Fuzz Testing on IPC Based OS Services with Resource State Awareness

Yuheng Shen
Tsinghua University
Beijing, China
syh1308@gmail.com

Yifei Chu
Tsinghua University
Beijing, China
chuyf24@mails.tsinghua.edu.cn

Fan Ding
Central South University
Changsha, China
strayingfar@outlook.com

Qiang Zhang*
Jiangxi University of Finance and Economics
Nanchang, China
zhangqiang9413@126.com

Jianzhong Liu
Tsinghua University
Beijing, China
j.zhong0@gmail.com

Heyuan Shi*
Central South University
Changsha, China
hey.shi@foxmail.com

Yu Jiang*
Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

Modern operating systems increasingly implement core functionality in IPC based OS services. Existing syscall and IPC fuzzers model producer-consumer dependencies to generate structured inputs and reach service handlers. However, handler reachability is insufficient when resources persist across requests, clients, and threads. Without tracking accepted resources, fuzzers may miss bugs requiring multiple requests. In this paper, we present RAIF, a fuzzer for IPC based OS services. RAIF organizes fuzzing around service-side resource dependencies rather than isolated IPC messages. It first generates workloads that connect resource creation, operations, and update APIs around the same service resources. It then dispatches test cases across clients, so related resources are exposed to the service in a compact request order. Finally, RAIF collects resource dependency information after IPC unmarshalling and uses it to guide subsequent fuzzing. Evaluation shows that RAIF identified 23 bugs on OpenHarmony, Android, and D-Bus. On the three Android services, it improved branch coverage by 25.8% and triggered 3.5× more resource state transitions than baselines.

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

Keywords: Operating System Security, Fuzz Testing, Bug Detection

ACM Reference Format:
Yuheng Shen, Yifei Chu, Fan Ding, Qiang Zhang, Jianzhong Liu, Heyuan Shi, and Yu Jiang. 2026. Fuzz Testing on IPC Based OS Services with Resource State Awareness. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3828523.3857101

*Corresponding authors.

SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857101

1 Introduction

Modern operating systems increasingly move core functionality out of the kernel and into long-running system services that are accessed through inter-process communication (IPC). Applications usually interact with these services by sending IPC requests rather than invoking all functionality directly through traditional system calls. Graphics, input, and media subsystems are commonly implemented in this way. For example, OpenHarmony's RenderService module and Android's SurfaceFlinger module provide graphics and display functionality through IPC interfaces. Bugs in these services can crash privileged service processes, causing denial-of-service to the entire system. Therefore, testing IPC based OS services is critical for OS robustness.

Background. Fuzz testing (fuzzing) generates inputs to trigger abnormal behavior such as memory corruption and system hang. Fuzzing has successfully found bugs in operating system kernels [4, 24, 27, 29, 30] and libraries [6, 18]. Kernel fuzzers such as Syzkaller [32] use system call descriptions [33] to generate structured system call sequences and use coverage to evolve the corpus. Current fuzzers model dependencies among kernel resources, like file descriptors, so that later syscalls can operate on objects created by earlier syscalls. Prior IPC fuzzers generate valid IPC requests and reach service handlers by modeling message schemas and argument constraints. For example, BinderCracker [9] records kernel IPC transactions, reconstructs transaction schemas, and replays dependent transactions. FANS [19] and NASS [21] recover Android service interface models and dependencies, while MachFuzzer [35] infers message format and ordering constraints for macOS Mach IPC services. These techniques help requests reach service handlers.

Limitations. Syscall fuzzers commonly adopt resource aware techniques to model producer–consumer dependencies or maintain valid resource inputs, while IPC and protocol fuzzers model message formats, request ordering, and protocol states. These dependencies help generate valid inputs and reach service handlers. For stateful IPC services,

however, handler reachability is not enough. A request may exercise only shallow validation logic if it is not connected to the resource state maintained inside the service. Such state is usually established, updated, and submitted across IPC requests under a client context. Therefore, the important dependency is not only whether requests are connected in the input, but whether they are connected through service side resource state during execution. Input dependency models do not directly capture this service level relation.

Even with well-defined API signatures, testing resource dependencies requires more than generating well-formed IPC requests. The test case must carry service resource state across the requests that create, update, and submit it under the proper client context. Thus, the effective testing unit is a resource based client transaction rather than a single IPC request. This raises three requirements for a fuzzer. First, it must generate transaction based workloads that preserve resource dependencies across requests and clients. Second, it must dispatch transactions based on resource dependencies, so that related resource updates are exposed to the service in nearby execution windows. Third, it must use runtime information collected after IPC unmarshalling to preserve resource dependencies that are actually exercised by the service and guide later mutation toward deeper service logic. These requirements lead to three challenges.

Challenges. First, resource based IPC workloads are hard to generate because IPC interfaces do not expose resource roles. A syntactically valid IPC request is often not a semantically effective service operation. The interface exposes callable methods and argument types, but it does not tell the fuzzer which earlier requests should create a required resource, which later requests should reuse it, or which submit API should make the update visible. For example, a layer update becomes meaningful only when the workload has created the layer and the current client can still reference it. If the fuzzer randomly combines valid requests, later requests may refer to missing objects, so the service rejects them early or executes only shallow error handling logic.

Second, related client transactions are hard to align during execution. Simply creating multiple clients or threads is not enough to exercise cross-client service behavior. For many IPC based OS services, the effective operation unit is not an isolated request, but a client transaction, including a group of requests that updates related resources and ends at an update API such as commit or flush. Two clients may both issue valid transactions, but they will not interact if they operate on unrelated resources or if their update points are far apart in the service execution timeline. Therefore, to expose cross-client behavior, the fuzzer must make transaction blocks from different clients operate on the same or related service resources and dispatch these blocks around submit APIs.

Third, useful resource dependencies are hard to preserve during mutation. The fuzzer must distinguish the resource identifiers it intended to use from the resources that the service actually accepted. The input records what the fuzzer intended to send before IPC unmarshalling, but not what the service actually executed. It does not reveal whether a resource identifier was accepted, whether two transaction blocks touched the same service resource, or whether an update crossed a submit boundary. Without this information, mutation may destroy useful identifiers. Therefore, feedback must be collected after unmarshalling to preserve and extend resource dependencies observed during execution.

Solutions. In this paper, we propose RAIF, an IPC service fuzzer that organizes fuzzing around resource-dependent client transactions. RAIF first uses an API list to identify which service APIs create resources, operate on existing resources, and submit pending updates. Based on this information, it builds client transactions that keep related operations on the same service resource. It then dispatches transactions from different clients according to their resource dependencies, so related updates are exposed to the service in nearby request windows. Finally, RAIF instruments resource operations after IPC unmarshalling and uses this feedback to preserve useful resource dependencies during mutation.

Evaluation. We evaluate RAIF on RenderService, SurfaceFlinger, InputFlinger, MediaCodec, and D-Bus. Results show that RAIF found 23 bugs in total, of which 14 OpenHarmony bugs have been fixed. On the three Android services, RAIF achieves an average of 25.8% branch coverage improvement and triggers 3.5× more resource state transitions than baseline fuzzers.

In summary, this paper makes the following contributions:

- We propose RAIF, a fuzzer for IPC based OS services that targets the service state exploration gap between reaching IPC handlers and exercising service logic.
- We present a fuzzing technique that uses service resource dependencies to guide workload generation, transaction dispatch, and mutation.
- We evaluate RAIF on representative IPC based OS services. RAIF detects 23 bugs and improves coverage by 25.8% over baseline fuzzers.

2 Background

In this section, we introduce the concept of IPC based OS services and the workflow of kernel fuzzing, which provide the basis for our motivation and design.

2.1 IPC Based OS Services

An inter-process communication (IPC) based OS service is a system component that exposes OS functionality through IPC APIs. Applications send IPC requests to a service process, which validates the request, unmarshals its arguments, and dispatches it to the corresponding service handler. Modern OSes commonly implement display, input, camera, and media functionality in this way, such as OpenHarmony RenderService, Android SurfaceFlinger, and MediaCodec.

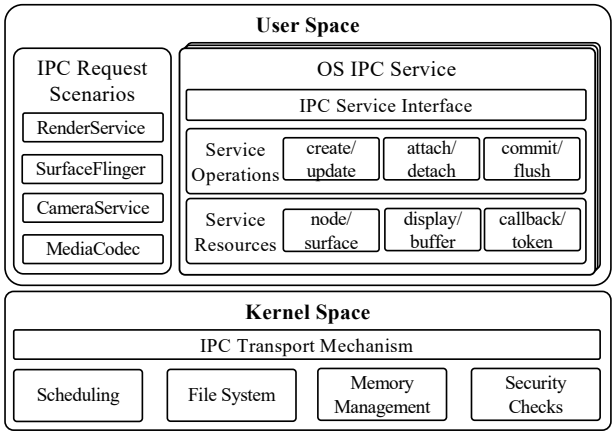


Figure 1. Illustration of the IPC based OS service model.

As shown in Figure 1, the IPC transport delivers requests between clients and services, while the service layer interprets these requests as service operations. These operations create, or update service resources that persist beyond a single IPC request. The service usually tracks these resources together with the client that owns or updates them. Many IPC services also batch updates rather than applying each request immediately. A client may first accumulate changes in local transaction state, and the service applies these changes to the global service state only when an update API such as commit or flush is executed. This creates resource dependencies across IPC requests, where a later operation may depend on a resource created or updated by earlier operations under the same client context. Individual request formats do not fully capture these runtime resource dependencies.

2.2 Operating System Fuzzing

Fuzz testing, or fuzzing [26, 28], generates malformed inputs to expose abnormal behaviors such as crashes and system hangs. Operating system fuzzers test OS components by generating inputs that cross system interfaces, executing them on the target system, and using runtime feedback to evolve later inputs. For kernel fuzzing, tools such as Syzkaller [32] generate syscall programs from syscall descriptions [33]. For IPC service fuzzing, tools instead generate IPC messages that are marshalled into concrete service requests.

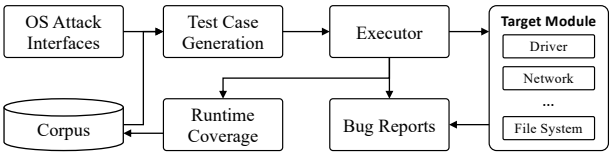


Figure 2. OS fuzzing overview.

The overall workflow is illustrated in Figure 2. A fuzzer starts from a model of OS interfaces, such as syscall descriptions or IPC interface specifications. These models specify callable operations, argument types, resource types, and basic constraints. The fuzzer then generates test cases, executes them on target modules, and monitors crashes. Coverage feedback is commonly used to evolve the corpus. Inputs that reach new code are kept and mutated in later iterations.

Resource modeling is important for generating valid OS inputs. In syscall fuzzing, later syscalls often depend on resources produced by earlier syscalls, such as file descriptors or device handles. For example, a test case may open a device file and pass the returned file descriptor to later `ioctl` or `read` calls. This producer-consumer model helps the fuzzer construct valid syscall sequences. IPC service fuzzers also model message formats and interface level dependencies, requests from different clients may need to operate on the same service resource, such as a layer or display.

3 Motivation

In this section, we use a reported SurfaceFlinger heap use after free bug [10] to illustrate why reaching an IPC handler is not sufficient for testing IPC based OS services.

As shown in Figure 3, SurfaceFlinger maintains an internal resource set `mOffscreenLayers`, which stores offscreen layer objects (middle column). During the commit path (right column), the SurfaceFlinger main thread executes `commitOffscreenLayers()` and traverses this set to apply pending layer transactions. Meanwhile, a binder thread (left column) processes a layer destruction request and frees a layer that is referenced by `mOffscreenLayers`. If the destroy path overlaps with the commit path, the main thread may dereference the freed `Layer*` and trigger a use after free.

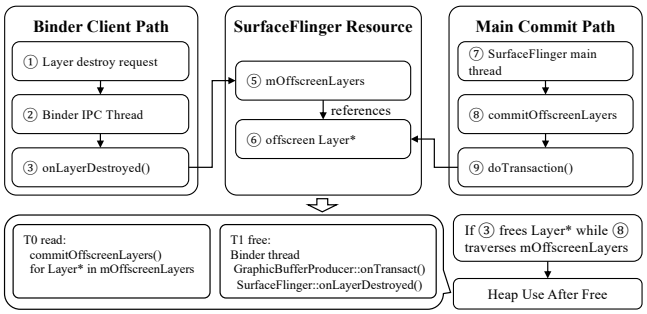


Figure 3. A use-after-free bug in Android's SurfaceFlinger.

To trigger this bug, a direct approach is to extend existing fuzzers with more IPC APIs and generate valid requests that reach the vulnerable code path. However, reaching the path is not enough; the bug requires the workload to create valid layer resources in `mOffscreenLayers`, make later update or destruction requests consistently operate on the same previously created layer, and submit these related operations near the commit path where SurfaceFlinger traverses the pending state of those layers. Otherwise, the same APIs may execute without exposing the bug.

This pattern is common in IPC services. Many IPC services organize behavior around resources that are created by one request, updated by later requests, and made visible at submit points. For example, SurfaceFlinger has 86 binder APIs, with 71 transaction APIs over layer and display resources. RenderService exposes 455 APIs, 342 of which are resource dependent. However, existing fuzzers improve message validity by modeling input dependencies or protocol states. These models alone do not ensure that related requests operate on resources actually accepted by the service. Testing these interactions also requires maintaining client contexts and dispatching related transactions around submit boundaries.

A preliminary study shows that, using the same IPC wrappers, targets, and permissions, a Syzkaller-style IPC baseline sends about 40K requests on SurfaceFlinger, but only about 10K requests are accepted and about 1K resource transitions are executed. Manual inspection shows that many requests lose resource dependencies because they refer to missing resources or generate updates that are never submitted. This suggests that reaching handlers alone is not enough; effective fuzzing must generate resource-based workloads, dispatch related transactions around submit APIs, and preserve resource dependencies observed during execution.

Challenge 1: Generating Resource Based IPC Workloads. Generating IPC requests that reach layer management handlers is not enough. The fuzzer must know which APIs create service resources, which APIs operate on existing resources, and which APIs submit pending updates. These roles are usually not explicit in IPC interfaces; they are implied by API names, argument usage, return values, and service side command wrappers. In the motivating bug, the workload must create layer resources and drive them into service side structures such as `mOffscreenLayers`; otherwise, later update or destruction requests may reach handlers but miss the resource state where the bug occurs.

Challenge 2: Dispatching Transactions Based on Resource Dependencies. Simply running multiple clients or threads does not guarantee meaningful service interaction. Different clients may operate on unrelated resources, or their requests may be applied too far apart in the service execution timeline. For IPC based OS services, the effective unit is often a client transaction, with a group of requests issued under one client context, operating on related resources, and ending at an update API. The challenge is to keep related transactions close without assuming control over the service's internal thread scheduling. In the SurfaceFlinger bug, the layer destruction path must be placed close to the commit path that traverses `mOffscreenLayers`; otherwise, the same operations may execute without exposing the use after free.

Challenge 3: Using Resource Behavior to Guide the Fuzzing. The input records what the fuzzer intended to send, but not what the service actually executed. It does not show whether an identifier was resolved to a service resource, whether two transaction blocks touched the same object, or which resource dependencies should be preserved during mutation. For the SurfaceFlinger bug, the fuzzer needs to know whether the generated layer identifier became a live `Layer*`, entered `mOffscreenLayers`, and was later accessed by both the destruction and commit paths. This information is needed to distinguish resource dependencies from requests that only look related in the input. Without such feedback, later mutation may break useful identifiers or add operations around resources that were never exercised.

4 Design

To address the above challenges, we propose RAIF, a fuzzer for IPC based OS services. As shown in Figure 4, RAIF maintains service resource dependencies with three components. First, resource based workload generation constructs candidate resource dependencies around persistent service resources. Second, resource aware transaction dispatch interleaves related blocks across clients so the service observes their effects in close update cycles. Third, resource behavior guided mutation collects runtime resource dependency and leverages it to guide subsequent fuzzing.

4.1 Resource Based Workload Generation

Independently generated calls may reference missing or unrelated resources or leave updates unsubmitted. RAIF addresses this by generating same-resource transaction blocks that end at submit APIs. To cover resource dependent service logic, RAIF generates workloads around service resources rather than isolated requests. The key idea is to make each generated workload carry the resource context needed by later operations. Instead of selecting IPC APIs independently, RAIF first groups APIs by the service resources they manipulate, then generates short transaction blocks that create or reuse these resources, bind later operations to existing resource identifiers, and end at update APIs when available. Finally, RAIF assembles these blocks into multi client workloads, where each client sequence preserves its own context.

Classifying Resource Operations. Before generating resource dependent workloads, RAIF identifies how IPC APIs manipulate service resources. However, IPC interfaces usually expose only callable methods and argument types. They do not explicitly specify which APIs create resources, which APIs operate on existing resources, which arguments carry resource identifiers, or which APIs submit pending updates. Therefore, RAIF builds an IPC API list that records the resource role of each supported API. The API list only provides lightweight role annotations; service side feedback later determines which generated resources are actually executed.

To draft the IPC API list, we give the LLM the service's interface header files and the corresponding developer documentation. The prompt specifies the expected pseudo-call structure and argument template, and the model generates pseudo-call signatures and arguments. We then manually

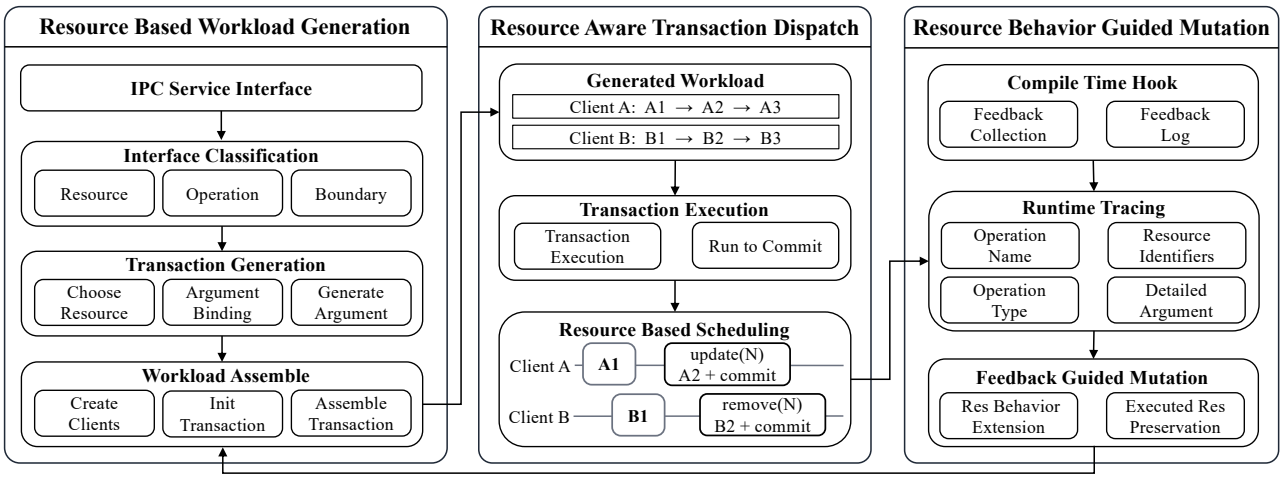


Figure 4. Overall workflow of RAIF.

validate the generated entries. Each entry records the API name, its role, the resource kind, and all resource-bearing argument positions. RAIF uses three roles: `create`, `operate`, and `submit`. A `create` API introduces a resource that later requests can use; an `operate` API consumes or modifies an existing resource; and a `submit` API makes accumulated changes visible to the service. With this list, the generator can choose APIs according to available resources, bind later operations to existing resource identifiers, and group related requests into client transactions.

Take RenderService as an example. As shown in Figure 5, `CreateDisplayNode()` is classified as a `create` API because it creates a display node and returns an identifier that later requests can use. `SetBounds()` is classified as an `operate` API because it updates an existing surface render node through the `nodeId` argument. Some APIs carry multiple resource arguments. For example, `RSBaseNodeAddChild` consumes the `parent` and `child` resources; RAIF maps their positions to separate typed placeholders, such as `parent`↦`R1` and `child`↦`R2`. The resources are tracked separately, blocks sharing either placeholder are related during dispatch, and mutation preserves their observed identifiers. `Commit()` is classified as a `submit` API because it submits accumulated operations and forms a transaction boundary. Based on these entries, RAIF can generate a transaction that first creates a resource, then applies operations to the same resource identifier, and finally ends the block with a submit API.

Generating Resource Based Transactions. Using the IPC API list, RAIF builds transaction blocks around selected service resources. A later IPC request is often meaningful only when it refers to the same service resource prepared by earlier requests. If IPC requests are selected independently, they may be well formed but still operate on unrelated or nonexistent resources. Therefore, RAIF first generates transaction blocks that preserve resource dependencies, instead of directly generating isolated IPC calls.

```
{
    "api": "CreateDisplayNode",
    "role": "create",
    "resource": "DisplayNode",
    "resource_args": {"return": "R1"}
},
{
    "api": "SetBounds",
    "role": "operate",
    "resource": "SurfaceRenderNode",
    "resource_args": {"nodeId": "R1"}
},
{
    "api": "RSBaseNodeAddChild",
    "role": "operate",
    "resource": "RenderNode",
    "resource_args": {"parent": "R1", "child": "R2"}
},
{
    "api": "Commit",
    "role": "submit",
    "resource": "Transaction",
    "resource_args": {}
}
```

Figure 5. Example entries in the RenderService IPC API list.

In detail, a transaction block is a short sequence of IPC APIs issued under one client context and ended by a `submit` API. A block may introduce a resource, operate on an existing resource, or remove a resource. It does not need to contain a complete create, operate, and submit pattern. For each block, RAIF selects compatible service resources, chooses matching `create`, `operate`, or `submit` APIs from the IPC API list, and maps every resource-bearing argument position to a placeholder such as `R1` or `R2`. For example, RAIF can

end blocks containing `CreateNode(R1)`, `SetBounds(R1)`, or `RemoveNode(R1)` with `Commit()`. These blocks are resource centered, as they preserve the same resource placeholder across multiple operations before concrete arguments are generated. In Figure 4, argument generation denotes preparing argument templates while retaining resource placeholders, rather than instantiating concrete arguments.

Generating Multi Client Workloads with Resource Dependencies. After generating transaction blocks, RAIF assembles them into multi client workloads. However, simply creating multiple clients is not enough if their requests operate on unrelated resources. Therefore, RAIF places transaction blocks that share at least one resource into different client sequences. Each client sequence keeps its own request order and will later be executed under a separate client context. This preserves per client state while exposing resource dependencies across clients.

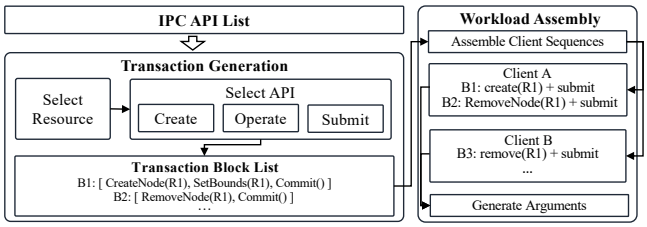


Figure 6. Resource dependent workload generation.

In Figure 6, the workload assembler places blocks over the same resource placeholder into different client sequences. For example, client A receives `B1` and `B2`, which create and remove `R1`, while client B receives `B3`, which also removes `R1`. At this point, `R1` is still a placeholder that preserves the intended resource dependencies across clients. After assembly, RAIF generates concrete IPC arguments by instantiating placeholders and filling other non resource arguments.

4.2 Resource Aware Transaction Dispatch

Multi-client workloads may still operate on unrelated resources or update them far apart. RAIF addresses this by preserving client contexts and scheduling blocks that share a resource at submit boundaries. After generating a multi client workload, RAIF executes it at transaction block granularity. The goal is not to control the internal thread scheduling, but to reduce the distance between related client transactions in the request order exposed to the fuzzer. If the fuzzer finishes one client sequence before running another, related cross-client operations may be separated by many unrelated requests and applied in different update windows. Instead, RAIF executes one transaction block until its submit API, and then tries to schedule another client's block that shares a resource placeholder.

Transaction Execution. Each client sequence is bound to a stable client identifier, so requests in the same sequence are handled under the same service side client context. This is necessary because many IPC services store pending changes in client local transaction state. For example, an update request may modify one client's pending transaction, and a later `submit` from the same client makes the update visible to the service. Therefore, RAIF does not switch clients in the middle of a transaction block. A block ends at a submit API recorded in the IPC API list, such as commit or flush.

Resource Guided Scheduling. As shown in Algorithm 1, at each fuzzing iteration RAIF either generates a new workload (lines 1–3) or mutates a corpus seed using its recorded resource feedback (lines 4–7). To execute the current workload, RAIF first initializes stable client contexts and execution positions (lines 8–10). It considers only each client's next unexecuted block and selects an initial block (line 11). RAIF executes the selected block under its client context through the final submit (lines 12–13). After execution, it advances only that client's position and retains the executed block's resource references (lines 14–15). For the next selection, it prioritizes a block from another client sharing a resource placeholder with the executed block; otherwise, it falls back to the next unfinished client sequence (line 16). This policy preserves each client's order while keeping related transactions close at submit boundaries.

Algorithm 1: RAIF Fuzzing Loop.

```
   Input: IPC API list A and corpus C
1  for each fuzzing iteration do
2      if generating a new workload or C = ∅ then
3          W ← GenerateResourceWorkload(A);
4      else
5          S, F_S ← SelectSeed(C);
6          W ← PreserveResourceRelations(S, F_S);
7          W ← InsertResourceBlocks(W, F_S, A);
8      InitClients(W);
9      next[c] ← 0 for each client c;
10     previous ← None;
11     B ← SelectNextBlock(W, next, previous);
12     while B ≠ None do
13         ExecuteBlock(B, Client(B));
14         next[Client(B)] ← next[Client(B)] + 1;
15         previous ← B;
16         B ← SelectNextBlock(W, next, previous);
17     K, F ← CollectFeedback(W);
18     if NewCoverage(K, C) or
         NewResourceBehavior(F, C) then
19         C ← SaveSeed(C, W, K, F);
```

This dispatch policy does not guarantee a particular internal service thread order. Instead, it gives the service a request order in which related resource updates, removals, and submits are placed near each other. For bugs that depend on

asynchronous update windows, this is only a request-level control point. RAIF does not force a binder, render, or callback thread interleaving. Instead, it improves request-level locality by avoiding schedules where related transactions are separated by many unrelated requests or executed under unrelated client contexts. Runtime feedback and later mutation then preserve the resource dependencies that were actually executed by the service.

4.3 Resource Behavior Guided Mutation

Intended resource identifiers may fail lookup, and mutation may break dependencies that the service executed. RAIF addresses this by recording resolved dependencies after unmarshalling and lookup, then preserving or extending them during mutation. Coverage feedback tells RAIF which code was reached, but it does not tell whether a request operated on a resource, or whether two transactions touched the same service resource. RAIF therefore records resource behavior after IPC unmarshalling and lookup to guide later mutation.

Runtime Behavior Collection. During workload execution, source level hooks record which service operations and resources are actually exercised. This is necessary because generated inputs describe intended requests, but do not show whether they reach service logic or which resource identifiers are used after unmarshalling. Therefore, RAIF collects runtime feedback at source level, after the service has unmarshalled an IPC request and dispatched it to a concrete resource operation. This position is different from the IPC transport layer, since the service has already interpreted the request, resolved the operation name, and exposed the resource identifier used by the operation.

For each executed resource operation, RAIF records a compact event containing the operation name, resource kind, resource identifiers when available, and a small state category such as create, update, remove, or submit. The identifiers are not treated as a complete model of the internal objects. Instead, they are used as stable keys when the service exposes them, such as node and layer identifiers. When a service does not expose a stable identifier, RAIF falls back to operation level feedback for that resource kind.

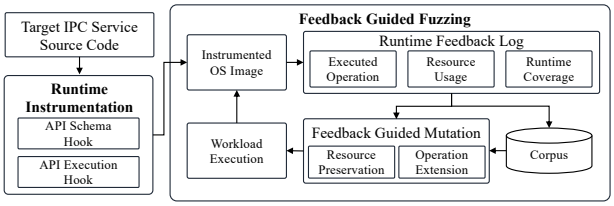


Figure 7. Illustration of resource behavior collection and feedback-guided mutation.

As shown in Figure 7, RAIF uses two types of source level instrumentation, which may be placed at multiple service-specific sites. Schema registration records static metadata such as the operation name, type, resource kind, and resource argument positions. Execution hooks report resolved operations and resource identifiers after unmarshalling and lookup. This allows RAIF to interpret an executed command as a concrete service operation rather than only as a numeric command identifier. These records are written to the runtime feedback log and later used by mutation.

The instrumentation is target specific but follows the same pattern across services. For RenderService, RAIF records events in the command processing path after command unmarshalling. For Android services, RAIF records events at the corresponding transaction or lifecycle operation paths, such as layer or display transaction handling in SurfaceFlinger and codec lifecycle operations in MediaCodec. The inserted code only emits compact resource events and does not participate in workload generation or scheduling.

Feedback Guided Mutation. After all client sequences finish, RAIF collects coverage and runtime resource feedback (Algorithm 1, line 17). A workload is kept with its feedback for new coverage or resource behavior, including newly observed resource identifiers or dependencies across client sequences (lines 18–19). This allows RAIF to preserve workloads that may not immediately increase coverage but already exercise useful resource behavior.

In later fuzzing iterations, RAIF selects a seed together with the feedback recorded for its earlier execution (line 5). This is necessary because a valid workload may depend on multiple requests operating on the same resource to trigger meaningful behavior. If mutation changes resource identifiers independently, it may break this dependency and turn a resource-centered workload into disconnected requests. Therefore, when an observed resource identifier appears in multiple related transactions, RAIF either keeps these occurrences unchanged or rewrites them consistently to the same new identifier (line 6). Then, RAIF extends the workload around observed resources. If runtime feedback shows that a certain operation was performed on a resource, RAIF selects additional APIs that operate on that resource and inserts compatible transaction blocks (line 7). If no runtime feedback is available for a resource, RAIF falls back to the default resource-centered generation strategy.

As shown in Figure 7, RAIF applies two feedback-guided mutation strategies. First, executed resource preservation keeps identifiers consistent across related transactions. If two observed transactions operate on resource N, mutation keeps both occurrences of N or rewrites them to the same new identifier. This preserves resource dependencies exercised during execution, rather than only encoded in the input. Second, resource behavior extension adds new operations around observed resources. Given an observed resource N, RAIF queries the API list for APIs operating on the same resource kind and inserts compatible transaction blocks using N. For example, after observing `SetBounds(N)`, RAIF may add `RemoveFromTree(N)` over the same node N.

5 Implementation

We implement RAIF on top of Syzkaller and reuse its corpus management, executor framework, crash detection, and coverage collection. We extend Syzkaller's generation, mutation, program encoding, and executor to represent resource dependent multi client workloads as pseudo call sequences, serialize per client stream metadata, execute each stream until submit boundaries, and attach runtime resource feedback to the corresponding seeds. The API list is drafted from interface header files and developer documentation with LLM assistance, then manually validated.

Target Adaptation. Adapting RAIF to a new IPC service requires three service specific components, while the generation, scheduling, feedback management, corpus management, and crash detection logic are shared across targets. First is to generate an *API list* that records each API's role and resource kind. The AOSP and OpenHarmony headers define their interfaces; RAIF converts and validates these definitions as pseudo calls and wrappers rather than guessing APIs. The prompt specifies the expected pseudo-call structure and argument template, and the model generates signatures and arguments. The validation checks whether an API creates reusable service resources, consumes existing resource identifiers, submits pending updates, and uses the recorded argument positions after unmarshalling.

Second is to write pseudo call wrappers that map Syzkaller arguments to concrete IPC requests and transaction commands. These wrappers are target-specific because each service has its own marshaling format, command encoding, and client setup logic. However, they only connect RAIF's target-independent workload representation to the target IPC interface. They do not decide which resources to generate, how transactions are scheduled, or which feedback is preserved for mutation. Thus, most wrapper code is mechanical interface glue rather than new fuzzing policy.

Third is to add feedback hooks that register operation metadata and record resource events after IPC unmarshalling. For RenderService, the hooks are placed after RSCommand unmarshalling and around paths that bind clients or commit transactions. For Android services, the hooks record service-specific resource events, such as layer transactions in SurfaceFlinger, dispatcher and window events in InputFlinger, and codec lifecycle or buffer queue events in MediaCodec.

Most adaptation code consists of mechanical wrappers, syscall specifications, and generation glue; the semantic work is validating resource metadata, identifying submit APIs, and locating feedback paths and fields. For SurfaceFlinger, InputFlinger, MediaCodec, and RenderService, manual validation covered 71, 27, 102, and 215 API-list entries; adaptation added 704, 295, 1,006, and 7.8K lines of target-specific code and 3, 2, 1, and 2 hook sites, respectively. All hooks reuse a shared 379-line runtime. Each Android service took about one person-day, while RenderService took seven person-days. The D-Bus adaptation required 1,642 lines of code for 38 pseudo-calls and a 1,597-line execution wrapper.

The LLM generated 394 of 415 complete pseudo-call entries without correction, including 66 of 71 for SurfaceFlinger, 26 of 27 for InputFlinger, 98 of 102 for MediaCodec, and 204 of 215 for RenderService; we manually corrected the remaining 21, mainly involving nested structures. Separately, 410 of 415 resource annotations were correct; correcting five errors took minutes. These errors can cause invalid bindings or rejected requests; runtime feedback can discard unaccepted dependencies but cannot recover missing or misclassified APIs. Extraction takes about 1–2 hours per service and approximately 260–520 total tokens per pseudo-call.

6 Evaluation

In this section, we evaluate RAIF with the following three research questions.

- **RQ1:** How effective is RAIF in detecting real bugs in IPC based OS services?
- **RQ2:** How does RAIF compare with existing fuzzers in exploring service logic and resource behavior?
- **RQ3:** How do RAIF's design components affect overall fuzzing effectiveness?

6.1 Evaluation Setup

Targets. Our main evaluation covers four IPC based OS services from OpenHarmony and Android. On OpenHarmony, we use RenderService [23], a graphics service with persistent render nodes, surfaces, and display resources. On Android, we use SurfaceFlinger [3], InputFlinger [1], and MediaCodec [2], covering graphics, input, and media services. We additionally evaluate RAIF on D-Bus to examine its applicability beyond these platforms.

Baselines. We compare RAIF with Syzkaller, NASS [21], and FANS [19]. All baselines use the same time budget, device configuration, permissions, and target services when applicable. Syzkaller uses the same IPC wrappers and Syzlang resource and type annotations as RAIF, but does not use resource based workload generation, resource aware transaction dispatch, or resource behavior guided mutation. NASS and FANS are evaluated on Android targets because they are designed for Android native services and do not support OpenHarmony's ZIDL and RSCommand model. The service side instrumentation used for resource metrics is passive for all baselines and is not used to guide their fuzzing.

Metrics. We use branch coverage to measure service logic exploration and service side resource metrics to measure resource behavior exploration. Specifically, we report seven metrics. (1) *branch coverage* counts unique branches reached during fuzzing. (2) *Resource state* counts unique tuples of service, resource kind, operation kind, and compact state category recorded at resource operation sites. (3) *Transition*

counts state-changing resource events executed by the service. (4) *Tran/S* reports transitions per resource state as an auxiliary density metric. (5) *Attempted* counts generated IPC operations that target service resources. (6) *Accepted* counts attempted resource operations executed at service side resource operation sites after IPC unmarshalling. (7) *Same-Res* counts how often executed operations in the same workload reuse the same service resource identifier.

Running Configuration. For bug detection on OpenHarmony, Android services, and D-Bus, we run RAIF for one week on each target and manually analyze reported crashes. We then use ThreadSanitizer to identify potential data races exposed by the generated workloads. RAIF controls transaction and submit order rather than internal service thread interleavings. For effectiveness analysis, each fuzzer runs for 24 hours on each target, repeated five times following fuzzing evaluation best practices [17]. All experiments are conducted on a 64 core AMD machine running Ubuntu 22.04.

6.2 Bug Detection Capability

To answer **RQ1** and evaluate RAIF's bug detection capability, we collected and analyzed the findings reported by RAIF.

In total, RAIF identified 23 bugs on OpenHarmony, Android services, and D-Bus. Of the OpenHarmony bugs, 14 have been confirmed and fixed.

Table 1. List of bugs identified by RAIF.

#	System	Failure Type	Function(s)	Status
1	OHOS	Logic error	`RSSurfaceRenderNodeDrawable`	Fixed
2	OHOS	Stack overflow	`RSRenderParams`	Fixed
3	OHOS	Stack overflow	`RSPaintFilterCanvas`	Fixed
4	OHOS	Stack overflow	`GetAllSubSurfaceNodes`	Fixed
5	OHOS	Stack overflow	`node_cast`	Fixed
6	OHOS	Data race	`lastSendingPid`	Fixed
7	OHOS	Data race	`RSVsyncRateReduceManager`	Fixed
8	OHOS	Data race	`RSRenderNodeDrawable`	Fixed
9	OHOS	Data race	`totalProcessNodeNum`	Fixed
10	OHOS	Stack overflow	`CollectSurface`	Fixed
11	OHOS	Stack overflow	`HasDisappearingTransition`	Fixed
12	OHOS	Logic error	`RSProperty`	Fixed
13	OHOS	Stack overflow	`SetUIFirstSwitch`	Fixed
14	OHOS	Data race	`rsDumpCallbackDirector`	Fixed
15	AOSP	Fatal assertion	`LayerHierarchy::traverse`	Reported
16	AOSP	Logic error	`convertCategory`	Reported
17	AOSP	Logic error	`convertFrameRateSelectionStrategy`	Reported
18	AOSP	Logic error	`convertCompatibility`	Reported
19	AOSP	Null-ptr deref	`processPointerId`	Reported
20	AOSP	Fatal assertion	`processMouseEventLocked`	Reported
21	AOSP	Logic error	`onWindowInfosChanged`	Reported
22	Linux	Data race	`_dbus_message_trace_ref`	Reported
23	Linux	Data race	`dbus_connection_dispatch`	Reported

Bug Statistics. Table 1 lists all 23 bugs detected by RAIF. Among these bugs, 14 were found in OpenHarmony RenderService and have been fixed by maintainers, while 7 were found in AOSP services, and 2 were found in D-Bus, and have been reported. The bugs include logic errors, stack overflows, data races, fatal assertions, and a null-pointer dereference. OpenHarmony RenderService contains the largest number of bugs in our evaluation, covering drawable state, render parameters, surface traversal, animation state, frame-rate management, pipeline parameters, and screen-property handling. AOSP SurfaceFlinger contributes 4 bugs, mainly caused by insufficient validation of transaction state and layer hierarchy structure. The remaining AOSP bugs affect InputFlinger. Under the same time budget, Syzkaller did not trigger the 14 OpenHarmony bugs, and Syzkaller, NASS, and FANS did not trigger the seven AOSP bugs.

Syzkaller's independent call and identifier mutation may break resource relations across transactions; NASS and FANS improve request validity and handler reachability but do not organize requests around accepted resources and submit boundaries. RAIF constructs blocks around shared resources, executes related blocks in stable client contexts near submit APIs, and uses post-unmarshalling feedback to preserve executed dependencies during mutation. The ablation study on the four OpenHarmony and Android services supports the contribution of these mechanisms: full RAIF finds 21 bugs, compared with 11 for RAIF-Resource, 14 for RAIF-Guide, and 9 for RAIF-NoDispatch.

Severity Analysis. For the OpenHarmony and Android bugs, stack overflows (#2–#5, #10, #11, and #13), fatal assertions (#15 and #20), and a null-pointer dereference (#19) can terminate services. Data races (#6–#9 and #14) may expose inconsistent state, while logic errors (#1, #12, #16–#18, and #21) affect rendering correctness. The observed OpenHarmony and Android impacts are denial of service and UI instability; the D-Bus findings (#22–#23) are data races, without demonstrated denial of service or UI impact.

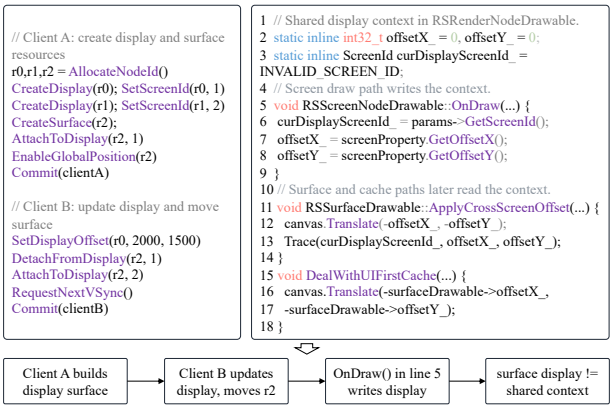
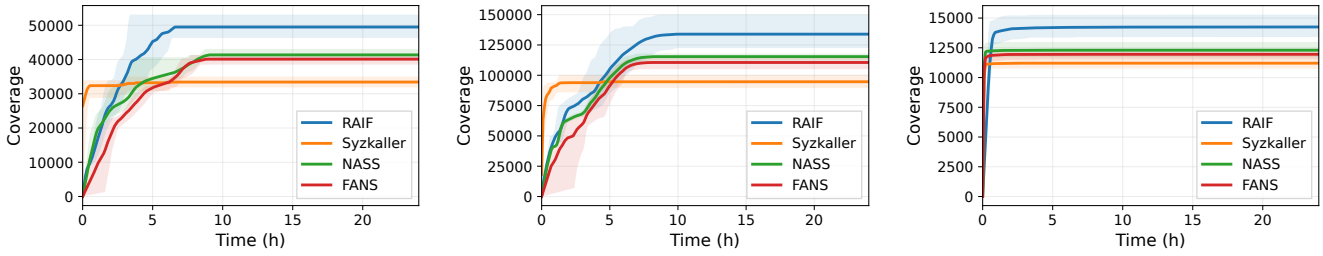


Figure 8. Illustration of an inconsistent pipeline state bug.

Case Study. We use bug #1 in Table 1 as a case study to show how RAIF exposes a resource dependent bug. RAIF generates a workload with two clients that builds display and surface states and perturbs them near a render update. As shown on the left of Figure 8, Client A creates two displays and one surface `r2`, attaches `r2` to display 1, enables global positioning, and commits the setup. Client B then updates the offset



(a) Branch Coverage on SurfaceFlinger. (b) Branch Coverage on InputFlinger. (c) Branch Coverage on MediaCodec.

Figure 9. Coverage growth over 24 hours on SurfaceFlinger, InputFlinger, and MediaCodec.

of display `r0`, moves `r2` from display 1 to display 2, requests the next VSync, and commits the update. This sequence keeps the same surface `r2` across transactions and submits the surface migration through `RequestNextVSync()` and `Commit()`, which makes the changed state visible to a following render update, where RenderService updates display state and `OnDraw()` writes the display context.

However, RenderService stores display-specific context, including `offsetX_`, `offsetY_`, and `curDisplayScreenId_`, in shared drawable fields (lines 1-3). `OnDraw()` writes these fields during screen rendering (lines 5-9), while `ApplyCross ScreenOffset()` and UIFirst cache rendering later read them (lines 11-17). When `r2` moves to display 2 but the fields retain another display's context, its surface path applies an incorrect offset, causing inconsistent drawable pipeline state.

Triggering this bug requires creating valid display and surface resources, preserving `r2` across setup and update transactions, moving it across displays while changing display offset state, and submitting the update through the update boundary so the render pipeline observes it. Removing any condition prevents the bug in our minimized workload: without setup, later operations fail resource lookup; without preserving `r2`, the two clients operate on unrelated surfaces; without the cross display move, the display context remains consistent with the surface attachment; and without `RequestNextVSync()` and `Commit()`, the changed state is not driven into the render pipeline at the required point.

Previous fuzzers are less likely to trigger this bug as it requires more than a valid IPC request or handler reachability. A flat Syzkaller style workload may invoke the same APIs but mutate calls and resource identifiers independently, breaking the same surface relation across transactions. Interface aware IPC fuzzers such as NASS and FANS improve request validity and handler reachability, but they do not explicitly organize client transactions around a shared surface and a render update boundary. In contrast, RAIF builds the display surface relation, preserves it across transactions, and places the surface migration adjacent to the submit operations in the request order. Runtime feedback guides mutation to preserve dependencies exercised during execution.

6.3 Overall Effectiveness Analysis

To answer **RQ2**, we compare RAIF with Syzkaller, NASS, and FANS on SurfaceFlinger, InputFlinger, and MediaCodec, and with Syzkaller on D-Bus. We report average results over five 24 hour fuzzing runs.

Runtime Coverage. As shown in Table 2, RAIF achieves the highest coverage on all three Android services. On SurfaceFlinger, RAIF improves coverage by 48.1% over Syzkaller, 19.7% over NASS, and 23.4% over FANS. On InputFlinger, RAIF improves coverage by 41.5%, 16.1%, and 21.1% over the three baselines, respectively. On MediaCodec, RAIF improves coverage by 27.2%, 15.9%, and 19.2%, respectively. On D-Bus, RAIF improves coverage by 6.7% over Syzkaller. On the three Android services, a Mann-Whitney U test shows significant gains over all baseline fuzzers ($p = 0.007937$).

Table 2. Coverage comparison with baselines. Coverage is the number of unique covered branches.

Tools	SurfaceFlinger	InputFlinger	MediaCodec	D-Bus
RAIF	**49493.4**	**133906.6**	**14249.8**	**3169.2**
Syzkaller	33428.6 (+48.1%)	94656.8 (+41.5%)	11206.2 (+27.2%)	2969.8 (+6.7%)
NASS	41359.0 (+19.7%)	115375.0 (+16.1%)	12297.8 (+15.9%)	—
FANS	40116.0 (+23.4%)	110544.8 (+21.1%)	11951.4 (+19.2%)	—

Figure 9 shows that RAIF also reaches higher coverage throughout the 24-hour runs. Compared with Syzkaller, RAIF benefits from generating transaction blocks that preserve resource dependencies across IPC requests, rather than treating service calls as isolated pseudo calls. Compared with NASS and FANS, RAIF further benefits from organizing requests around service resources and client contexts. Thus, RAIF reaches more service code under the same time budget.

Resource Behavior Statistics. Table 3 and Table 4 further examine whether the coverage gain corresponds to deeper resource behavior. As shown in Table 3, RAIF consistently triggers the most resource transitions across the three services, indicating that it drives more state-changing resource events under the same time budget. On SurfaceFlinger and MediaCodec, RAIF also reaches the highest resource state count among the evaluated fuzzers. InputFlinger

Table 3. Resource state comparison with baselines.

Service	Tool	Resource State	Transitions	Tran/S
SurfaceFlinger	RAIF	**6080.0**	**35385.2**	5.8
	Syzkaller	256.0 (+2275.0%)	1407.8 (+2413.5%)	5.5
	NASS	912.0 (+566.7%)	13338.0 (+165.3%)	14.6
	FANS	1104.0 (+450.7%)	14443.6 (+145.0%)	13.1
InputFlinger	RAIF	66716.6	**244461.4**	**3.7**
	Syzkaller	**124296.4** (-46.3%)	75865.6 (+222.2%)	0.6
	NASS	110183.2 (-39.4%)	122270.4 (+99.9%)	1.1
	FANS	49050.2 (+36.0%)	160614.0 (+52.2%)	3.3
MediaCodec	RAIF	**12329.8**	**9642.2**	**0.8**
	Syzkaller	11325.8 (+8.9%)	8183.0 (+17.8%)	0.7
	NASS	11858.0 (+4.0%)	8558.4 (+12.7%)	0.7
	FANS	11715.6 (+5.2%)	8459.6 (+14.0%)	0.7

shows a different pattern, where Syzkaller and NASS reach more resource states, but RAIF triggers the most transitions and achieves the highest Tran/S. This suggests that RAIF does not simply maximize distinct state labels; instead, it tends to keep reached resources active and repeatedly drives them through state-changing operations.

Table 4. Resource behavior comparison with baselines.

Service	Tool	Attempted	Accepted	Same-Res
SurfaceFlinger	RAIF	**62359709.8**	**22555741.6**	**6012.0**
	Syzkaller	44977583.0 (+38.6%)	17590042.8 (+28.2%)	256.0 (+2248.4%)
	NASS	28760129.8 (+116.8%)	10856600.8 (+107.8%)	912.0 (+559.2%)
	FANS	31555918.0 (+97.6%)	11901979.2 (+89.5%)	1102.6 (+445.3%)
InputFlinger	RAIF	**249330.0**	**184211.0**	**27269.6**
	Syzkaller	216963.6 (+14.9%)	143787.0 (+28.1%)	18329.0 (+48.8%)
	NASS	218270.0 (+14.2%)	150613.4 (+22.3%)	25492.4 (+7.0%)
	FANS	235389.8 (+5.9%)	167963.2 (+9.7%)	17603.8 (+54.9%)
MediaCodec	RAIF	**5595.6**	**5591.6**	**4349.0**
	Syzkaller	3910.8 (+43.1%)	3906.8 (+43.1%)	2813.8 (+54.6%)
	NASS	4090.2 (+36.8%)	4086.2 (+36.8%)	2935.0 (+48.2%)
	FANS	4096.0 (+36.6%)	4092.0 (+36.6%)	2942.2 (+47.8%)

Table 4 further shows how resource operations are organized. Across all three services, RAIF issues more attempted and accepted resource operations and achieves the highest same resource count. This indicates that RAIF reuses the same resource more often across operations, which is consistent with the higher transition count. On InputFlinger, this also explains why RAIF has fewer resource states than Syzkaller and NASS but higher Transition and Tran/S, since it keeps reached resources active through more repeated operations. Overall, RAIF improves not only code reachability but also resource behavior exploration. Averaging the relative improvements across the nine comparisons of the three Android services and three baselines gives 25.8% higher branch coverage and 3.5× more resource transitions.

6.4 Component-Wise Effectiveness Analysis

To answer **RQ3**, we evaluate how RAIF's design components affect fuzzing effectiveness by comparing the full system with corresponding ablation variants.

Ablation Variants. We compare RAIF with three variants. RAIF-Resource keeps only resource based workload generation. RAIF-Guide keeps workload generation and transaction dispatch, but disables runtime resource feedback. RAIF-NoDispatch keeps workload generation and runtime feedback, but disables resource aware transaction dispatch. These variants isolate the effects of resource based generation, transaction dispatch, and runtime feedback.

Coverage Statistic. Table 5 summarizes the coverage results. Compared with RAIF-Resource, RAIF improves coverage by 6.7% to 20.8%. Compared with RAIF-Guide, RAIF improves coverage by 12.5% to 18.2%. Compared with RAIF-NoDispatch, RAIF improves coverage by 11.8% to 20.0%. A Mann-Whitney U test shows significant gains over all ablation variants ($p = 0.007937$).

Table 5. Average coverage comparison in the ablation study.

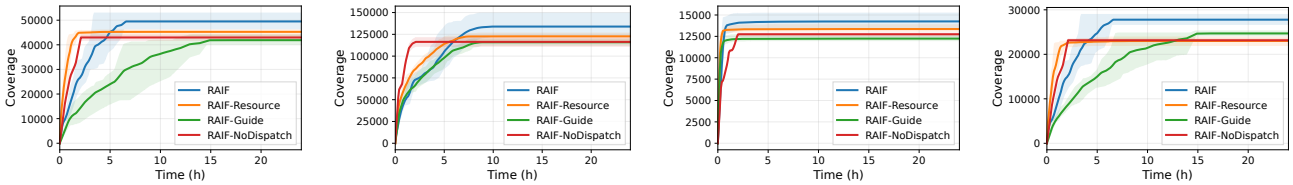
Tools	SurfaceFlinger	InputFlinger	MediaCodec	RenderService
RAIF	**49493.4**	**133906.6**	**14249.8**	**27785.2**
RAIF-Resource	45264.8 (+9.3%)	122706.2 (+9.1%)	13358.4 (+6.7%)	23007.0 (+20.8%)
RAIF-Guide	41879.6 (+18.2%)	115920.4 (+15.5%)	12240.6 (+16.4%)	24694.2 (+12.5%)
RAIF-NoDispatch	42984.8 (+15.1%)	116348.2 (+15.1%)	12748.2 (+11.8%)	23157.8 (+20.0%)

These results show that resource based generation, transaction dispatch, and runtime feedback are complementary. RAIF-Guide adds dispatch but lacks feedback, so it can narrow exploration before useful resource dependencies are known. RAIF-NoDispatch keeps feedback and outperforms RAIF-Guide on the three Android services, but still falls behind the full RAIF on all four services.

Table 6. Resource-state comparison in the ablation study.

Service	Tool	Resource State	Transitions	Tran/S
SurfaceFlinger	RAIF	**6080.0**	**35385.2**	5.8
	RAIF-Resource	2688.0 (+126.2%)	22640.4 (+56.3%)	8.4
	RAIF-Guide	2432.0 (+150.0%)	21936.4 (+61.3%)	9.0
	RAIF-NoDispatch	2432.0 (+150.0%)	14314.4 (+147.2%)	5.9
InputFlinger	RAIF	66716.6	**244461.4**	3.7
	RAIF-Resource	93458.4 (-28.6%)	155102.8 (+57.6%)	1.7
	RAIF-Guide	**105421.0** (-36.7%)	117300.4 (+108.4%)	1.1
	RAIF-NoDispatch	21917.0 (+204.4%)	56516.6 (+332.5%)	2.6
MediaCodec	RAIF	12329.8	**9642.2**	**0.8**
	RAIF-Resource	9620.6 (+28.2%)	7047.8 (+36.8%)	0.7
	RAIF-Guide	**12420.8** (-0.7%)	9286.4 (+3.8%)	0.7
	RAIF-NoDispatch	4178.6 (+195.1%)	2993.2 (+222.1%)	0.7
RenderService	RAIF	**8305.0**	**6477.0**	**0.8**
	RAIF-Resource	6429.6 (+29.2%)	4696.8 (+37.9%)	0.7
	RAIF-Guide	6791.2 (+22.3%)	5055.8 (+28.1%)	0.7
	RAIF-NoDispatch	7085.8 (+17.2%)	5221.2 (+24.1%)	0.7

Resource State Statistic. Table 6 further evaluates how each component affects resource-state exploration. Across all four services, RAIF triggers more transitions than all variants in the ablation study. Compared with RAIF-Resource, RAIF improves Transition by 56.3%, 57.6%, 36.8%, and 37.9% on SurfaceFlinger, InputFlinger, MediaCodec, and RenderService, respectively. Compared with RAIF-Guide, the improvements



(a) Coverage on SurfaceFlinger. (b) Coverage on InputFlinger. (c) Coverage on MediaCodec. (d) Coverage on RenderService.

Figure 10. Coverage growth over 24 hours in the ablation study.

are 61.3%, 108.4%, 3.8%, and 28.1%. Compared with RAIF-NoDispatch, the improvements are 147.2%, 332.5%, 222.1%, and 24.1%. This shows that transaction dispatch and feedback are complementary for more effective resource exploration.

Resource state shows a more nuanced pattern. On InputFlinger and MediaCodec, some variants reach more distinct resource states, but they trigger fewer transitions. This indicates that they touch more distinct resource-state categories, while RAIF drives reached resources through more state-changing operations. Table 7 shows the same trend: RAIF achieves the highest attempted operations, accepted operations, and same resource reuse across all four services. Overall, resource-based generation builds the workload structure, transaction dispatch keeps related operations close in the request order, and runtime feedback preserves executed resource dependencies for later mutation.

Table 7. Resource behavior comparison in the ablation study.

Service	Tool	Attempted	Accepted	Same-Res
SurfaceFlinger	RAIF	**65088343.6**	**22555741.6**	**6019.2**
	RAIF-Guide	59635946.8 (+9.1%)	17924493.4 (+25.8%)	2681.2 (+124.5%)
	RAIF-Resource	39576403.6 (+64.5%)	13195285.4 (+70.9%)	2025.4 (+197.2%)
	RAIF-NoDispatch	55324272.8 (+17.6%)	18661604.0 (+20.9%)	2432.0 (+147.5%)
InputFlinger	RAIF	**262519.2**	**194628.4**	**27269.6**
	RAIF-Guide	201046.4 (+30.6%)	142378.0 (+36.7%)	22660.4 (+20.3%)
	RAIF-Resource	208579.8 (+25.9%)	143055.2 (+36.1%)	24831.6 (+9.8%)
	RAIF-NoDispatch	216250.6 (+21.4%)	158968.0 (+22.4%)	21917.0 (+24.4%)
MediaCodec	RAIF	**6007.3**	**6003.3**	**4673.7**
	RAIF-Guide	4595.7 (+30.7%)	4591.7 (+30.7%)	3411.7 (+37.0%)
	RAIF-Resource	4645.7 (+29.3%)	4641.7 (+29.3%)	3387.3 (+38.0%)
	RAIF-NoDispatch	4182.6 (+43.6%)	4178.6 (+43.7%)	2993.2 (+56.1%)
RenderService	RAIF	**320038.8**	**236816.4**	**31186.0**
	RAIF-Guide	198994.2 (+60.8%)	138619.8 (+70.8%)	22178.4 (+40.6%)
	RAIF-Resource	265998.2 (+20.3%)	184455.0 (+28.4%)	22951.4 (+35.9%)
	RAIF-NoDispatch	300941.4 (+6.3%)	220168.2 (+7.6%)	30429.0 (+2.5%)

Bug Detection Statistic. We examine whether detected bugs on our target services depend on RAIF's components. RAIF-Resource finds 11 bugs: #2-5, #8, #11, #13, #16-18, and #21. RAIF-Guide finds 14 bugs: #2-4, #8-15, #17-18, and #21. RAIF-NoDispatch finds 9 bugs: #3-5, #7, #13, #15, #17-18, and #20. The full RAIF finds all 21 OpenHarmony and Android bugs (#1–#21) listed in Table 1, finding 10, 7, and 12 more bugs than the three variants, respectively. RAIF-Resource lacks the dispatch and feedback needed to turn generated relations into executed service side dependencies. RAIF-Guide lacks post unmarshalling feedback, so it cannot tell which dependencies are actually accepted by the service. RAIF-NoDispatch observes accepted dependencies, but does not place related client transactions around submit APIs where these dependencies become visible. Bugs #1, #6, and #19 are missed by all three variants, suggesting that they require the full combination of these components to trigger.

Instrumentation Overhead. We measure the overhead of RAIF's resource feedback hooks by running each target for one hour with hooks enabled and disabled. With hooks disabled, Resource State and transition drop to zero on all four services, confirming that these metrics come from service side observations rather than fuzzer inputs. Hooks reduce throughput because they record executed resource events after IPC unmarshalling, with throughput decreasing from 1175.4 to 620.6 inputs/s on SurfaceFlinger, 213.2 to 132.2 on InputFlinger, 122.8 to 50.4 on MediaCodec, and 110.4 to 78.6 on RenderService. The generated feedback logs remain compact: 4.22 MB/hour on SurfaceFlinger, 12.90 MB/hour on InputFlinger, 0.46 MB/hour on MediaCodec, and 5.48 MB/hour on RenderService. This overhead is a tradeoff for the improved resource behavior preservation and bug detection.

API-List Validation. We reran RAIF using the unvalidated LLM-generated API list. Compared with the validated list, branch coverage decreased from 27,785.2 to 25,839.2 on RenderService (7.0%), from 49,493.4 to 45,781.4 on SurfaceFlinger (7.5%), from 133,906.6 to 129,889.4 on InputFlinger (3.0%), and from 14,249.8 to 13,394.8 on MediaCodec (6.0%).

7 Related Work

Kernel Fuzzing. Kernel fuzzing has been widely studied for testing the reliability and robustness of operating system kernels. Syzkaller [32] uses system call descriptions to generate structured syscall programs and iteratively evolves the generated programs using runtime coverage feedback [33]. KSG [29], SyzDescribe [11], SyzGen [8], and SyzGPT [37] generate or infer syscall specifications, interface constraints, dependencies, or seeds for kernel fuzzing. For execution control, Razzer [13] and Krace [34] bias syscall executions toward racy kernel schedules, SegFuzz [14] guides fuzzing with scheduling points, and DDRace [36] and Ozz [12] target concurrent memory accesses between different threads to expose kernel concurrency bugs.

Resource-Aware Fuzzing. SyzGen++ [7] statically infers producer-consumer handle relations, but does not capture service objects' dynamic creation, update, and submission. FUZZNG [5] reshapes pointers and file descriptors with generic hooks, but cannot track IPC resource identifiers across requests and clients. RAIF instead observes accepted identifiers after lookup, reuses them across clients, and schedules related operations at submit boundaries.

Protocol Fuzzing. Protocol fuzzers guide message construction and sequencing using protocol-state abstractions. Peach [31] uses manually specified formats and state models. AFLNet [25] and StateAFL [22] guide sequences with response codes and memory snapshots, while DynPRE [20] rewrites session identifiers. Instead, RAIF constructs transaction blocks around shared resource identifiers, executes them in stable client contexts around submit boundaries, and observes accepted resources after unmarshalling and lookup.

IPC Service Fuzzing. IPC service fuzzing has been studied for testing cross process interfaces exposed by system services, applications, and browser components. Prior work improves IPC fuzzing by recovering message formats, interface models, transaction dependencies, or service feedback. BinderCracker [9] records binder transactions and reconstructs transaction schemas. FANS [19] extracts Android native service interface models and dependencies from source code. NASS [21] recovers RPC interface definitions for Android services and uses stable service related coverage feedback. MachFuzzer [35] infers message formats and ordering constraints for macOS Mach IPC services. Other work studies Android Intent fuzzing and Firefox IPC fuzzing [15, 16], also emphasizing structured message generation.

Prior IPC fuzzers model message schemas, request orders, or interface dependencies for request validity. These dependencies help fuzzers generate valid requests, but they do not directly capture whether the service accepts a resource identifier, which client owns it, whether an update crosses a submit API, or whether another client later operates on the same executed resource. RAIF focuses on this service level resource behavior: it preserves executed resource identifiers, executes related transaction blocks around submit APIs, and uses post-unmarshalling feedback to keep resource dependencies executed by the service. RAIF is complementary to prior IPC fuzzers; it does not replace schema recovery or interface-model extraction, but organizes generated IPC requests into resource based workloads.

8 Discussion

Limited control over service internal scheduling. RAIF controls the order in which client transaction blocks are submitted to the service, but it does not control how the service schedules its internal threads. After an IPC request enters the service, it may be processed by binder thread pools, main service threads, render threads, callback queues, or worker tasks. Therefore, RAIF provides request-level resource locality rather than deterministic thread scheduling. Its dispatch policy places related transactions close to submit APIs and to each other, so that the service has more opportunities to observe related resource updates in nearby execution windows. This is useful for bugs that depend on accepted resource dependencies, but it does not guarantee a specific internal thread interleaving. Bugs that require an exact order between service threads may still be missed. Explicit control of internal threads would require additional instrumentation, which can perturb timing and increase timeout noise during testing. A future direction is to combine transaction dispatch with lightweight scheduling signals, such as task enqueue events, callback entry points, or render-thread progress.

Timing control in asynchronous pipelines. Some IPC services continue processing a submitted update after the submit API returns. For example, RenderService may enqueue render tasks, update caches, invoke callbacks, or apply changes in later frames. In this setting, RAIF's feedback means that the service accepted and executed a resource operation at the observed point, but it does not mean that all downstream asynchronous work has completed. Thus, RAIF uses such feedback mainly to preserve accepted resource dependencies and guide later mutations around them. It does not claim to control render-thread timing, callback timing, or frame-level progress. Moreover, although the feedback hooks are lightweight, source-level instrumentation can still perturb fine-grained timing in highly asynchronous paths. Future work can add a timing aware feedback, such as task start or callback entry, so that the fuzzer can derive executions toward request orders and delay windows that are more likely to expose more timing-sensitive bugs.

9 Conclusion

This paper presents RAIF, a fuzzer for IPC based OS services. RAIF targets the gap between reaching IPC handlers and exercising service logic. To explore this behavior, RAIF generates resource based workloads, dispatches related client transactions around submit APIs and uses service side feedback to preserve resource behavior and guide further mutation. Our evaluation identifies 23 bugs on OpenHarmony, Android, and D-Bus. On the three Android services, RAIF has an average improvement of 25.8% in branch coverage and triggers 3.5× more resource-state transitions than baseline fuzzers. These results show that organizing IPC service fuzzing around resource behavior is effective for testing stateful OS services above the IPC transport layer.

Acknowledgments

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62525207,62472448).

References

[1] Android Open Source Project. 2026. InputFlinger source code. https://android.googlesource.com/platform/frameworks/native/+/master/services/inputflinger. Accessed: 2026-05-27.

[2] Android Open Source Project. 2026. MediaCodecService source code. https://developer.android.com/reference/android/media/MediaCodec. Accessed: 2026-05-27.

[3] Android Open Source Project. 2026. SurfaceFlinger and WindowManager. https://source.android.com/docs/core/graphics/surfaceflinger-windowmanager. Accessed: 2026-05-27.

[4] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. 2024. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In *31st ACM SIGSAC Conference on Computer and Communications Security, CCS 2024* (Salt Lake City, UT, USA) *(CCS '24)*. ACM, New York, NY, USA, 1315–1329. doi:10.1145/3658644.3690320

[5] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions. doi:10.14722/ndss.2023.24688

[6] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *39th IEEE Symposium on Security and Privacy, SP 2018*. IEEE Computer Society, 711–725.

[7] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4661–4677. doi:10.1109/SP54263.2024.00269

[8] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. 2021. SyzGen: Automated Generation of Syscall Specification of Closed-Source macOS Drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 749–763.

[9] Huan Feng and Kang G. Shin. 2016. Understanding and Defending the Binder Attack Surface in Android. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 398–409.

[10] Google. 2020. surfaceflinger new issue. https://issuetracker.google.com/issues/160631698. Accessed: 2020-06-07.

[11] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. SyzDescribe: Principled, Automated, Static Generation of Syscall Descriptions for Kernel Drivers . In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 3262–3278. https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179298

[12] Dae R. Jeong, Yewon Choi, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2024. OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles*. 229–248.

[13] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. [n. d.]. Razzer: Finding Kernel Race Bugs through Fuzzing. In *40th IEEE Symposium on Security and Privacy*. 754–768.

[14] Dae R. Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. SegFuzz: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *44th IEEE Symposium on Security and Privacy*. 2104–2121. doi:10.1109/SP46215.2023.10179398

[15] Anatoli Kalysch, Mark Deutel, and Tilo Müller. 2020. Template-based Android Inter Process Communication Fuzzing. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*.

[16] Christoph Kerschbaumer and Christian Holler. 2021. Effectively Fuzzing the IPC Layer in Firefox. https://blog.mozilla.org/attack-and-defense/2021/01/27/effectively-fuzzing-the-ipc-layer-in-firefox/. Accessed: 2026-05-26.

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804

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

[19] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. 2020. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 307–323.

[20] Zhengxiong Luo, Kai Liang, Yanyang Zhao, Feifan Wu, Junze Yu, Heyuan Shi, and Yu Jiang. 2024. DynPRE: Protocol Reverse Engineering via Dynamic Inference. In *Proceedings of the 2024 Network and Distributed System Security Symposium*. doi:10.14722/ndss.2024.24083

[21] Philipp Mao, Marcel Busch, and Mathias Payer. 2025. NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage. In *34th USENIX Security Symposium*. 4225–4243.

[22] Roberto Natella. 2022. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering* 27, 191 (2022).

[23] OpenHarmony Project. 2026. OpenHarmony graphic_graphic_2d: RenderService source code. https://gitee.com/openharmony/graphic_graphic_2d. Accessed: 2026-05-27.

[24] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743.

[25] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 460–465. doi:10.1109/ICST46399.2020.00062

[26] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1980–1997. doi:10.1109/TSE.2019.2941681

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

[28] Leo Stone, Rishi Ranjan, Stefan Nagy, and Matthew Hicks. 2023. No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4913–4929.

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

[30] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. *HEALER: Relation Learning Guided Kernel Fuzzing*. ACM, New York, NY, USA, 344–358.

[31] Peach Tech. 2013. Peach Fuzzer. https://gitlab.com/peachtech/peach-fuzzer-community

[32] Dmitry Vyukov. 2015. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller

[33] Dmitry Vyukov and Andrey Konovalov. 2015. Syzlang: System Call Description Language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md

[34] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *41st IEEE Symposium on Security and Privacy, SP 2020*. 1643–1660.

[35] Kun Yang, Hanqing Zhao, Chao Zhang, Jianwei Zhuge, and Haixin Duan. 2019. Fuzzing IPC with Knowledge Inference. In *2019 IEEE 38th International Symposium on Reliable Distributed Systems*. 11–20.

[36] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. 2023. DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing. In *32nd USENIX Security Symposium*. USENIX Association, 2849–2866.

[37] Zhiyu Zhang, Longxing Li, Ruigang Liang, and Kai Chen. 2025. Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-Based RAG. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025).