions (see Sect. 4.1), with 17 of them present in the kernel for more than five years.

Issue II. Current descriptions are inaccurate. Although Syzkaller's descriptions are written by experts, they contain errors. As evidence, we identified and reported 85 errors (see Sect. 4.1) in Netlink-related descriptions. Besides, we

[5] netlabels, nl80211, ipset, nfc, batman-adv, smc, nldev, mptcp
[6] CVE-2023-4147,5197,4569,4015,5345

also identified many *redundant attributes* (either unused or repeated). Specifically, since there is no available approach to analyze the message parsing process, e.g., the parsing code listed in Fig. 1, the existing descriptions choose to include all specified attributes in the parsing policies into their message structure definitions, regardless of whether their attributes are accessed or not. Consequently, these inaccuracies lead to inefficient generation and mutation, hindering fuzzing performance. For example, in our experiment on the Netlink family in Fig. 1, only 19.3% of the attributes sent by Syzkaller were actually accessed, while over 20.1% were redundant.

Our Solution. We found that both issues stem from the complexity of Netlink messages. Compared to ioctl arguments, Netlink messages exhibit significantly deeper nesting (average depth of 15 vs. 3), as shown in existing descriptions. The complexity makes previous approaches ineffective. For example, we tested SyzGen++ [6], which uses symbolic execution to infer the input structure of the operation shown in Fig. 1. However, the analysis either times out or runs out of memory. To address this, we propose a specialized and systematic solution. It analyzes how these messages are parsed, enabling the generation of complete and accurate descriptions to enhance fuzzing for Netlink families. These automatically generated descriptions solves above issues. First, they remain up to date with new or modified kernel code. Using them, our enhanced fuzzing uncovered 12 vulnerabilities in code unsupported by existing descriptions. Second, the improved accuracy of the descriptions enhances fuzzing efficiency. In our evaluation, our system not only improves code coverage but also detects previously unknown bugs. Notably, it discovered a previously undetected null pointer dereference in the Linux kernel, which had gone unnoticed for over 12 years, despite five years of testing with related descriptions in Syzkaller (see Sect. 4.3).

3 System Design & Implementation

Entrypoint Identification. Similar to prior work [8] that identifies device driver handlers, we use interface-specific models to detect target Netlink families and their corresponding operation functions. For example, the operation `rtnl_setlink` (Fig. 1) is registered via `rtnl_register` [7] (registration details omitted). By analyzing all calls to `rtnl_register`, we can identify other related operations. To our knowledge, this work presents the first systematic summary of models for each Netlink interface. These models allow us to extract all operation functions for Netlink families in the kernel, serving as entrypoints for Netlink message analysis.

Parse Graph Internal and Construction. To analyze the Netlink message parsing process, we propose a dedicated graph model. This key insight is that the parsing process follows a principled approach, with most parsing actions handled by a limited set of library functions (defined in `lib/nlattr.c`, highlighted in red in Fig. 1). Hence, we can first model these limited library functions and use them

[7] The latest kernel uses a similar function named `rtnl_register_many`.

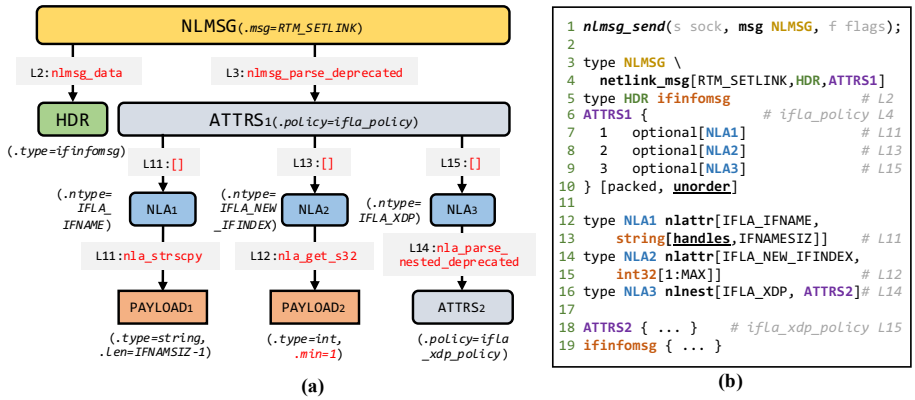

Fig. 2. Example parse graph for Fig. 1 (a) and the demo descriptions (b).

as waypoints to analyze the entire parsing process. Based on this insight, we propose parse graphs to model how Netlink messages are parsed. For example, Figure 2 (a) presents an example of the code listed in Fig. 1. In this graph, the nodes represent parsing elements in five concluded types: *NLMSG* for the entire message; *HDR* for the family-specific header; *ATTRS* for the stream of attributes; *NLA* for the sole attribute; and *PAYLOAD* for the payload carried in the attribute. Nodes with these types are highlighted with different colors in Fig. 2 (a). The edges represent associated parsing actions, such as the calls to library functions (L2,3,11,12,14) and array expressions of attributes array (L11,13,15). Furthermore, as demonstrated in Fig. 2(a), each node contains additional information. For example, the NLMSG node contains the message type (`.msg`), which determines the handler function (in this case, `rtnl_setlink`). The HDR and PAYLOAD nodes contain type information (`.type`) inferred from related expressions—for example, the ifinfomsg* type inferred from the return value of `nlmsg_data`. Some types may also have additional constraints, such as length (`.len`) for string type in PAYLOAD1. The ATTRS node contains the parsing policy (`.policy`), while the NLA node includes the attribute type (`.ntype`). In summary, the graph node contains key information and constraints necessary for recovering the message structure and generating accurate descriptions.

To construct the parse graph, we start by identifying all parsing actions and collecting relevant information through an iteration over all basic blocks. Each identified action defines a parsing element (e.g., the call to `nlmsg_parse_deprecated` (L3) defines ATTRS1), which is marked as a taint source. Using static taint analysis, we trace how this tainted element is used by other actions (e.g., ATTRS1) is used in array expressions (L11,13,15). These tainted elements are then connected to form parse paths, and the collection of all such paths constitutes the complete parse graph (see Table 3 for more details). When the taint is propagated to an indirect call site, apply the MLTA algorithm [26] to determine the possible callee candidates. In addition, our taint analysis also traces all comparisons involving the extracted payloads. This information can be used to refine the payload types. For instance, if the analysis reveals that a payload is involved in an enum-based switch statement, we extract the corresponding enum

values and include them in the fuzzing dictionary (represented as *flags* in the Syzlang syntax) when specifying this payload during description generation.

Updating with Netlink Policy. Since our taint analysis does not traverse pre-modeled library functions (e.g., `nlmsg_parse_deprecated`), the resulting graphs lack internal details and are therefore incomplete. To address this, we refine the parse graph using Netlink policies, which reflect the side effects of these library functions and provide more detailed constraints for attribute payloads. Specifically, we first dump all Netlink policies from the kernel and convert them into graphs with a structure compatible with the parse graph. Each policy graph has a parent node of type ATTRS and child nodes of type NLA, containing all defined constraints. If a policy defines nested constraints (e.g., using `NLA_POLICY_NESTED`) for an attribute NLAnested, we first identify the associated nested policy, convert it into another policy graph, and then attach it as a child subgraph of node NLAnested. This process is performed recursively in a depth-first manner, terminating upon encountering a repeated policy to prevent infinite recursion. After preparing all policy graphs, we update each ATTRS node in the parse graph by: (1) Locating the corresponding policy graph based on `.policy`; (2) Matching child NLA nodes between the parse graph and located policy graph based on `.ntype`; (3) Merging information and constraints from the grandchild PAYLOAD nodes. This ensures the parse graph incorporates all constraints from Netlink policies. For example, in Fig. 2 (a), the red-marked constraint (`.min = 1`) in PAYLOAD2 originates from the policy `ifla_policy`. It is worth noting that the update process also helps detect parsing errors. For example, we identify conflicts between the parsing code and the defined parsing policies (refer to Sect. 5.1 for further details).

Resolving Cross-Message Dependency. Once constraints are updated with Netlink policy, the parse graph should contain sufficient information to define valid messages for a *single operation*. To further improve fuzzing performance, we must also account for cross-message dependencies involving *multiple operations*. For example, when a userspace application configures or deletes a network link, the messages must reference an existing link created by prior messages. Unlike the open syscall, which returns a descriptor, such references are not explicitly returned by the kernel in syscall output. That is, cross-message dependency is a type of implicit dependency and the most accurate solution is cross-syscall input propagation analysis via techniques like symbolic execution [23].

However, porting symbolic execution to analyze Netlink messages in the Linux kernel is non-trivial. Instead, inspired by the existing descriptions, we adopt a heuristic approach that leverages our constructed parse graphs to resolve cross-message dependencies. The key insight is that, unlike in device driver targets, where dependency-related payloads are encapsulated in various structures, Netlink families always encapsulate these payloads in the same type of attribute across messages. For instance, the `IFLA_IFNAME` attribute appears in both link creation and link configuration/deletion messages and contains a payload string for identifying the network device. We refer to such attributes, whose payloads must remain consistent across messages, as Handle Attributes (HAs).

To identify HAs based on parse graphs, we investigated the existing descriptions and summarized below heuristic rules: (1) Use Case Rule: a HA-related node should exist in more than one parse graph. (2) Type Rule: a HA-related payload should be of a simple type, such as a string or integer, rather than a more complex structure. (3) Compare Rule: a HA-related payload should be compared with a non-constant value. The first and third rules are straightforward because we know that handle attributes are used in more than one operation, and the payload should be compared to check for existence. The second rule is empirically motivated based on the existing descriptions. To reduce false negatives, based on the found HAs, we do back-propagation from where their payloads are extracted to identify more HAs that serve the same purpose. After identification, we impose constraints to confine HAs' payload values to a limited set, thereby increasing the chance that payload hits the same value across messages during the fuzzing process. Note that potential false positives from our heuristic approach do not negatively impact fuzzing performance for two main reasons. First, we do not apply constraints to payloads that already have defined restrictions (such as value ranges or constant enums). As a result, our method avoids introducing conflicting constraints or reducing the randomness of such payloads. Second, regarding any unnecessary constraints, we observed that Syzkaller's SQUASH trait [19] will randomly convert constrained data structures into random blobs during fuzzing. This mechanism allows the fuzzer to bypass unnecessary constraints and continue exploring the input space effectively.

Description Generation. Finally, we traverse the parse graphs and translate each visited node to the Syzlang type. The node-to-type translation follows the schemas below (all line number references refer to the demo in Fig. 2 (b)):

- *NLMSG* nodes are translated into the Syzlang predefined struct `netlink_msg`, with message type and references to child nodes (e.g., the L3 `NLMSG` definition).
- *ATTRS* nodes have two translation methods, depending on how their represented elements are parsed. (1) For ATTRS nodes parsed via loops (e.g., `nlmsg_for_each_attr`), which allow repeated attributes, we adopt a definition consistent with existing descriptions. Specifically, we use Syzlang union and variable-length array to model the stream as a "permutation with repetition". (2) For ATTRS nodes parsed by library functions (e.g., `nlmsg_parse_deprecated`) that reject repeated attributes, we introduce a new type **unordered struct**, with Syzlang optional type to define the stream as a "permutation without repetition". For example, see the definition of `ATTRS1` at L6. This method can save the fuzzing effort wasted on generating and mutating redundant attributes, thereby improving overall efficiency.
- *NLA* nodes are translated into Syzlang's predefined structs (`nlattr` and `nlnest`). For example, the definitions of `NLA1,2,3` at L12,14,16.
- *PAYLOAD* nodes are translated based on their detailed type: For C structs, we use Syzkaller's built-in tool `syz-headerparser` to extract the struct definitions (e.g., the `ifinfomsg` definition at L19). For C scalars (e.g., integers), we use the corresponding Syzlang type (e.g., `int`) and leverage features like min, max, and flags to define constraints (e.g., the NLA2 payload definition

at L15). For C strings, we use Syzlang's `string` or `stringnoz` types. Other Syzlang types (e.g., `const`, `void`) are applied as appropriate. Additionally, for identified handle attributes, we use `handles` for string-type handles (e.g., the payload of NLA1 at L13), and `handlei` for integer-type handles. These special types use Syzlang's `proc` type, allocating 16 unique slots per fuzzing process.

After translating all nodes, we wrap the root type with a pseudo-syscall `nlmsg_send` (e.g., L1 in Fig. 2 (b)) to finalize the description for the operation.

Implementation Details. The static analysis component of NLSaber is implemented using CodeQL v2.18.2 [14]. We selected CodeQL due to its efficiency (well-designed multi-process and caching) and complete support for taint analysis. It enables users to define sources and sinks easily to detect desired taint flows. The entire system consists of 7.4K lines of Python code, 5K lines of QL code, and 500 lines of Syzkaller patches (e.g., adding new pseudo-syscalls and types). NLSaber is the first system designed to enhance Netlink family fuzzing by generating descriptions. To promote the development of similar systems and address concerns of reproducibility, the source code is available at `https://github.com/TroySysSec/NLSaber`.

4 Evaluation

General Setup. We conducted all experiments on a machine equipped with 48 Intel(R) Xeon(R) CPU E5-2678 and 128 GB of RAM, running Ubuntu 20.04 LTS. The target Linux kernel version is 6.1.70 LTS (released in Feb 2024). The Syzkaller and related Syzlang descriptions version is based on commit 8d34fd8d3a26 (released in Feb 2025). Like previous kernel fuzzing research, we use the kernel configuration provided by Syzbot [17], which incorporates the best practices from Google.

Research Questions. This section addresses the following research questions:

- **RQ1:** Are generated descriptions more complete and accurate?
- **RQ2:** Can generated descriptions enhance fuzzing's code coverage?
- **RQ3:** Can generated descriptions help find new vulnerabilities?

4.1 Descriptions Generation (RQ1)

During the target family identification, we consider a family interesting if it contains at least one operation with a PAYLOAD node. These families are then selected as fuzzing targets. Using this criterion, NLSaber identified 76 target families and 865 operations across six Netlink interfaces by scanning the entire 6.1.70 kernel. These targets span over 2,700 files and more than 2,000 KLOC. In addition, the constructed parse graphs contain an average of 285 nodes, and the graph for *route CORE* and the *generic nl80211* family exceeds 3k nodes. Such complexity highlights the need for an automated tool like NLSaber, as manual auditing is insufficient for comprehensive analysis.

To evaluate the correctness of the generated descriptions, we adopted a methodology inspired by previous work [22], cross-validating our results against

manually written descriptions. Specifically, we begin by identifying all existing message definitions based on their types (e.g., `netlink_msg`), and then group them according to the belonging Netlink families. Next, we (reverse-)translate the existing descriptions into parse graphs, enabling the comprehensive comparison with our constructed graphs. We focused on four key aspects: (1) target families (#fam), (2) operation functions (#op), and (3) parse paths (#ppath), which represent how the payload is extracted, (3) handle attributes (#ha), about cross-message dependency. We developed scripts to identify the same contents like nodes, edges, and paths, as well as to highlight differences, between these graphs. Then, we manually verified the true positives and false cases. While the process was primarily manual, it was significantly supported by our static analysis tool, which reports precise code locations for all detected parsing actions (see comments in Fig. 2(b)). Overall, the validation process took approximately a person-month to complete across all target families. The validation results are summarized in Table 1.

Table 1. Descriptions comparison results.

interface	SYZ						SABER					
	#fam	#op		#ppath		#ha	#fam	#op		#ppath		#ha
		TP	FP	TP	FP			TP	FP	TP	FP	
NETLINK_CRYPTO	1	7	0	2	3	0	1	7	0	2	0	0
NETLINK_GENERIC	22	400	4	2,053	2,808	41	40	583	0	3,428	431	249
NETLINK_NETFILTER	8	65	0	2,390	4,441	51	8	67	0	3,530	12	86
NETLINK_RDMA	1	26	0	73	5	6	3	48	0	134	11	13
NETLINK_ROUTE	20	106	0	2,295	6,010	30	23	136	0	3,812	389	238
NETLINK_XFRM	1	22	2	99	563	1	1	24	0	102	0	1
#total	53	626	6	6,912	13,830	129	76	865	0	11,008	843	587

Overall, the generated descriptions are more comprehensive than the existing ones. They cover 43% more target families (76 vs. 53) and support 38% more operations (865 vs. 626). Among the 23 newly supported families, the oldest was introduced in 2008 (Linux 2.6) and the newest in 2022 (Linux 5.17). Of the 240 previously untested operations, 108 belong to these 23 families, while the remaining 132 are newly added operations in families already supported by Syzkaller. Our analysis shows that 92 of these 132 operations (70%) have been present in the kernel for over three years. During validation, we identified six false positives in the Syzkaller-defined operations. These were caused by missing callback functions in the kernel: two due to deprecated commands and four due to human errors. This underscores the incompleteness and obsolescence of the existing descriptions and demonstrates the value of our automated approach in improving the situation.

In terms of parse path (#ppath) comparison, the generated descriptions contain significantly more true parse paths (11,008 vs. 6,912) and fewer false ones (843 vs. 13,830), indicating that our method can better guide fuzzers in generating and mutating attributes, thereby improving code coverage. This advantage is evident not only in interfaces where NLSaber covers more target families, such as `NETLINK_GENERIC`, with 67% more paths (3,428 vs. 2,053), but also in `NETLINK`

`_NETFILTER` (3,530 vs. 2,390). The latter benefits from frequent updates and the addition of new attributes. Two additional observations are noteworthy. First, we identified **85 broken parse paths** in Syzkaller's descriptions, caused by incorrect assumptions about kernel parsing logic, leading to improperly defined structures. These errors have been reported to and confirmed by the Syzkaller team, underscoring the importance of automated approaches in reducing human error. Second, our generated descriptions include 843 false parse paths, all of which arise from challenging corner cases in static analysis. Some attribute types depend on dynamically determined variables, making static resolution infeasible. To preserve soundness, NLSaber includes all possible outcomes, introducing some redundancy. Additionally, certain families use shared callback functions with internal flags to parse different messages, causing our tool to merge multiple parse graphs together and resulting in false positives. Fortunately, such cases are rare, and our approach still achieves a high accuracy of 93%.

NLSaber identifies 587 handle attributes (#ha). To evaluate these, we conduct an inclusion test to determine whether these attributes extracted from Syzlang descriptions are present in the identified set. The test reveals that only five out of 129 cases are missed, indicating that our generated descriptions guide the fuzzer in resolving implicit dependencies at least as effectively as Syzkaller. For the remaining 463 identified handle attributes, they may include false positives, as our method relies on a simple heuristic rather than precise cross-syscall input propagation analysis [23]. Determining whether these attributes correspond to shared resources requires in-depth knowledge of the underlying implementation; thus, we defer more precise verification to future work.

Answer to RQ1: Descriptions generated by NLSaber are more complete (cover 43% more families and 38% more operations) and achieve higher accuracy in parse paths (93% vs. 33%).

4.2 Fuzzing Effectiveness: Coverage (RQ2)

Fuzzing Experiment Setup. In this section, we evaluate the effectiveness of NLSaber by comparing it with other kernel fuzzers. Alongside the original Syzkaller, we include two description-free fuzzers for a comprehensive evaluation. The first is **FuzzNG** [2], a kernel device driver fuzzer based on LibFuzzer [18]. Instead of relying on complex descriptions to define pointer types in driver handlers, FuzzNG instruments kernel I/O functions (e.g., `copy_from_user` (CFU) to intercept kernel-user space interactions. This enables the LibFuzzer engine to generate structured inputs when the kernel accesses user-provided data. To adapt FuzzNG for fuzzing Netlink families, we modified its agent to: (1) open the appropriate Netlink socket; (2) invoke the `send` syscall instead of driver-specific `read`, `write`, and `ioctl`; and (3) generate random messages upon CFU happens. The second fuzzer is **WEIZZ** [12], an AFL [16] variant that also operates without predefined descriptions. WEIZZ introduces a "surgical" stage that infers message structures by tagging input bytes. To enable Netlink fuzzing, we developed a harness that: (1) encodes WEIZZ inputs into Netlink messages; (2)

sends them to the kernel using `send` syscall; and (3) collects KCOV coverage and returns it to the fuzzer via shared memory. We used virtme [1] to run the WEIZZ fuzzer in a QEMU KVM environment, consistent with the other fuzzers.

Each Netlink family was tested in a dedicated fuzzing session, with duration based on the number of available operations (#op) : 24 hours for fewer than 10 operations, 48 hours for fewer than 20, 72 hours for fewer than 50, and 120 hours otherwise. Netlink families previously exploited in kCTF were always tested for 120 hours. In all sessions, the fuzzer was allocated 4 GB of memory and a single CPU core. Following prior work [22], we disabled crash reproduction to focus solely on coverage, and we deleted all seed programs in both Syzkaller and NLSaber to ensure a fair comparison.

Fuzzing Experiment Results. Each fuzzing session ran three times, and the final average coverage (#cov) is summarized in Table 4. Moreover, we calculate Vargha and Delaney's $\hat{A}_{12}$ statistics to estimate the effect size of coverage advantage. The results demonstrate that:

- *Fuzzing with descriptions significantly outperforms the description-free counterparts.* Specifically, Syzkaller and NLSaber achieve higher coverage across all target Netlink families than FuzzNG and WEIZZ (more than doubling the overall coverage). Our analysis of the fuzzing logs shows that the latter two fuzzers often fail early due to attribute checks, which require each attribute in the stream to have a valid type and length. This underscores the benefit of using descriptions to guide Netlink family fuzzing. Additionally, the significant coverage advantage of Syzkaller-based fuzzers is partly attributable to its advanced features, such as support for virtual netdevices, WiFi emulation, and packet injection, which enable exploration of more code paths.
- *WEIZZ outperforms FuzzNG on more targets.* Specifically, it achieved better coverage, with a large effect size ($\hat{A}_{12}(\mathrm{n}) \geq 0.71$) in 27 sessions (marked by underlines). Furthermore, it achieved a 9.5% higher overall coverage (90,120 vs. 82.297). This improvement is due to WEIZZ's ability to incrementally infer message structures during fuzzing, enabling the generation of more valid inputs and deeper code path exploration. However, this ability is not sufficient to fully recover the message structures, and the related corpus gets easily rejected when one attribute is mutated erroneously. Worth noting that FuzzNG achieved higher coverage with $\hat{A}_{12}(\mathrm{n}) \leq 0.29$ in 15 sessions. Our analysis indicates that this advantage stems from FuzzNG's optimized snapshot-fuzzing engine [2] and its instrumentation of kernel functions such as `strncmp` and `memcmp`, which requires kernel modifications not supported by WEIZZ or other fuzzers. These enhancements contribute to FuzzNG's better performance in some targets.
- *NLSaber outperforms Syzkaller and achieves the best coverage for most targets.* Specifically, it achieved better coverage, with a large effect size ($\hat{A}_{12}(\mathrm{d}) \geq 0.71$) in 37 sessions (also marked by underlines). Furthermore, it achieved a 9.1% higher overall coverage (163,670 vs. 150,026) than Syzkaller in both supported targets. The improvement increases to 40.8% (211,271 vs. 150,026) if including NLSaber-specific targets. In addition, our tool achieves the highest

average coverage in 43/50 sessions (highlighted in bold). Such advantages indicate the effectiveness of the generated descriptions in enhancing the Netlink family fuzzing. Worth noting that, although Syzkaller achieves higher coverage on some targets, there are no cases where it shows a large effect size ($\hat{A}_{12}(\mathrm{d}) \leq 0.29$). In five cases, it exhibits a medium effect size, and in two cases, a small effect size, indicating that NLSaber performs comparably. Taking the 802154 session as an example, Syzkaller achieves 3.9% higher coverage. Our analysis reveals that descriptions used by Syzkaller deliberately exclude two operations: `NL802154_CMD_DEL_INTERFACE` and `IEEE802154_DEL_IFACE`. These operations can remove fuzzer pre-configured devices, which could disrupt fuzzing. By omitting them, Syzkaller achieves better coverage compared to our approach, which retains these operations. Interestingly, our analysis also shows that the slightly lower coverage in these cases may result from NLSaber detecting more crashes. For instance, in the nbd family, our tool triggered a use-after-free vulnerability that Syzkaller missed.

Answer to RQ2: Descriptions generated by NLSaber improves the fuzzing with more than 9.1% greater code coverage compared to the original Syzkaller.

4.3 Fuzzing Effectiveness: Vulnerability (RQ3)

We used the generated descriptions to guide fuzzing for zero-day vulnerability detection. Specifically, we conducted fuzzing campaigns on interesting kernel versions, like the stable and associated candidate versions. So far, our tool has uncovered in total 19 previously unknown vulnerabilities, all of which were reported and confirmed, with 12 CVEs assigned.

Table 2. New vulnerabilities detected by enhanced fuzzing.

	description	lifespan	CVE	effect
1	deadlock in iwpm_hello_cb	6yr 3mo	-	DoS
2	global overflow of ksmbd_nl_policy	2yr 10mo	CVE-2024-26608	DoS
3	global overflow of loggers	8yr 4mo	CVE-2023-6040	Leak
4	heap overflow in xt_find_target	10yr 2mo		Exp
5	global overflow of rmnet_policy	5yr 10mo	CVE-2024-26597	DoS
6	global overflow of wwan_rtnl_policy	3yr 9mo	CVE-2024-50128	DoS
7	heap off-by-one in ieee80211_tx_control_port	2yr 7mo	CVE-2024-56663	Exp
8	information leak in fl_set_geneve_opt	6yr 8mo	CVE-2025-22055	Leak
9	information leak in ip_tun_parse_opts_geneve	5yr 5mo		
10	information leak in nft_tunnel_obj_geneve_init	5yr 2mo		
11	information leak in tunnel_key_copy_geneve_opt	7yr 3mo		
12	information leak in xfrm_address_filter	9yr 4mo	CVE-2023-39194	Leak
13	information leak in xfrm_update_ae_params	1yr 8mo	CVE-2023-3773	Leak
14	null-ptr-deref in xfrm_update_ae_params	12yr 5mo	CVE-2023-3772	DoS
15	stack overflow in nft_set_desc_concat_parse	2yr 4mo	CVE-2022-1972	Exp
16	type confusion in nft_tunnel_obj_geneve_init	5yr 2mo	CVE-2025-22056	Exp
17	type confusion in nft_tunnel_opts_dump			Leak
18	use-after-free in handshake_req_submit	-	-	Exp
19	use-after-free in nfc_genl_llc_get_params	10yr 2mo	CVE-2023-3863	Leak

Among the 19 vulnerabilities, 13 are found in code (either new families or new functions within existing families) not supported by existing descriptions (highlighted in gray in the table). All of them remained in the kernel (and could have been exploited in the wild) for over a year. This again underscores the importance of developing automated tools to keep testing new and updated targets. We also assessed the exploitability and impact of each vulnerability with our best efforts. As shown in the table: (1) *DoS* marks indicate that the corresponding vulnerability leads to Denial of Service in the kernel. For example, the first deadlock bug could be exploited to spray non-killable threads and cause the kernel to hang. (2) *Leak* marks indicate that the corresponding vulnerability could cause kernel information leakage. Take the 11th vulnerability as an example: the root cause of this one is an integer overflow that leads to type confusion. We exploited this to get a heap out-of-bounds read primitive that allows us to read the adjacent kmalloc-512 cache. Then, using heap spraying techniques with `tty_port` as the spray object, we filled the kmalloc-512 cache with controlled data and leaked a function pointer to bypass KASLR. (3) *Exp* marks are the worst, as they enable stronger memory corruption, such as out-of-bounds or arbitrary write primitives. To demonstrate this, we developed a complete exploit for the 16th vulnerability which gains a kernel space arbitrary code execution primitive and achieves the Local Privilege Escalation (LPE). Similar to the aforementioned heap out-of-bounds read one, we also use heap spray techniques on the kmalloc-512 cache to exploit this. Specifically, we choose the `nft_object` as the victim object and use heap overflow to hijack the code pointer, thereby achieving ROP by pivoting the stack to a controlled heap location. A demonstration of this exploit is also available in our released artifacts.

Answer to RQ3: Descriptions generated by NLSaber help detect new vulnerabilities. As evidence, our enhanced fuzzing uncovered 19 previously unknown vulnerabilities, with 12 CVEs assigned.

5 Discussion

5.1 Finding Parsing Errors via Static Testing

During parse graph construction, e.g., when updating with Netlink policies, we perform static testing to find potential parsing errors. Specifically, we verify the following rules during parsing: *(R1)*. All active Netlink attributes must have corresponding entries in the associated parsing policies. This error typically occurs when new attributes are introduced into the kernel but the corresponding policy is not updated. As a result, there are no constraints on the attribute, allowing attackers to craft malicious payloads for harmful purposes. *(R2)*. Attributes that are validated must also be used; otherwise, the validation is redundant, causing confusing code or even functionality problems. *(R3)*. Library functions must be used correctly. For example, function `nlmsg_parse_deprecated` requires the user to provide a destination array whose size should be equal to the value of `maxtype` + 1. In practice, we identified 35 issues: 25 violations of R1, 4 of

R2, and 6 of R3. These parsing errors complement fuzzing-based approaches, as they typically do not cause memory corruptions and thus remain undetectable by sanitizers such as KASAN. All findings were reported, confirmed by the kernel community, and fixed successfully.

5.2 Limitations

Inter-Attributes Relation. NLSaber does not account for the relations between attributes within a Netlink message, instead allowing the fuzzer to explore attribute combinations randomly (e.g., using optional type or union type). However, certain attributes are mutually exclusive, while others must appear together. By explicitly modeling these constraints and guiding the fuzzer to generate or mutate inputs accordingly, the fuzzing accuracy can be improved, and fuzzing cycles wasted on wrong relations can be saved. Unfortunately, such relationships are difficult to analyze automatically so they are left as future work.
Other interfaces. NLSaber targets operation functions that handle userspace messages, as these are observed to be the most vulnerable. However, certain complex Netlink families involve additional interfaces, making operation-level fuzzing insufficient to uncover all bugs. For example, CVE-2022-10155 [9] is a complex vulnerability that is triggered during packet processing by kernel background threads, following a misconfiguration caused by malformed messages in operation functions. Identifying all relevant interfaces and performing comprehensive fuzzing is non-trivial. Therefore, we currently focus on individual operations and leave multi-interface scenarios for future work.
Other input structures. NLSaber focuses on Netlink families, so the proposed analysis is primarily designed for the Netlink side. This naturally limits the scope of our tool. Nevertheless, we believe that modeling the parsing process using parse graphs is also effective in other scenarios involving complex input structures, which we leave as future work.

6 Related Work

Structure-aware Fuzzing. Structure-aware fuzzing uses fuzzers that understand input formats [5,38], allowing them to generate and mutate inputs more effectively, which significantly improves performance [5]. To achieve structure-aware fuzzing, some fuzzers rely on predefined input descriptions. For userspace programs, tools such as AFLSmart [31] and Peach [11] allow users to define complex input formats and protocols using grammar-based representations. For kernel cases, Syzkaller [35] uses syscall descriptions and demonstrates its effectiveness with thousands of reported bugs. Several previous studies [33,7,22,21] aim to automate the generation of these descriptions for Syzkaller. For example, KSG [33] uses dynamic probe analysis targeting device drivers and network protocols. SyzDescribe [22] offers a systematic approach to generating syscall descriptions for Linux kernel drivers. SyzGen++ [6] uses symbolic execution for both Linux device drivers and closed-source macOS IOKit modules. However,

none of these approaches is effective in fuzzing Netlink families because they cannot handle complex Netlink messages. For example, we tried using SyzGen++'s type analysis to infer the message structure in Fig. 1, but the symbolic execution failed because it either timed out or ran out of memory.

Some other fuzzers do not rely on pre-existing descriptions. Instead, they infer input structures dynamically using runtime feedback. For instance, WEIZZ [12] infers message fields by analyzing dependencies between input bytes and comparison instructions. Polyglot [3], NestFuzz [10], and AIFORE [32] apply dynamic taint analysis to extract message structure. However, these approaches are difficult to adapt to Netlink families, as they typically rely on frameworks like DFSan [14], which are not supported in the upstream Linux kernel. In our evaluation, we ported the WEIZZ fuzzer since the Linux kernel's KCOV feature supports collecting the comparison coverage it requires. Additionally, we ported FuzzNG [2] as it also operates without relying on syscall descriptions.

Improving Kernel Fuzzing via Other Ways Rather than focusing on description generation, many existing approaches enhance kernel fuzzing via other solutions. Moonshine [30] derives and refines seed test cases from dynamic execution traces to improve code coverage. SyzVegas [36] applies reinforcement learning to optimize the mutation strategy. Healer [34] proposes a relation learning-based solution to improve the choice table algorithm of Syzkaller. StateFuzz [38] leverages state coverage to guide fuzzing toward deeper program paths. Actor [13] uses action-guided synthesis with specialized templates to help fuzzers discover more bugs. Note that most of these studies require ready-made syscall descriptions, so our work is orthogonal to theirs. That is, the descriptions we generate can serve as a complement to these tools when fuzzing Netlink families.

7 Conclusion

In this paper, we present NLSaber, the first specialized tool that automatically generates complete and accurate descriptions for fuzzing Netlink families. NLSaber employs static taint analysis to construct parse graphs to model the Netlink message parsing process. These graphs capture critical parsing elements, associated actions, and other relevant details, enabling the generation of high-quality descriptions for fuzzing. We evaluated NLSaber on Linux 6.1.70, demonstrating that its generated descriptions are more complete and accurate than existing ones. Using these generated descriptions, our enhanced fuzzing results in over 9.1% improved code coverage. Additionally, it uncovered 19 previously unknown vulnerabilities, with 12 CVEs assigned.

Acknowledgement. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant U21A20464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies. We thank all anonymous reviewers for their invaluable comments.

Appendix Tables

Pre-Modeled Parsing Actions The complete list of pre-modeled parsing actions is provided in Table 3. In each entry, capitalized keywords denote taint elements or essential information values. `XXXsrc` (`XXX` could be `HDR`, `NLMSG`, `ATTRS`, `NLA`, and `PAYLOAD`) indicates a new taint source introduced by the action, while `XXXdst` represents a potential taint sink in some expected taint flows. During the taint analysis, we mark all identified `XXXsrc` as the taint source and track where the taint element flows to. If the flow reaches any of the `XXXdst`, we wrap a graph node of `XXX` type and connect it according to the tainted action. Other keywords are about the to-collect information, for example, the `POLICY` keyword assists the analysis in identifying the corresponding Netlink policy, which provides details such as attribute constraints. `NTYPE` helps determine the attribute type critical for reconstructing the message structure. `MAXTYPE` enables further verification, as discussed in Sect. 5.1. Parsing actions 42–48 are special cases that require additional explanation. In these actions, the inner parsing *is not nested;* instead, it occurs at the same "parsing level". In other words, these actions operate directly on the attribute itself, rather than on its payload. When handling these cases, we use sibling nodes rather than child nodes in the parse graph to ensure a correct definition of the message structure.

Table 3. List of pre-modeled parsing actions and to-collect information.

	Parsing Actions
1	`HDRsrc=nlmsg_data(NLMSGdst);`
2–4	`nlmsg_parse{_deprecated}{_strict}(NLMSGdst, ∼, ATTRSsrc, MAXTYPE, POLICY, ∼);`
5	`nlsmg_validate_deprecated(NLMSGdst, ∼, MAXTYPE, POLICY, ∼);`
6	`NLAsrc = nlmsg_attrdata(NLMSGdst);`
7	`NLAsrc = nlmsg_find_attr(NLMSGdst, ∼, NTYPE);`
8	`NLAsrc = ATTRSdst[NTYPE];`
9–24	`PAYLOADsrc = nla_get_{u/s/be/le}{8/16/32/64}(NLAdst);`
25	`PAYLOADsrc = nla_data(NLAdst);`
26–30	`PAYLOADsrc = nla_get_{flag/msecs/in_addr/in6_addr/bitfield32}(NLAdst);`
31–32	`nla_{strscpy/memcpy}(PAYLOADsrc, NLAdst, SIZE);`
33	`PAYLOADsrc = nla_strdup(NLAdst, ∼);`
34–35	`nla_{strcmp/memcmp}(NLAdst, ∼, ∼);`
36	`NLAsrc = nla_data(NLAdst);`
37	`NLAsrc = nla_find_nested(NLAdst, NTYPE);`
38–39	`nla_parse_nested{_deprecated}(ATTRSsrc, MAXTYPE, NLAdst, POLICY, ∼);`
40–41	`nla_validate_nested{_deprecated}(NLAdst, MAXTYPE, POLICY, ∼);`
42	`NLAsrc* = nla_next(NLAdst*);`
43	`NLAsrc* = nla_find(NLAdst*, ∼, NTYPE);`
44–46	`nla_parse{_deprecated}{_strict}(ATTRSsrc*, MAXTYPE, NLAdst*, ∼, POLICY, ∼);`
47–48	`nla_validate{_deprecated}(NLAdst*, ∼, MAXTYPE, POLICY, ∼);`

Fuzzing Coverage Experiment Results Due to space limits, we present the fuzzing coverage results for Sec. 4.2 in Table. 4.

Table 4. Fuzzing coverage results.

interface	family	FuzzNG	WEIZZ	$\hat{A}_{12}$(n)	Syzkaller		NLSaber		$\hat{A}_{12}$(d)
		#cov	#cov		#op	#cov	#op	#cov	
NETLINK_CRYPTO	-	1,523	1,363	0.0	7	1,819	7	**1,862**	0.89
NETLINK_GENERIC	802154*	2,257	2,845	1.0	55	**7,989**	57	7,690	0.33
	batadv	1,371	1,403	1.0	19	3,508	19	**3,733**	0.89
	devlink	1,348	1,357	0.56	35	5,970	73	**7,430**	1.0
	ethtool	2,875	1,737	0.0	39	4,121	58	**4,225**	1.0
	fou	1,958	2,148	1.0	4	4,067	4	**4,101**	0.78
	gtp	1,299	1,342	1.0	5	**3,274**	5	3,261	0.44
	hsr	1,230	1,217	0.22	/		2	**2,938**	1.0
	ila	1,647	1,479	0.0	/		5	**3,111**	1.0
	ioam6	1,478	1,373	0.0	/		7	**2,977**	1.0
	ipvs	1,373	1,384	0.67	17	4,337	17	**4,444**	0.78
	l2tp	1,373	2,279	1.0	11	4,141	11	**4,367**	1.0
	macsec	1,281	1,357	1.0	/		11	**3,388**	1.0
	mptcp	1,373	1,404	0.89	12	2,194	12	**2,953**	1.0
	nbd	2,959	2,973	0.67	4	**5,008**	4	4,945	0.33
	ncsi	1,322	1,282	0.0	/		7	**3,089**	1.0
	net_dm	1,102	1,499	1.0	2	2,729	5	**3,070**	1.0
	netlabels*	1,468	1,794	1.0	24	3,728	24	**3,734**	0.67
	nfc	1,547	2,999	1.0	19	5,075	20	**5,113**	0.89
	nl80211	4,316	5,108	1.0	114	12,441	122	**14,497**	1.0
	openvswitchs*	1,686	2,287	1.0	/		23	**6,204**	1.0
	seg6	1,261	1,371	1.0	4	2,971	4	**2,978**	0.44
	smbd	1,227	1,424	1.0	/		16	**2,932**	1.0
	smcs*	1,384	1,349	0.11	5	2,955	23	**3,076**	1.0
	taskstats	1,417	1,378	0.11	/		1	**2,980**	1.0
	tcm_user	1,197	1,229	1.0	/		4	**2,828**	1.0
	tcp_metrics	1,337	1,260	0.11	/		3	**2,963**	1.0
	team	1,277	1,326	1.0	4	**3,089**	4	3,084	0.33
	thermal	1,231	1,205	0.11	/		5	**2,785**	1.0
	tipcv2	1,436	1,529	0.78	27	5,540	28	**5,562**	0.67
	vdpa	1,329	1,267	0.22	/		9	**2,490**	1.0
NETLINK_NETFILTER	ACCT	1,301	1,332	0.78	4	**1,994**	4	1,969	0.33
	CONNTRACK*	1,509	1,596	0.78	20	2,613	20	**2,956**	1.0
	IPSET	1,343	1,398	0.78	15	4,927	16	**5,124**	0.67
	NFTABLES	1,399	1,428	0.56	23	3,147	23	**5,418**	1.0
	QUEUE	1,273	1,272	0.56	3	**1,958**	4	1,936	0.33
NETLINK_RDMA	IWCM	1,147	1,418	1.0	/		8	**2,431**	1.0
	LS	1,128	1,217	0.89	/		3	**2,352**	1.0
	NLDEV	1,402	1,397	0.44	26	5,175	37	**5,274**	0.67
NETLINK_ROUTE	CORE*	7,213	9,744	1.0	54	17,351	75	**19,147**	1.0
	DCB	1,274	1,245	0.33	/		2	**2,056**	1.0
	IPV6SPEC*	1,392	1,419	1.0	8	2,260	8	**2,312**	1.0
	MDB	1,287	1,237	0.11	3	2,296	3	**2,364**	0.89
	NEXTHOP*	1,343	1,374	0.78	8	3,662	10	**3,962**	1.0
	NSID	1,330	1,321	0.22	3	**2,796**	3	2,789	0.44
	RULE	1,898	2,020	1.0	7	3,928	9	**3,935**	0.44
	SCHED*	1,804	1,818	0.67	20	5,417	20	**7,831**	1.0
	TUNNEL	1,292	1,241	0.22	/		3	**2,077**	1.0
	VLAN	1,247	1,194	0.0	3	3,010	3	**3,472**	1.0
NETLINK_XFRM	-	1,833	2,481	1.0	22	4,536	24	**5,056**	1.0
#total common	-	61,074	68,241	-	626	150,026	756	**163,670**	-
#total all	-	82,297	90,120				865	**211,271**	

* means there are multiple families grouped together for better fuzzing;

/ means the family is not supported.

References

1. amluto: An easy way to virtualize the running system, https://github.com/amluto/virtme
2. Bulekov, A., Das, B., Hajnoczi, S., Egele, M.: No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In: Network and Distributed System Security (NDSS) Symposium (2023)
3. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 317–329 (2007)
4. Chao Ma, Han Yan, T.X.: Linkdoor: A hidden attack surface in the android netlink kernel modules
5. Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., Liu, W.: A systematic review of fuzzing techniques. Computers & Security **75**, 118–137 (2018)
6. Chen, W., Hao, Y., Zhang, Z., Zou, X., Kirat, D., Mishra, S., Schales, D., Jang, J., Qian, Z.: Syzgen++: Dependency inference for augmenting kernel driver fuzzing. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 4661–4677. IEEE (2024)
7. Chen, W., Wang, Y., Zhang, Z., Qian, Z.: Syzgen: Automated generation of syscall specification of closed-source macos drivers. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 749–763 (2021)
8. Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G.: Difuze: Interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2123–2138 (2017)
9. David: How the tables have turned: An analysis of two new linux vulnerabilities in nf_tables (2022)
10. Deng, P., Yang, Z., Zhang, L., Yang, G., Hong, W., Zhang, Y., Yang, M.: Nestfuzz: Enhancing fuzzing with comprehensive understanding of input processing logic. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1272–1286 (2023)
11. Eddington, M.: Smartfuzzer peach. (2021), https://peachtech.gitlab.io/peach-fuzzer-community/
12. Fioraldi, A., D'Elia, D.C., Coppa, E.: Weizz: Automatic grey-box fuzzing for structured binary formats. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis. pp. 1–13 (2020)
13. Fleischer, M., Das, D., Bose, P., Bai, W., Lu, K., Payer, M., Kruegel, C., Vigna, G.: {ACTOR}:{Action-Guided} kernel fuzzing. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 5003–5020 (2023)
14. Github: Codeql: the libraries and queries (2021), https://github.com/github/codeql
15. Google: Public kctf vrp kernelctf responses, https://github.com/google/security-research/tree/master/pocs/linux/kernelctf
16. Google: american fuzzy lop - a security-oriented fuzzer (2024), https://github.com/google/AFL/tree/master
17. Google: stable-6.1-kasan.config (2024), https://github.com/google/syzkaller/blob/master/dashboard/config/linux/stable-6.1-kasan.config
18. Google: Structure-aware fuzzing with libfuzzer. (2025), https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md

19. Google: Syzkaller squashany (2025), https://github.com/google/syzkaller/blob/master/prog/mutation.go#L138
20. Group, N.W.: Rfc3549, linux netlink as an ip services protocol (2003)
21. Han, H., Cha, S.K.: Imf: Inferred model-based fuzzer. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2345–2358 (2017)
22. Hao, Y., Li, G., Zou, X., Chen, W., Zhu, S., Qian, Z., Sani, A.A.: Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 3262–3278. IEEE Computer Society (2023)
23. Hao, Y., Zhang, H., Li, G., Du, X., Qian, Z., Sani, A.A.: Demystifying the dependency challenge in kernel fuzzing. In: Proceedings of the 44th International Conference on Software Engineering. pp. 659–671 (2022)
24. Lever, C.: Another crack at a handshake upcall mechanism, https://lwn.net/Articles/922553/
25. Linux: Introduction to netlink, https://docs.kernel.org/userspace-api/netlink/intro.html
26. Lu, K., Hu, H.: Where does it go? refining indirect-call targets with multi-layer type analysis. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 1867–1881 (2019)
27. melifaro: netlink: add netlink support. https://reviews.freebsd.org/D36002?id=109872 (2022)
28. Mongodin, A.: Yet another bug into netfilter (2022), https://www.randorisec.fr/yet-another-bug-netfilter
29. Nguyen, A.: Turning \x00\x00 into 10000$ (2021), https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html
30. Pailoor, S., Aday, A., Jana, S.: Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 729–743 (2018)
31. Pham, V.T., Böhme, M., Santosa, A.E., Căciulescu, A.R., Roychoudhury, A.: Smart greybox fuzzing. IEEE Transactions on Software Engineering **47**(9), 1980–1997 (2019)
32. Shi, J., Wang, Z., Feng, Z., Lan, Y., Qin, S., You, W., Zou, W., Payer, M., Zhang, C.: {AIFORE}: Smart fuzzing based on automatic input format reverse engineering. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4967–4984 (2023)
33. Sun, H., Shen, Y., Liu, J., Xu, Y., Jiang, Y.: {KSG}: Augmenting kernel fuzzing with system call specification generation. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22). pp. 351–366 (2022)
34. Sun, H., Shen, Y., Wang, C., Liu, J., Jiang, Y., Chen, T., Cui, A.: Healer: Relation learning guided kernel fuzzing. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 344–358 (2021)
35. Vyukov, D.: Syzkaller: an unsupervised, coverage-guided kernel fuzzer (2019)
36. Wang, D., Zhang, Z., Zhang, H., Qian, Z., Krishnamurthy, S.V., Abu-Ghazaleh, N.: {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2741–2758 (2021)
37. wikipedia: Netlink. https://en.wikipedia.org/wiki/Netlink (2023), last edited: 2023-4-21
38. Zhao, B., Li, Z., Qin, S., Ma, Z., Yuan, M., Zhu, W., Tian, Z., Zhang, C.: {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3273–3289 (2022)