DROIDFUZZ: Proprietary Driver Fuzzing for Embedded Android Devices

Jianzhong Liu*, Yuheng Shen*, Yifei Chu*, Qiang Zhang†, Heyuan Shi‡, Wanli Chang†, Yu Jiang*✉
*School of Software, Tsinghua University †Hunan University ‡Central South University
Emails: *{liujz21, shenyh20, chuyf24}@mails.tsinghua.edu.cn, jiangyu198964@126.com
†zhangqiang9413@126.com, wanli.chang.rts@gmail.com ‡hey.shi@foxmail.com

***Abstract*—Embedded Android Devices have proliferated in many security-critical embedded scenarios, requiring sufficient testing to root out vulnerabilities. Due to Android's architecture, which uses a Hardware Abstraction Layer (HAL) for vendor-specific driver implementations, traditional kernel testing techniques cannot detect such bugs within the actual driver logic, which are commonly proprietary and vendor-specific. In this paper, we propose DroidFuzz, an embedded Android system fuzzer that targets such vendor-specific driver implementations to find such bugs. Through leveraging pre-testing HAL driver probing, kernel-user relational payload generation, and cross-boundary execution state feedback, we effectively test the proprietary drivers in both the kernel and the HAL layer. We implemented DroidFuzz and evaluated its effectiveness on 7 embedded Android devices, and found 12 security-critical previously unknown bugs, all of which have been confirmed by the respective vendors.**

***Index Terms*—Android testing, fuzz testing, embedded Linux, bug detection**

I. INTRODUCTION

The widespread adoption of Android in mobile devices has led to its increasing presence in embedded systems, where it is used to power a diverse range of applications, from set-top boxes and automated kiosks to automotive systems, medical devices, and industrial control systems. As the complexity of these systems grows, so does the attack surface, with device drivers emerging as a critical component that interacts directly with hardware components and manages their operations. With millions of lines of code in modern Android-based embedded systems, ensuring the security and reliability of device drivers has become a pressing concern.

Android's architecture utilizes a Hardware Abstraction Layer (HAL) to allow high-level applications to interface with hardware devices and peripherals. HAL is situated between the Android framework and the Linux kernel, where it provides a standardized interface for accessing hardware resources, while also allowing hardware vendors to implement proprietary and closed-source code that is specific to their hardware platforms. As a result, much of the device driver's logic resides within the HAL layer, such that a significant portion of the driver codebase remains opaque and inaccessible to external scrutiny, making it challenging to detect and mitigate potential security vulnerabilities or bugs within the HAL layer.

We use the example of CVE-2021-0673 [26] to demonstrate this point. This vulnerability is a heap-buffer-overflow found in the audio component of MediaTek chipset's HAL, which at the time accounted for 43% of Android devices shipped. When exploited, the bug results in privilege escalation, allowing malicious attackers to gain access to the entire system. Due to MediaTek's HAL being closed source and proprietary, applying public scrutiny to ensure the safety of its components is difficult, hindering the bug discovery process, consequently demanding more effective and systematic methods in rooting out such vulnerabilities in Android proprietary drivers.

Software testing is a crucial step in ensuring the reliability, security, and quality of modern software systems. Among various testing techniques, fuzz testing (fuzzing) has emerged as an effective approach for detecting bugs and vulnerabilities in software applications. It mainly involves providing large amounts of randomized input to a target program to trigger bugs within. In the context of operating system kernels, fuzzing typically involves generating payloads that mimic real-world workloads and injecting them into the kernel through various interfaces such as system calls. Syzkaller [39] is a state-of-the-art kernel fuzzer, and has found more than 5000 bugs in the Linux kernel that have subsequently been confirmed and fixed.

However, despite the importance of driver security, existing research and industry application of Android fuzzing has primarily focused on fuzzing and testing applications and system frameworks, leaving a significant gap in the state of the art for effective fuzzing techniques tailored to detecting bugs in Android's device drivers, consisting of code in kernel drivers and HAL drivers. Established methods for kernel fuzzing on Android mainly focuses on testing the Linux kernel itself, whereas Android's HAL layer, which is in the userspace, cannot be directly reached using these methods. Furthermore, effectively covering Android's kernel drivers require meaningful interaction with its corresponding HAL, which is difficult to emulate using simply system call invocations.

This research aims to address this knowledge gap by exploring new methodologies for adapting fuzzing techniques to the unique characteristics of HAL drivers in embedded Android frameworks. To do this, we need to solve the following challenges effectively. 1) First, in addition to system call interfaces, we need to model the interfaces exposed by HAL to the Android runtime and libraries such that we can generate effective invocations and payloads to the HAL's interfaces, which is proprietary and mostly undocumented, barring traditional static analysis or LLM-based methods from achieving such goals. 2) Second, we face the critical problem of effectively generating test cases that are meaningful to both Android's kernel and HAL, which are further complicated as HAL's underlying functionalities interact with the kernel through system calls, thus direct randomized and static generation yields poor results. 3) Finally, as the kernel and HAL run on different permission modes and memory spaces, and provide differing execution state tracking, we cannot directly interpret the execution state of both entities directly to identify interesting inputs, as the semantics of these are vastly different.

To address the aforementioned challenges, we propose DROIDFUZZ, a fuzzer targeting proprietary drivers in embedded Android devices. In order to effectively conduct fuzz testing on both Android's kernel drivers and HAL layer, DROIDFUZZ introduces three approaches that individually address the challenges. First, DROIDFUZZ provides a *pre-testing HAL driver probing* pass, where the HAL is loaded, probed, to obtain its exposed interfaces, argument types, and the weighting factors for each interface. Second, DROIDFUZZ also integrates a *kernel-user relational payload generation* technique, where it allows the fuzzer to produce test cases that synergistically

tests the kernel drivers and HAL components in an embedded Android device. Finally, we design a *cross-boundary execution state feedback* mechanism, where it coalesces the kernel's code coverage and HAL's kernel execution behavior to analyze the state changes in both components, and in turn assist in further input generation to produce more meaningful input payloads.

We implemented DROIDFUZZ on 7 embedded Android devices from well-known hardware vendors, including Xiaomi, Sunmi, Raspberry Pi, etc., and evaluated its driver code coverage and bugs finding capabilities on these devices. We further compared our performance to that of Syzkaller to demonstrate the effectiveness in tapping into embedded Android's proprietary driver framework. Our results show that DROIDFUZZ found 12 new bugs in these firmware, which have 7 from kernel drivers and 5 from the HAL layers, where Syzkaller was only able to find 2, both of which are from the kernel. Furthermore, through evaluating per-driver coverage in the kernel, DROIDFUZZ achieves a 17% increase on average, demonstrating that interactions with HAL can create meaningful workloads that further traverse the states of the kernel drivers.

II. BACKGROUND AND RELATED WORK

A. Android on Embedded Devices

Embedded Android is the use of the Android operating system in embedded devices, such as automated kiosks, set-top boxes, medical devices, and industrial control systems [13]. These devices typically use various peripherals such as displays, cameras, and sensors, which often require many proprietary drivers to run [43].

Android in embedded systems allows developing complex applications, through its high-level APIs and abstractions that other embedded OSs fail to offer [32]. As shown in Figure 1, an embedded Android system consists of the Linux kernel, the Hardware Abstraction Layer (HAL), Android Runtime and Libraries, and the Android Applications [12]. The Linux kernel [27] provides the underlying system services such as scheduling, process isolation, and kernel drivers, whereas Android's framework provides the core APIs for building applications. The HAL is sandwiched between the two, providing a standardized interface for accessing hardware resources and abstracting away low-level details [14].

B. Fuzz Testing

Fuzz testing, a.k.a. fuzzing, is a software testing technique used to detect security vulnerabilities and bugs in applications [21]–[23]. It mainly involves feeding invalid, unexpected, or random input to an target program and monitoring its behavior for crashes, hangs, or other anomalous behavior. Coverage-guided mutation-based fuzzing uses genetic algorithms to evaluate the effectiveness of test cases by whether they trigger new execution states to preserve those for further mutation into new test cases [16], [31]. Generation-based fuzzing uses input format specifications to produce valid and well-formed test cases. Fuzz testing is widely used in various domains, including protocol security [19], program library security [2]–[4], [10], and system vulnerability detection [5], [20].

Syzkaller is a state-of-the-art fuzzer that leverages both coverage-guided mutation-based fuzzing and generation-based fuzzing techniques through the use of *kcov* [35] and *Syzlang* [40]. Many works use Syzkaller as a basis for improving fuzzing techniques. For instance, Moonshine [28] aims to distill high-quality initial test cases for Syzkaller. Horus [17] reduces data transfer overheads between the fuzzer and manager by offloading RPCs to direct memory accesses.

kAFL [33] is another kernel fuzzer that maximizes throughput by leveraging architectural features in Intel processors. HEALER [37] is a kernel fuzzer targeted at system call relation learning to generate more effective test cases.

There are also works that fuzz tests Android's systems and framework [36], [38]. Difuze [6] addresses the problem of kernel driver fuzzing by introducing *interface-aware fuzzing*, which extracts valid commands and associated data structures through static analysis. Atlas [42] uses static analysis to deduce correct calling sequences and parameters of native APIs in closed-source Android native libraries and uses heuristics for optimizing the generated harness. FuzzGen++ [29] generates fuzzing driver programs for OEM Android libraries and applies automatic cutoff for low-quality driver programs. In comparison, DROIDFUZZ addresses the problem of efficient proprietary embedded Android driver fuzzing, which also requires the joint fuzzing of both the Linux kernel and the HAL layer.

Works aimed towards fuzzing embedded systems [1], [15], [24], [25], [41] are appearing. Tardis [34] adapts coverage-guided fuzzing to embedded operating systems running on emulators. EmbSan [18] addresses the issue of generalized porting of sanitizers to embedded platforms. Gustave [7] transfigurates embedded OS fuzzing into application fuzzing through input conversion.

III. MOTIVATION

Unlike traditional Android application or framework fuzzing, testing proprietary drivers requires generating effective payloads that trigger complex interactions between the kernel and HAL components. To demonstrate the difficulties involved, we show how an embedded Android application accesses the hardware in Figure 1 to demonstrate the complex interactions between HAL drivers and kernel drivers. As shown in the figure, Android's HAL resides between the high-level application frameworks and the low-level kernel, where it accesses low-level interfaces through device files and abstract them into high-level APIs. HAL drivers are also stateful and have vendor-specific features, allowing proprietary drivers to be shipped without source code, common practice for Android vendors.

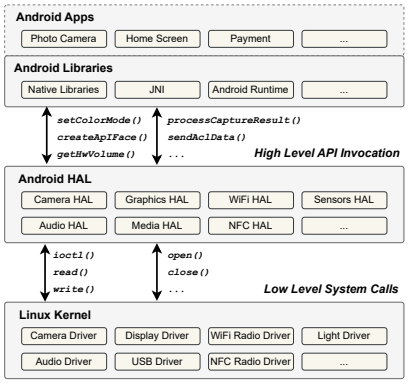


Fig. 1. Architecture of Android Relative to Apps Accessing Hardware

Fuzzing only kernel system calls or the HAL interfaces as a library is not sufficient to trigger complex and meaningful interactions between the HAL and kernel drivers, as profound bugs often require correct states from both components to trigger. Therefore, our goal is to develop a method that can fuzz proprietary drivers, by identifying driver interfaces and interpreting their affinity with other APIs and system calls, i.e. *relations*, generate meaningful sequences of API invocations and system calls that direct fuzzing towards exploring states in the proprietary drivers, and interpret execution feedback from both the kernel and the HAL layer as coverage, ultimately providing an effective way to detect bugs in embedded Android's

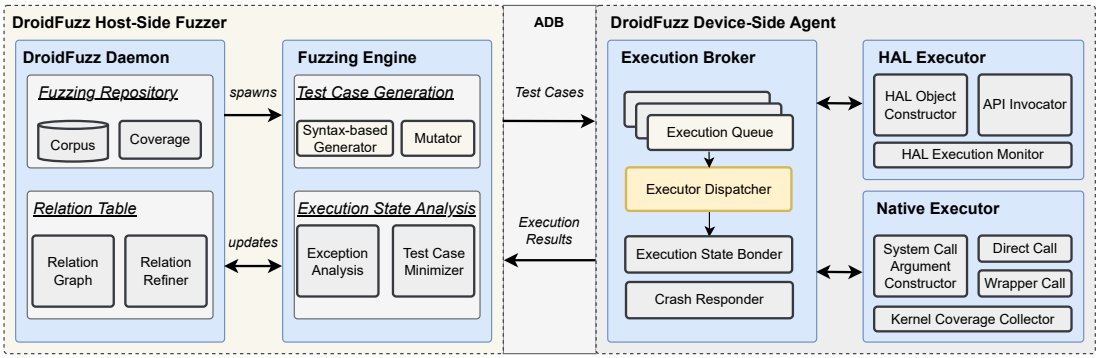


Fig. 2. Overview of DROIDFUZZ's Architecture

drivers. Designing such methods poses several challenges that need to be adequately addressed:

First, we need to determine the list of interfaces to test and the argument syntax and semantics of each individual interface. While previous work such as Difuze and Syzkaller have provided methods and syntax for system calls, the interfaces on the HAL layer are largely undocumented and lack available source code, making code-level analysis difficult.

Second, the challenge lies in generating meaningful sequences of HAL API invocations and system calls that effectively test both sides synergistically. The kernel and HAL are tightly coupled, with the HAL providing a standardized interface for accessing hardware resources, while the kernel provides the underlying operating system services. To detect profound bugs, we need to generate sequences of API invocations and system calls that trigger these interactions and exercise both components simultaneously.

The last challenge is interpreting the execution feedback as coverage also poses significant challenges due to its cross-boundary characteristics. The HAL layer is closed-source, making it difficult to obtain precise information about the internal workings of this component. Furthermore, the kernel's behavior is deeply intertwined with the HAL's behavior, making it challenging to determine whether a particular sequence of API invocations and system calls has effectively exercised both components.

IV. DROIDFUZZ DESIGN

DROIDFUZZ is an operating system fuzzer that proposes solutions to the aforementioned challenges. It takes inspiration from state-of-the-art kernel fuzzers, including techniques such as system call description-based generation and coverage-guided mutation. We show the overall architecture diagram of DROIDFUZZ in Figure 2. The major components of the fuzzing harness include the host-side Daemon and Fuzzing Engine, and the device-side Execution Broker, HAL Executor, and Native Executor, of which each component will be briefly outlined in Section IV-A.

Our primary contributions are mainly covered in the following three designs: 1) a *pre-testing HAL driver probing* pass to obtain the HAL's exposed interfaces and associated argument types, and assess *weights* of each interface (Section IV-B); 2) a *kernel-user relational payload generation* approach that allows the fuzzer produce test cases that jointly tests the HAL drivers and corresponding kernel drivers (Section IV-C); 3) a *cross-boundary execution state feedback* mechanism that interprets the HAL's execution behavior to merge with the kernel's code coverage for uniform analysis of new states, and further produce more meaningful input payloads (Section IV-D).

A. Fuzzing Harness and Execution Agents

To facilitate proprietary driver fuzzing for embedded Android devices, we compartmentalize each required functionality into separate components, including DROIDFUZZ's Daemon, Fuzzing Engines, Execution Brokers, and Executors for HAL and Native system calls.

The root process of DROIDFUZZ is the Daemon, which mainly coordinates synchronization between fuzzing processes and maintains persistent data, such as the *seed corpus*, overall coverage statistics, and relation table which records relations' weights between interfaces and system calls.

When the daemon finishes initialization, it spawns one instance of a Fuzzing Engine for each device. The Fuzzing Engine produces test cases for execution on the target device, and subsequently analyzes feedback for each execution. Communication to the target device leverages the Android Debug Bridge (ADB) [11]. Test cases generated are sequences of HAL interface and Linux kernel system call invocations in a Domain Specific Langauge (DSL) form. The engine also receives execution state information from the device to test against previous runs for any new behavior.

The aforementioned two components are run on the host machine, while the following components are run on the device-under-test.

The Execution Broker is responsible for reliably communicating test cases to run and execution results with its parent fuzzing engine. It spawns a HAL executor and Native executor to run each HAL API or system call invocation. It maintains an internal execution queue that holds all test cases waiting to be executed, where each element of a sequence is dispatched according to their type. The feedback is then bonded to form a uniform feedback statistic, and is passed back to the fuzzing engine for analysis.

The HAL and Native Executor uses the instantiated DSL and constructs corresponding dependencies of objects as required. It then invokes the relevant API or system call, after which its execution feedback is collected and returned to the broker.

B. Pre-Testing HAL Driver Probing

Direct extraction of HAL interfaces is not easily achieved, as their descriptions from proprietary HALs in release builds of the firmware are not readily available. Instead, we take a poke and probe approach to observe actual invocations of the HAL from Android applications and extract a set of interfaces through observation of how the Android framework communicates with the HAL.

Essentially, we use two components, a Poke application running atop the Android framework, and a probe utility that runs natively, i.e. without Android's abstractions. The probe utility first uses Android

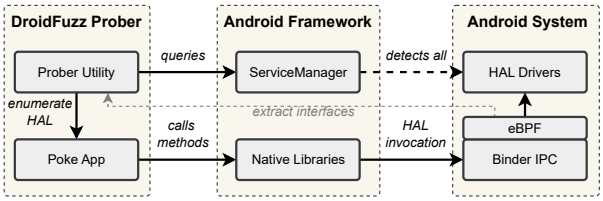


Fig. 3. Process of HAL Interface Probing

utilities (e.g. *lshal* and ServiceManager) to obtain a list of running services and HALs on the device. Then, for each running HAL service, it passes the relevant information to the Poke application, which then requests the service's interface through Android's ServiceManager. Next, the probe utility inserts eBPF code into the kernel that monitors for accesses to Binder IPC from the Poke app. The Poke application then conducts a short trial of all exposed interfaces by marshaling the parameters and invoking the corresponding methods reflected by ServiceManager, which is called into the native libraries, where it is translated into Binder IPC calls, during which the interaction is recorded from the eBPF hooks. The prober utility extracts the actual IPC data between the HAL and the Poke App, and filters out relevant interfaces and arguments. This process is depicted in Figure 3.

We also rank a HAL interface by calculating normalized occurrence through observing the number of instances that a certain interface is triggered through invoking high-level Android APIs. This gives the fuzzer more information on which interfaces to invoke during testing.

C. Kernel-User Relational Payload Generation

Generating effective test cases that traverse deeply into the execution states of drivers in both the kernel and HAL requires the fuzzer to understand the *relations* between HAL interfaces and kernel system calls. To do this, we construct a *relation graph* $\mathrm{G_{rel}} = (\mathrm{V}, \mathrm{E})$, where the set of vertices $\mathrm{V} = \{(v, w) | v \in S \vee v \in H, w \in (0, 1)\}$, of which S is the set of individual system calls and H is the set of individual HAL interfaces, and edges in E being directed and weighted. Each vertex also carries a fixed weight w, which represents the ranking of each individual system call or HAL interface, and corresponds to the probability at which the system call or interface is chosen during generation as the *base invocation*. The direction of edges in E represents the perceived dependency between the kernel system calls or HAL interfaces, whereas the weights represent the confidence regarding the dependencies between the system calls or interfaces.

During initialization, we create the relation graph with the set of vertices being filled with all system calls and HAL interfaces, and their weights from either system call descriptions or evaluated interface weights. The set of edges is initialized to an empty set, i.e. $E = \emptyset$. As we fuzz Android drivers, we detect new coverage and update the edges and weights between the HAL APIs and kernel system calls accordingly. When a new coverage is detected, we *minimize* the call to the bare bones API and system calls, ensuring that only the most essential invocations that trigger the same execution behavior are exercised. The new relations are then saved into the relation graph with the maximum normalized weights through the following process. For each adjacent pair of calls a and b, where the dependency is $a \rightarrow b$, their weights are calculated using the formula:

$$w_{(a,b)} = 1 - \Sigma_{\forall e=(x,b), x \neq a} w_{(x,b)}/2 \quad (1)$$

whereas weights for other edges with the same endpoint are halved.

Upon generating an input payload, we randomly pick a system call or an interface as the *base invocation*, i.e. the API or system call that acts as the basis for all depending invocations to form a sequence, based on the weights of each vertex. We instantiate the call in the DSL with parameters and objects based on the descriptions, using a combination of syntax-based generation and historical payload mutation. After this, we traverse the relation graph from the current vertex to a dependent vertex with a probability based on the edge weight. We may choose to stop altogether if the random value dictates such. Then, we repeat the process given above. After ending such a search, we iterate over all calls sequentially, find unresolved argument or parameter values, and find *producer calls*, i.e. system calls or API invocations that return the required argument values, instantiate the call accordingly, and insert it into the call sequence as a prefix to the current call. Finally, the generated call sequence, in the DSL representation, is passed to the device for execution.

To encourage more diverse payloads and prevent our fuzzing process from getting stuck in a local optimum, we periodically reduce the weights of all learned relations by multiplying all edge weights with a factor less than 1. This reduction process incentivizes DROIDFUZZ to explore different interaction paths between the HAL APIs and kernel system calls during test case generation, leading to more comprehensive coverage of the Android drivers.

D. Cross-Boundary Execution State Feedback

To conduct effective fuzzing of Android drivers in both the kernel and HAL components, we need to gather execution state feedback that spans across these two boundaries. For the Linux kernel, we can directly utilize debugging facilities such as *kcov* by recompiling the kernel with relevant configurations enabled. However, due to the closed-source nature of most vendor-specific HALs, we cannot directly employ tools such as LLVM's SanitizerCoverage [30] to evaluate its code coverage.

Our observation is that, as HAL abstracts low-level details into high-level interfaces, we can monitor the system call invocation behavior of the HAL layer to identify new behavior. However, we cannot directly interpret kernel code coverage, as it only records the system calls and code blocks executed, but disregards the order in which they are executed.

Thus, we use *directional* system call invocation coverage to reflect the execution behavior of the HAL layer's code. This requires that the HAL executor inserts eBPF probes into the kernel at runtime to detect system calls originating from the HAL. When the HAL executor detects system calls originating from the HAL layer, we record the specifics, including its number, critical position arguments (e.g. *request* in *ioctl()*), and the order in which it appears.

To mimic the output of code coverage, we use a lookup table compiled at initialization consisting of all possible system calls, including *specialized* system calls, which divide system calls that take generalized argument (e.g. *ioctl()*) according to their critical arguments and assign them unique IDs. We arrange the coverage into a sequence of system call IDs that correspond to the specific system calls invoked by the HAL, and append it to the kernel code coverage figures obtained from *kcov*. This joint-state feedback represents a comprehensive view of the execution state of both the kernel and HAL components, allowing us to understand how these two components interact and identify potential issues in their interaction. The analysis logic for both types of coverage remain the same, allowing for simplified processing while gaining comprehensive understandings of how the kernel and HAL components interact.

V. Implementation and Evaluation

We implemented DroidFuzz using 25930 lines of Rust and Go for the daemon, fuzzing engine, and broker components, 1694 lines of C for the executor components and prober utility, and 503 lines of Java for the Poke App. Our implementation borrowed system call descriptions and native executor components from Syzkaller to execute test payloads intended for the kernel. To demonstrate our applicability to real-world embedded Android devices, we adapted DroidFuzz to test 7 devices, which encompass off-the-shelf systems and development boards, all from renowned hardware vendors in the mobile and embedded space using processors commonly used in embedded Android devices. The list of the devices are shown in Table I, with their vendors, CPU architecture, the AOSP version and kernel version of their firmware specified.

TABLE I
List of Embedded Android Devices Tested

ID	Device	Vendor	Arch.	AOSP	Kernel
A1	Phone Dev Board	Xiaomi	*aarch64*	15	6.6
A2	Tablet Dev Board	Xiaomi	*aarch64*	15	6.6
B	Pi 5	Raspberry Pi	*aarch64*	15	6.6
C1	Commercial Tablet	Sunmi	*aarch64*	13	5.15
C2	Cashier Kiosk	Sunmi	*aarch64*	13	5.15
D	LubanCat 5	EmbedFire	*aarch64*	13	5.10
E	UP Core Plus	AAEON	*amd64*	13	5.10

To evaluate the effectiveness of our approach, we ran DroidFuzz on these embedded devices to assess its ability to find real-world bugs on the 7 embedded devices, and its effectiveness in covering driver code compared to state-of-the-art fuzzing approaches. We also analyzed the individual contributions of pre-testing HAL driver probing, kernel-user relational payload generation, and cross-boundary execution state feedback by performing ablation testing that removes each component for a controlled experiment.

A. Evaluation Settings

All experiments are conducted on a host server with dual Intel Xeon Silver 4120R CPUs, 192GiB of RAM, and running *amd64* Debian Linux 12.8. We use one device per experiment to conduct testing. The devices used in this experiment are running rooted firmware with their kernels recompiled with *kcov* and KASAN [9] enabled. DroidFuzz is configured to reboot the target devices upon encountering any bugs during testing, including kernel panics, assertions, and HAL errors. The versions of Syzkaller and Difuze used are commits *fb88827* and *3290997*. Each experiment is repeated for 10 times to eliminate statistical errors. We use the Mann-Whitney U Test to assess the existence of statistical significance, where data groups that do not exhibit such significance will be labelled explicitly.

B. Bug Detection

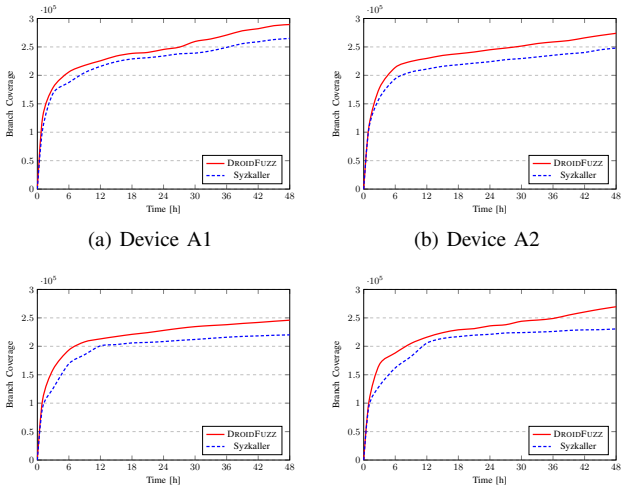
To evaluate DroidFuzz's bug finding abilities, we ran DroidFuzz on each target embedded device for 144 hours. All bugs triggered were initially minimized, deduplicated, and reproduced. Manual effort was also involved in analyzing the bugs to detect duplicates and recover corrupted log messages. Over the course of testing, DroidFuzz found 12 new bugs in the devices' respective firmwares, both in the kernel and HAL layer. All bugs have been confirmed by the respective vendors, where bugs fixes are underway at the time of writing. Some bugs have redacted information in their Bug info column as relevant details are currently embargoed.

The list of bugs are shown in Table II. As shown in the table, the new bugs found by DroidFuzz are found in both the kernel and HAL components, demonstrating that DroidFuzz is capable of effectively finding bugs in vendor-specific HAL drivers itself. Specifically, 3 bugs triggered crashes in the HAL layer, whereas the other 9 bugs were found in the kernel, showing the versatility of DroidFuzz, being capable of jointly fuzzing drivers in both the system kernel and HAL layer. As expected, the bugs triggered in the HAL are all architectural exceptions such as segmentation faults, demonstrating the effectiveness of the test cases generated. Furthermore, we can predict that this bug list is not conclusive, as some bugs triggered in the HAL layer were not perceivable without the use of sanitizers or other bug detection tools.

C. Kernel Coverage

To understand why DroidFuzz was able to find new bugs, we conduct an experiment that uses code coverage of the entire kernel and specific targeted kernel drivers as a proxy to identify the covered states of the drivers in the Android system. We then compare DroidFuzz's statistics to that of Syzkaller and Difuze. Syzkaller is also able to generate test cases for kernel drivers, and thus we can perceive the additional code coverage that jointly fuzzing the HAL layer brings. Difuze is an interface fuzzer that specifically generates *ioctl()* calls to device drivers, thus allowing us to compare

1) Comparison with Syzkaller: We compared DroidFuzz's kernel code coverage on all tested devices with that of Syzkaller's to understand the effect on kernel driver execution state exploration through joint testing the HAL layer. We run each experiment for 48 hours, and take the average coverage at each timestamp.



(a) Device A1 (b) Device A2

(c) Device B (d) Device C1

Fig. 4. Coverage comparison between DroidFuzz, Syzkaller, on select tested devices over 48 hours.

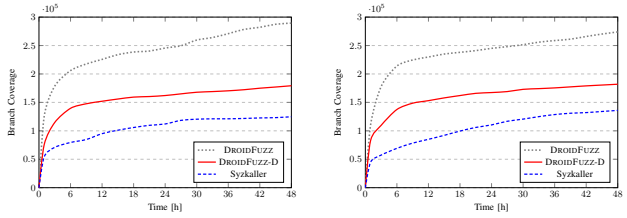
We show the relevant coverage for devices A1, A2, B, and C in Figure 4, whereas the the coverage statistics for devices D, E, and F follow this pattern and are omitted. As shown in the plot, DroidFuzz achieves better coverage than Syzkaller consistently, demonstrating the effectiveness of our approach towards testing Android's kernel and HAL layers together in triggering more execution states, particularly in the kernel, which also shows that system call fuzzers such as Syzkaller still fall short of providing realistic and meaningful payloads to the kernel drivers.

TABLE II
LIST OF ALL NEW BUGS FOUND BY DROIDFUZZ.

№	Device	Bug Info	Bug Type	Component
1	A1: Xiaomi Phone Dev Board	WARNING in rt1711_i2c_probe	Logic Error	Kernel Driver
2	A1: Xiaomi Phone Dev Board	Native crash in Graphics HAL (redacted)	Memory Related Bug	HAL
3	A1: Xiaomi Phone Dev Board	BUG: looking up invalid subclass: NUM	Logic Error	Kernel Subsystem
4	A1: Xiaomi Phone Dev Board	WARNING in tcpc (redacted)	Logic Error	Kernel Driver
5	A2: Xiaomi Tablet Dev Board	Infinite Loop in driver (redacted)	Logic Error	Kernel Driver
6	A2: Xiaomi Tablet Dev Board	Native crash in Media HAL (redacted)	Memory Related Bug	HAL
7	A2: Xiaomi Tablet Dev Board	KASAN: invalid-access in hci_read_supported_codecs	Memory Related Bug	Kernel Driver
8	B: Raspberry Pi 5	WARNING in l2cap_send_disconn_req	Logic Error	Kernel Subsystem
9	C1: Sunmi Commercial Tablet	Native crash in Camera HAL (redacted)	Memory Related Bug	HAL
10	C2: Sunmi Cashier Kiosk	WARNING in rate_control_rate_init	Logic Error	Kernel Driver
11	D: LubanCat 5	KASAN: slab-use-after-free Read in bt_accept_unlink	Memory Related Bug	Kernel Driver
12	E: AAEON UP Core Plus	WARNING in v4l_querycap	Logic Error	Kernel Driver

Thus, we conclude that jointly testing both the kernel and the HAL layer results in more state transitions within the kernel itself, increasing the probability of discovering new bugs.

2) Comparison with Difuze: Difuze utilizes a fuzzer called MangoFuzz (built upon Peach [8]), which feeds *ioctl()* interface invocations based on extracted interfacesto the kernel. As it is based on components designed for legacy kernels targeting select chipsets manufacturers, we only adapted it to test devices A1 and A2, where it succeeded in extracting 285 and 232 driver interfaces from the firmware of devices A1 and A2. To understand the effect and quality of HAL's *ioctl()* calls reflected upon kernel code coverage, we derive another variant DROIDFUZZ-D which limits the executor and HAL to only calling *ioctl()*s, whereas other requests will be blocked.



(a) Device A1 (b) Device A2

Fig. 5. Coverage comparison between DROIDFUZZ, Difuze, and DROIDFUZZ-D for over 48 hours.

The results are shown in Figure 5. As shown in the plot, DROIDFUZZ's coverage far outpaces that of Difuze's, which is expected due to the increase in interfaces tested. A more relevant comparison is DROIDFUZZ-D and Difuze, which shows that DROIDFUZZ-D leads Difuze's coverage by 34%. Our analysis shows that while the invoked *ioctl()* remains the same, sending the requests through fuzzing HAL produces significantly improved results compared to specification-based generation techniques.

Thus, we conclude that jointly testing the kernel and HAL produced additional benefits in that the HAL traps into the kernel with more realistic and meaningful payloads, giving the fuzzer an edge in discovering new states.

D. Ablation Tests

We also wish to understand the effect of each design component on the overall performance of DROIDFUZZ, and thus introduce two variations of DROIDFUZZ, DROIDFUZZ-NoRel, and DROIDFUZZ-NoHCov, which have modifications that disable kernel-user relational payload generation thus solely rely on randomized dependency generation, and removes HAL system call directional coverage from cross-boundary execution state feedback, respectively. We still use kernel coverage as a proxy to convey a statistic on the scope of execution states traversed. We use Syzkaller's statistics as a baseline to better understand the effects. The overall results are shown in Table III.

TABLE III
COVERAGE STATISTICS FOR ABLATION TESTS (48H)

Device	DROIDFUZZ	DF-NoRel	DF-NoHCov	Syzkaller
A1	289402	269281	274021	264920
A2	273930	250192	259402	247958
B	245930	210583	225126	220094
C1	269593	246930	240194	230664
C2	279305	239382	259492	220950
D	250295	237492	232910	220175
E	298593	247294	284629	257320

1) Effect of Kernel-User Relational Payload Generation: We find that the statistics for removing relational generation produces coverage consistently less than DROIDFUZZ. This is due to the generated test cases contain less meaningful and potentially incorrect usage of the system calls and APIs, and thus cannot cover as much code in the kernel. However, it still manages to frequently cover more code than Syzkaller, which indicates that HAL's use of the system call interface can produce more valid semantics.

2) Effect of Cross-Boundary Execution State Feedback: Removing cross-state feedback also results in lower coverage statistics than DROIDFUZZ. The reason is that the fuzzer can only refine the relations based on kernel code coverage alone, and thus produces less-than-optimal test cases. Even so, it still consistently performs better than Syzkaller, as joint fuzzing of both the kernel and HAL can trigger more execution states within these components.

Therefore, we conclude that both relational payload generation and joint state feedback contributes to DROIDFUZZ's overall effectiveness, allowing DROIDFUZZ generate more high-quality test cases by refining relations between system calls and HAL interfaces.

VI. CONCLUSION

DROIDFUZZ leverages pre-testing HAL driver probing, kernel-user relational payload generation, and cross-boundary execution state feedback to jointly test drivers in both Android's HAL layer and kernel. Our evaluations show that DROIDFUZZ finds 12 bugs in 7 embedded Android devices, and covers execution states more effectively than the state of the art.

VII. ACKNOWLEDGMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), and NSFC Program (No. U2441238, 62021002).

References

[1] Richard Barry et al. Freertos, 2003. https://www.freertos.org/.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[4] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: Fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 499–513, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NTFUZZ: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 1973–1989, 2021.

[6] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2123–2138, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Stéphane Duverger and Anaïs Gantet. Gustave: Fuzz it like it's app. *DMU Cyber Week*, 2021.

[8] Michael Eddington. Peach fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/.

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

[10] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Annual Computer Security Applications Conference*, ACSAC '20, page 360–372, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Google Inc. Android debug bridge (adb). https://developer.android.com/tools/adb.

[12] Google Inc. Architecture overview. https://source.android.com/docs/core/architecture.

[13] Google Inc. Embedded. https://developer.android.com/reference/androidx/room/Embedded.

[14] Google Inc. Hardware abstraction layer (hal) overview. https://source.android.com/docs/core/architecture/hal.

[15] Silicon Lab. Ucos. https://www.silabs.com/developers/micrium.

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

[17] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures. *ACM Trans. Softw. Eng. Methodol.*, 33(1), November 2023.

[18] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, Heyuan Shi, and Yu Jiang. Effectively sanitizing embedded operating systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, New York, NY, USA, 2024. Association for Computing Machinery.

[19] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: packet sequence oriented fuzzing for protocol implementations. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[20] Dominik Maier and Fabian Toepfer. Bsod: Binary-only scalable fuzzing of device drivers. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 48–61, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Sanoop Mallissery and Yu-Sung Wu. Demystify the fuzzing methods: A comprehensive survey. *ACM Comput. Surv.*, 56(3), October 2023.

[22] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140*, 2018.

[23] Lucas McDonald, Muhammad Ijaz Ul Haq, and Ashley Barkworth. Survey of software fuzzing techniques.

[24] Anas Nashif. Zephyr is a new generation, scalable, optimized, secure RTOS, 2016. https://github.com/zephyrproject-rtos/zephyr.

[25] Henry Neugass, G Espin, Hidefume Nunoe, Ralph Thomas, and David Wilner. Vxworks: an interactive development environment and real-time kernel for gmicro. In *Eighth TRON Project Symposium*, pages 196–197. IEEE Computer Society, 1991.

[26] National Institute of Standards and Technology. National vulnerability database: Cve-2021-0673 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-0673.

[27] The Linux Kernel Organization. The linux kernel archives. https://kernel.org.

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

[29] Shiyan Peng, Yuan Zhang, Jiarun Dai, Yue Gu, Zhuoxiang Shen, Jingcheng Liu, Lin Wang, Yong Chen, Yu Qin, Lei Ai, Xianfeng Lu, and Min Yang. Applying fuzz driver generation to native c/c++ libraries of oem android framework: Obstacles and solutions. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2035–2040, New York, NY, USA, 2024. Association for Computing Machinery.

[30] LLVM Project. Llvm sanitizercoverage. https://clang.llvm.org/docs/SanitizerCoverage.htmll.

[31] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, San Diego, CA, August 2014. USENIX Association.

[32] Teresa Reidt. Several reasons for choosing embedded android. https://emteria.com/learn/embedded-operating-system.

[33] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[34] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 41(11):4563–4574, nov 2022.

[35] SimonKagstrom. Kcov. https://github.com/SimonKagstrom/kcov.

[36] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

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

[38] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. Property-based fuzzing for finding data manipulation errors in android apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1088–1100, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.

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

[41] Bernard Xiong and Man Jianting. RT-Thread is an open source IoT operating system., 2007. https://github.com/RT-Thread/rt-thread.

[42] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. Atlas: Automating cross-language fuzzing on android closed-source libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 350–362, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Karim Yaghmour. *Embedded android: Porting, extending, and customizing*. O'Reilly, 2017.