Rethinking Process Management for Interactive Mobile Systems

Jianwei Zheng[1,2], Zhenhua Li[1]✉, Feng Qian[3], Wei Liu[1], Hao Lin[1]
Yunhao Liu[1], Tianyin Xu[4], Nan Zhang[2], Ju Wang[2], Cang Zhang[2]
[1]Tsinghua University [2]Xiaomi Inc. [3]University of Southern California [4]UIUC

ABSTRACT

Modern mobile systems are featured by their increasing interactivity with users, which however is accompanied by a severe side effect—users constantly suffer from slow UI responsiveness (SUR). To date, the community have limited understandings of this issue for the challenges of comprehensively measuring SUR events on massive mobile devices. As a major Android phone vendor, in this paper we close the knowledge gap by conducting the first large-scale, long-term measurement study on SUR with 47M devices. Our study identifies the critical factors that lead to SUR from the perspectives of device, system, application, and app market. Most importantly, we note that the largest root cause lies in the wide existence of "hogging" apps, which persistently occupy an unreasonable amount of system resources by leveraging the optimistic design of Android process management. We have built on the insights to remodel Android process states by fully considering their time-sensitive transitions and the actual behaviors of processes, with remarkable real-world impact—the occurrences of SUR are reduced by 60%, together with 10.7% saving of battery consumption.

CCS CONCEPTS

• **Human-centered computing** → **Mobile phones**; *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering** → **Process management**.

KEYWORDS

Interactive Mobile Systems; Slow UI Responsiveness; Android Resource Management; App Behaviors.

ACM Reference Format:
Jianwei Zheng, Zhenhua Li, Feng Qian, Wei Liu, Hao Lin, Yunhao Liu, Tianyin Xu, Nan Zhang, Ju Wang, Cang Zhang. 2024. Rethinking Process Management for Interactive Mobile Systems. In *The 30th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '24), September 30-October 4, 2024, Washington D.C., DC, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3636534.3649357

ACM MobiCom '24, November 18-22, 2024, Washington D.C., DC, USA

ACM ISBN 979-8-4007-0489-5/24/09
https://doi.org/10.1145/3636534.3649357

1 INTRODUCTION

Rapid advancements in mobile hardware (e.g., high-resolution screens and high refresh rates) and network access (e.g., 5G and WiFi 6E) enable a wide range of highly interactive applications, such as real-time 3D games, UltraHD videos, AR/VR, and metaverse. Inevitably, these applications require considerable system resources to ensure smooth user interactions. Over the years, tremendous efforts have been made to optimize the user interface (UI) smoothness of mobile systems. For instance, Android 11 introduces *dynamic refreshing* [15] and *render-ahead pipeline* [12] to better support the emerging 120Hz refresh rate, while iOS 14 delivers the similar *ProMotion* [4] techniques. Despite these efforts, as a major Android phone vendor with hundreds of millions of users, we find that *slow UI responsiveness* (SUR) remains a major cause of users' complaints regarding the quality of experience (QoE).

To date, the community have very limited understandings on the prevalence, characteristics, and root causes of SUR, thus hindering practical solutions to address them. Existing studies are either based on controlled experiments at small scales [38], or confined to rare and extreme SUR cases like application-not-responding (ANR) and system-not-responding (SNR) [37]. As a matter of fact, Android does not even expose the SUR-related tracking interfaces to user-space applications [23], and these interfaces do not provide sufficient information to understand SUR in detail.

Understanding Android SUR at Scale. To close the knowledge gap, we conduct the first large-scale, long-term study on the SUR events occurring on massive Android devices. Taking the liberty of customizing the Android system (dubbed Android-MOD after customization) before releasing it to our customers, we build a continuous monitoring infrastructure by instrumenting a variety of system components/services to collect critical data (including the timestamp, foreground app name, CPU usage, memory consumption, I/O activity, etc.)

upon every SUR event. Meanwhile, we employ state-of-the-art optimizations (e.g., in-memory ring buffer and expensive synchronization avoidance) [42] to reduce the measurement overhead to a negligible level. We invited all our customers to participate in the study by installing Android-MOD on their phones; finally, 47M users opted in and gathered data for us during four months (Jun.–Sep. 2022), involving 48 different models of Android phones and 1M+ apps.

Our study shows that SUR occurs prevalently and frequently on all the studied devices; on average, as many as 7.97% of graphic frames rendered by an Android device are subject to SUR. While not all the SUR events are perceptible to all users, 73.65% of them have obvious impacts on almost all users, so in the remainder of this paper we concentrate on only these *obvious* SUR events. Counter-intuitively, we observe that better hardware cannot effectively reduce SUR events, and newer OSes do not alleviate the problem either—as Android upgrades from v10 to v12, the occurrences of SUR per phone increase by an average of 1.9%.

Delving deep, our analysis reveals that the majority (59%) of SUR events are attributed to system-wise resource contention, mostly caused by the wide existence of "hogging" apps; the remainder are owing to design/implementation issues of specific apps (e.g., overly complex UI layout and message processing in the main thread). Hogging apps keep alive in the background by registering the so-called "user-perceptible" [19] internal components (e.g., Activity, Foreground Service, and Broadcast Receiver), but do not provide *really perceptible* services (e.g., audible audio and active GPS navigation) to users. In fact, most hogging apps are non-malicious and many are rather popular; they keep alive to stealthily escalate user retention, enable in-app advertisement, facilitate data harvesting, and wake up other apps.

In comparison, we conduct small-scale measurements on mainstream iPhones, finding that they are significantly less prone to SUR (by ~10×) and iOS barely suffers from hogging apps. This is primarily ascribed to the stringent scrutiny policy of App Store [3]. In theory, hogging apps can also exploit the APIs (e.g., registering background audio or location services) that iOS provides for legitimate apps (e.g., music playback and navigation apps) to keep them alive [1, 7, 53]. However, they must undergo App Store's rigorous code inspection and manual verification, and thus can hardly get passed and released—hogging behavior can easily incur an app's rejection or removal from App Store, sometimes even with the developers' accounts being terminated.

Remodeling Android Process Management. Due to the decentralized nature of the Android ecosystem, App Store's early detection approach is not suited to Android for addressing hogging apps. People can acquire Android apps from diverse sources whose scrutiny policies differ greatly from each other. Therefore, in this paper we attempt to tackle the problem in a different stage (i.e., during an app's run time) from a novel perspective (i.e., inspecting the internal mechanisms of Android). In addition, we do not limit ourselves to pure system-wise solutions, but also collaborate with app developers to address app-specific SUR issues.

By carefully examining the various keep-alive strategies adopted by hogging apps, we note that they all leverage the optimistic design of Android process management, i.e., the Android system assumes all the registered "user-perceptible" internal components by a process to be really perceptible to its users. Concretely, a hogging process would register one or more such components that typically possess high priorities, so as to grant itself a high priority [20]; also, a hogging app oftentimes has multiple processes that can awake each other on demand. In this way, a hogging app can easily keep a large portion of its processes alive in the background, even when it is not really being used. With the insight, we strive to reshape the design of Android process management to identify hogging apps effectively and efficiently, as well as to safely reclaim their occupied resources.

To achieve this goal, at first glance it appears that we only need to insert a "hogging" state to the Android process model based on the hindsight acquired above. In practice, however, it is challenging to translate hindsight to foresight. First, we do not know the actual behaviors of existing processes, e.g., all processes in the "foreground" state are simply classified as "user-perceptible" by Android regardless of whether they are actually user perceptible. Second, deterministic state transitions in the current Android process model cannot describe the dynamic behaviors of certain processes, e.g., a seemingly idle process at the moment may perform periodic tasks in the future (rather than keep hogging system resources).

To practically address the challenges, we comprehensively profile real-world Android process states and their transition patterns on the studied 47M devices. Based on the collective wisdom, we build a time-inhomogeneous Hidden Markov model [41] (TIHMM) to describe the complex state transitions (in particular the dynamic behaviors of certain processes) in a time-sensitive manner. To maximize the accuracy, TIHMM adaptively inserts hidden states among existing Android process states and our defined "hogging" state, while constantly updating state transition probabilities over time.

To capture processes' actual behaviors to inform TIHMM, we develop a uniform authentic sensing layer in Android, which efficiently monitors their usages of user-perceptible hardware (such as audio, GPS, Bluetooth, and network) via eBPF-based system probes [9]. When TIHMM determines a hogging app, we do not simply terminate it at once. Instead, to mitigate the side effect of improper terminations of certain working processes (originating from the foresight errors of TIHMM), we leverage the kernel's process freezing

capability [50] to only suspend the execution of suspicious processes rather than killing them, thus preserving their execution contexts for probable reuse in the near future.

Real-World Performance and Robustness. We have implemented the innovative design in Android-MOD and deployed it on 28M Android devices that opted in for our evaluation from Jan. 1st 2023 to Feb. 28th 2023. Also, for app-specific design/implementation issues, we have reported them to the corresponding app developers, and 278 out of all the 415 issues have been confirmed and addressed.

With these efforts, we have substantially reduced the occurrences of SUR events by an average of 60% (ranging from 36.51% to 82.12%) for each device model. Besides, through a meticulous A/B test [35], we confirm that the vast majority (66.44%) of SUR reductions come from the innovative design (TIHMM) in Android-MOD. Moreover, due to the effective throttling of resource usages from hogging apps, the energy consumption has been reduced by an average of 10.69% (ranging from 25.07% to 1.18%) for each device model.

It is worth noting that our design targets non-malicious hogging apps that *leverage* the inherent design of Android to gain resource advantages, as opposed to malicious apps that *exploit* system vulnerabilities for illegal purposes. We present an in-depth analysis of TIHMM to investigate whether the developers of a non-malicious hogging app will be able to avoid our detection by adapting their app's behaviors. We show that it is rather difficult for a hogging app to circumvent our detection without user consent or exploiting system vulnerabilities, so its developers are left with a significantly weaker range of adapting capabilities. For those malicious hogging apps, mainstream Android app markets can usually recognize them and thus fall outside our concerns.

Summary. We make the following contributions:

- We conduct the first large-scale and in-depth study on the slow UI responsiveness (SUR) problem of Android, which is pivotal to the QoE of interactive mobile systems but never investigated systematically. We present its prevalence, frequency, and multi-faceted characteristics.
- We locate the major root cause of SUR to be the resource abuse of hogging apps, which leverage the optimistic design of Android process management to keep alive.
- We refactor the process management model of Android to effectively identify and suppress hogging apps. The design is implemented and deployed at scale, showing that it can significantly reduce both SUR and energy consumption.
- We study SUR of iOS and find it fairly mild. The reason mostly lies in the closed ecosystem of Apple, esp., the sole authority and stringent scrutiny of App Store, which is thus inapplicable to decentralized mobile OSes like Android, Fuchsia, KataOS, HarmonyOS, and Ubuntu Touch.

The relevant code and data have been released at https://Android-SUR.github.io to benefit the community.

2 BACKGROUND AND RELATED WORK

In this section, we introduce the pipeline of frame rendering and the process management model in Android to show how SUR happens. Also, we review related work on diagnosing and mitigating UI responsiveness issues of mobile systems.

2.1 Android Frame Rendering Pipeline

Rendering a frame in Android involves five steps:

- **Input event dispatch**. Frame rendering is usually initiated when users interact with the touchscreen. The system service, `WindowManagerService`, captures these interactions, translates them into input events (encoded as messages), and dispatches them to the relevant app.
- **UI thread processing**. Upon receiving messages, the app's main thread (i.e., the `UI` thread) calculates the size and relative position of each involved UI element.
- **RenderThread execution**. Then, the `RenderThread` sends rendering commands to the GPU for converting the properties of UI components to bitmap pixels.
- **SurfaceFlinger composition**. Subsequently, the system service, `SurfaceFlinger`, takes charge of UI composition by obtaining the bitmap, superimposing the bitmap with other visual components, and inscribing the final composite image into a frame buffer.
- **Hardware display**. Finally, the `Hardware Composer` reads the final composited image from the frame buffer and displays it on the screen.

Each of the abovementioned components communicates with the subsequent one via the message mechanism [2] and the `Binder` inter-process communication (IPC) mechanism [5]. These components collaboratively ensure that user interactions are translated smoothly to visual updates on the screen. Unfortunately, due to the intricate nature of Android's multi-step rendering process, any inefficiencies in each component can slow down frame rendering. As for a 60 FPS screen, if a frame takes more than 16.67 ms to render, this is regarded as a slow UI responsiveness (SUR) event [26].

2.2 Android Process Management Model

The process management model manages the lifecycle of each app process and decides which process(es) should be kept alive or killed when system resources (such as CPU scheduling quota, available memory capacity, and I/O bandwidth) become constrained. As shown in Figure 1, Android divides the lifecycle of a process into five states: `Foreground`, `Visible`, `Service`, `Background`, and `Empty`. At the top of

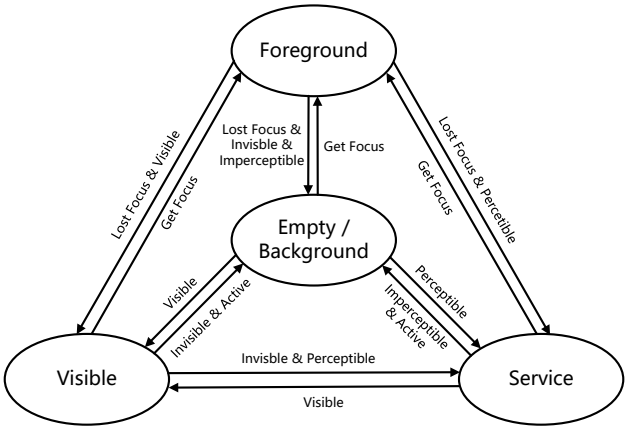


Figure 1: The state machine that models the process management in Android.

Figure 1, a process enters the `Foreground` state when running any of the following components: a visible Activity that users are focusing on, a "user-perceptible" Service bound to a visible Activity, or an active Broadcast Receiver.

When the user stops interacting with the visible Activity running in a `Foreground` process, the process transitions to the `Visible` state. Further, when the visible Activity terminates but there is still a "user-perceptible" Service (e.g., audio playback in the background) running in a `Visible` process, the process transitions to the `Service` state. Sometime later, when the "user-perceptible" Service stops but there is still an active invisible Activity or imperceptible Service running in a `Service` process, the process goes to the `Background` state. At last, when there is no active component running in a process, the process goes to the `Empty` state. Note that there exist other transition paths among the five states, as illustrated in Figure 1.

In any state, an app process can be killed to reclaim system resources. For an `Empty` process (of course with a very low priority), once the available system memory falls below a pre-defined threshold, it will be killed to release some memory capacity. When all the `Empty` processes have been killed but there is still insufficient memory capacity released, Android will start to kill `Background` processes. Likewise, when all the `Background` processes have been killed but the released memory capacity is still deficient, Android will start to kill `Service` processes. In extreme cases, even `Visible` and `Foreground` processes can be killed by Android.

It is easy to see that once there are numerous hogging apps living in the system, the other apps (i.e., benign apps) will be subject to frequent resource deficiencies which would in turn trigger frequent resource reclaim operations. Since resource reclaim operations are usually rather time-consuming, their frequent occurrences will remarkably aggravate SUR.

2.3 Related work

Diagnosis Approaches. Considerable work has been conducted to diagnose performance issues of mobile apps. First, static code analysis is utilized to pinpoint buggy code patterns that damage the system and/or app performance, such as malicious behaviors [51], resource leakage [61], layout defects [11, 62], improper Activity lifecycle patterns [10], and UI thread blockages [46]. Given the limitation of static analysis in handling dynamically loaded code in runtime, researchers resort to dynamic analysis-based approaches, such as code instrumentation [52] and resource amplification [58], to study runtime behaviors and identify blocking or computation-intensive operations.

Moreover, several diagnostic tools help developers analyze the root causes of mobile systems' performance issues. Logging tools [24, 63] allow streamlined insertion of log statements for fine-grained monitoring, but lack capabilities for dynamic interaction tracking. Tracing tools [22, 25, 30] offer dynamic profiling across various performance metrics (e.g., CPU/memory utilization). Nonetheless, due to mobile devices' interactive nature and resource limitations, these tools can significantly slow down apps in production [42].

To minimize overhead, bytecode tracing tools have been introduced for Android. Nanoscope [56] embeds tracepoints within the Android virtual machine's source code at the start and end of ArtMethod execution. Hubble [42] instruments every non-inlined bytecode method's entry and exit by leveraging the runtime environment to automatically embed its tracing logic into the compiled binary or interpreted logic. However, these tools need either adjustments to the ART source code or modifications to the compiler and interpreter.

Mitigation Approaches. Various optimizations have been made to enhance the responsiveness of mobile systems. First, some researchers [38–40, 45] leverage code refactoring to improve the performance of mobile apps. For example, Lin et al. [40] propose two tools ASYNCHRONIZER and ASYNCDROID to locate and refactor long-running operations for real concurrent code execution and higher responsiveness. Besides, quite a few solutions [6, 8, 28, 33, 36] use computation offloading that transfers computation-intensive tasks to external resource-rich computing units, like cloud servers.

Some people [55, 57] adapt the processor to software for accelerating the execution of time-consuming tasks in mobile devices. For instance, ACCELDROID [57] accelerates the execution of bytecode on the hardware/software co-designed processor of Android by only translating bytecode once. Others [32, 37, 44] optimize I/O operations to improve responsiveness. To address Android's well-known defect of Write Amplification Mitigation (WAM) that could trigger ANR and SNR, Jeong et al. [32] prioritize the quasi-asynchronous I/O

Table 1: Hardware configurations of our studied 48 phone models, generally ordered from low-end to high-end. The *Version* and *Users* columns correspond to Android version and user percentage, respectively.

Model	CPU	Memory	Storage	Version	Users	Model	CPU	Memory	Storage	Version	Users	Model	CPU	Memory	Storage	Version	Users
1	2.0 GHz	3 GB	32 GB	10.0	1.73%	17	2.4 GHz	6 GB	128 GB	11.0	3.10%	33	2.84 GHz	12 GB	256 GB	10.0	1.76%
2	2.0 GHz	3 GB	64 GB	11.0	1.23%	18	2.4 GHz	8 GB	128 GB	12.0	2.87%	34	2.84 GHz	12 GB	256 GB	11.0	1.61%
3	2.0 GHz	4 GB	32 GB	11.0	1.87%	19	2.84 GHz	6 GB	64 GB	10.0	2.12%	35	2.84 GHz	12 GB	256 GB	10.0	2.50%
4	2.0 GHz	4 GB	64 GB	12.0	2.16%	20	2.84 GHz	6 GB	128 GB	12.0	3.24%	36	2.84 GHz	12 GB	512 GB	12.0	1.79%
5	2.0 GHz	4 GB	64 GB	10.0	0.84%	21	2.84 GHz	8 GB	64 GB	11.0	1.98%	37	2.96 GHz	8 GB	128 GB	11.0	0.92%
6	2.0 GHz	4 GB	128 GB	11.0	0.93%	22	2.84 GHz	8 GB	128 GB	12.0	3.17%	38	2.96 GHz	8 GB	256 GB	10.0	1.13%
7	2.0 GHz	6 GB	64 GB	12.0	1.12%	23	2.84 GHz	8 GB	128 GB	10.0	4.21%	39	2.96 GHz	12 GB	128 GB	12.0	0.99%
8	2.0 GHz	6 GB	128 GB	10.0	1.79%	24	2.84 GHz	8 GB	128 GB	11.0	5.27%	40	2.96 GHz	12 GB	256 GB	11.0	1.27%
9	2.0 GHz	8 GB	128 GB	11.0	2.10%	25	2.84 GHz	8 GB	128 GB	11.0	2.01%	41	3.0 GHz	8 GB	128 GB	12.0	2.70%
10	2.0 GHz	8 GB	256 GB	12.0	1.87%	26	2.84 GHz	8 GB	256 GB	11.0	3.76%	42	3.0 GHz	8 GB	256 GB	11.0	2.28%
11	2.3 GHz	2 GB	32 GB	11.0	1.46%	27	2.84 GHz	8 GB	256 GB	11.0	3.78%	43	3.0 GHz	12 GB	128 GB	12.0	1.16%
12	2.3 GHz	2 GB	64 GB	11.0	1.39%	28	2.84 GHz	8 GB	256 GB	11.0	3.27%	44	3.0 GHz	12 GB	256 GB	12.0	0.78%
13	2.3 GHz	3 GB	32 GB	10.0	2.01%	29	2.84 GHz	8 GB	256 GB	12.0	3.95%	45	3.2 GHz	8 GB	128 GB	12.0	1.23%
14	2.3 GHz	3 GB	64 GB	12.0	1.85%	30	2.84 GHz	8 GB	512 GB	10.0	2.27%	46	3.2 GHz	8 GB	256 GB	11.0	1.84%
15	2.3 GHz	8 GB	128 GB	12.0	2.45%	31	2.84 GHz	12 GB	128 GB	11.0	2.01%	47	3.2 GHz	12 GB	128 GB	12.0	1.17%
16	2.3 GHz	8 GB	256 GB	12.0	2.70%	32	2.84 GHz	12 GB	128 GB	12.0	1.87%	48	3.2 GHz	12 GB	256 GB	12.0	0.49%

operations while Li et al. [37] strike a balance between real-time and lazy WAM mechanisms.

Comparison with our work. As a large-scale Android phone vendor, we explore the unique opportunity to instrument various system components/services for in-depth performance diagnosis of mobile apps. Somewhat like Hubble [42], we also take measures to essentially reduce the instrumentation overhead. To mitigate the discovered performance issues, we take a more fundamental approach of refactoring the basic model of Android process management, and validate its efficacy through massive deployments.

3 STUDY METHODOLOGY

3.1 Continuous Monitoring Infrastructure

There have been several methods for monitoring SUR-related events on Android, typically based on call stack tracing [37, 38]. These methods require pausing the target app to collect the entire call stack information, and thus are confined to rare and extreme SUR cases like application-not-responding (ANR) and system-not-responding (SNR). For our large-scale, online, and continuous monitoring, such call stack-based methods are not suited since common SUR events could be transient (e.g., lasting for only tens of milliseconds) and frequent (e.g., affecting up to 14% of the rendered frames), which could incur tremendous overhead.

To address the problem, we design a lightweight continuous monitoring infrastructure for capturing SUR events by taking the liberty of customizing the Android system (the customized Android is dubbed as Android-MOD). Note that we develop the lightweight monitoring infrastructure inside the kernel and Android framework, which is passively triggered to collect data only when SUR events occur. Specifically, our method consists of the following three phases.

- **SUR event identification.** In order to continuously identify potential SUR events, we use the `Choreographer` API of Android to retrieve the timing information from Android's display subsystem in an event-driven manner. Such information contains the timestamps about when a VSync [27] arrives and when a frame is rendered. Thus, we can calculate the rendering delay of each frame as well as the overall frame drop rate. We then judge whether an *obvious* SUR event occurs based on an empirical threshold of the rendering delay (50 ms, detailed below).
- **System service instrumentation.** Once we detect an SUR event, we wish to distinguish whether it is caused by system-wise factors or inefficient UI design of the app itself. To this end, we instrument the Android services related to frame rendering like `ActivityManagerService`, so that we can monitor the lock contention among the processes that request the services. We also record the duration of the app's UI thread on key states (i.e., running, runnable, and sleep) to discern the app's own inefficiency.
- **Cross-layer in-situ information tracing.** When we determine that the SUR event derives from system-wise factors (in particular the system services), we further capture fine-grained in-situ information about the SUR event from both Android's framework and the kernel. Specifically, we record the dispatched message from the `Looper` thread [21], and leverage the `atrace` [47] utility to obtain the system's detailed information (e.g., the current process list, binder transactions, thermal status code, as well as the utilization of CPU, memory, and I/O). We also add a new logging module to the kernel to capture I/O delay, memory swapping/reclaiming delay, and the duration of all threads in different thread states.

In order to determine whether an SUR event is *obvious* to users, we conducted a user study by recruiting 30 volunteers of different genders and ages (ranging from 20 to 55). We inject SUR events with various frame drop rates (ranging from 1/60 to 10/60) to 60-FPS phones by modifying the Android system, and ask the volunteers to interact with our modified Android system (without third-party apps installed to avoid potential interference). Because in each test only one SUR event is injected and the frames are always consecutively

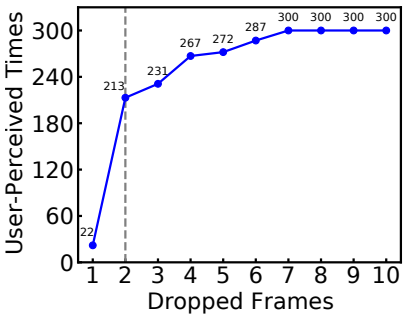


Figure 2: User-perceived times of consecutively dropped frames.

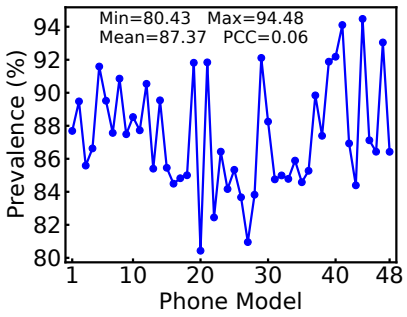


Figure 3: Prevalence of SUR events on each model of phones.

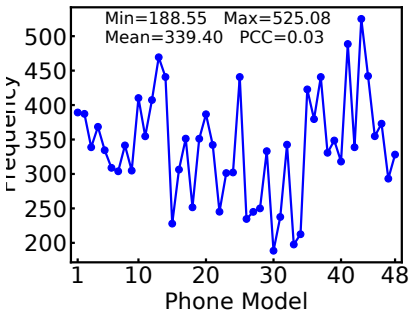


Figure 4: Frequency of SUR events on each model of phones.

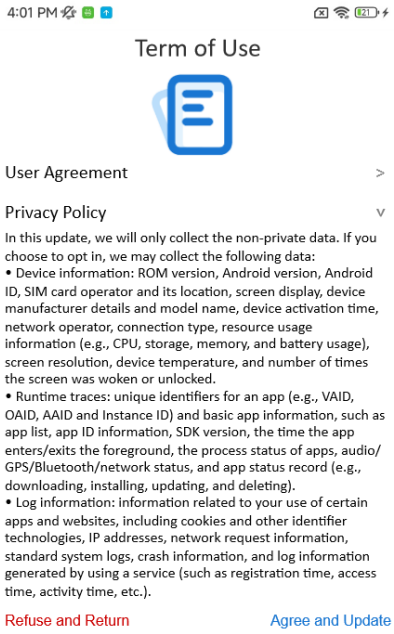


Figure 5: The screenshot of user consent form.

dropped, the volunteers only need to recognize whether there is an occurrence of SUR events when perceiving them. We repeat the experiment ten times (i.e., 300 tests in total). As shown in Figure 2, when two frames are consecutively dropped (i.e., the rendering delay is ~50 ms, covering the total rendering time of three consecutive frames), the vast majority (71%) of volunteers could perceive it. Thus, we set the threshold for determining an *obvious* SUR event as 50 ms.

In addition, we also leverage state-of-the-art optimization techniques [42] to reduce the overhead of our monitoring infrastructure. For example, we avoid the usage of expensive synchronization primitives to reduce the communication overhead between the tracing thread and the traced UI thread. Also, we store all the tracing/logging data in an in-memory ring buffer, which is flushed only when an SUR event is detected. Moreover, we send the results to our analysis server in a batched manner, so as to minimize the network traffic usage. Even for a low-end Android phone, Android-MOD only incurs <1.8% CPU utilization, <10 MB of memory usage, <5 MB of storage space, and <20 MB network usage per day.

3.2 Large-Scale Deployment

As a major Android phone vendor, in Jun. 2022 we invited all our customers to participate in our measurement study by installing Android-MOD on their phones. The installation is a lightweight update that will not impact their installed apps, existing data, and OS version. Finally, 47,387,684 users opted in and contributed data over a four-month period (Jun.-Sep. 2022), involving a wide variety of phone models as listed in Table 1 (all their CPUs own eight cores).

Ethical Concerns. This study is conducted under a well-established IRB. All the participants opted in with an informed consent, and no personally identifiable information (e.g., phone number, IMEI, and IMSI) was ever gathered. Figure 5 shows the update UI of Android-MOD, where the list of data to be collected is explicitly displayed to participators.

4 MEASUREMENT RESULTS

4.1 General Statistics

With the four-month crowd-sourcing from 47,387,684 Android-MOD user devices with 48 different phone models (listed in Table 1) and 1,037,261 apps, we record the system-level traces with regard to an average of 13,029,922,803 SUR events per day. To facilitate longitudinal analysis, we also refer to our measurement reports in recent years when necessary. To the best of our knowledge, this is so far the largest dataset regarding SUR events in the wild.

First, we are concerned with the *prevalence* of SUR events which represents the average proportion of devices experiencing at least one SUR event per day. As shown in Figure 3, the prevalence of SUR events on different phone models ranges from 80.42% to 97.73% with an average of 86.95%. Then, we analyze the *frequency* of SUR events that denotes the average number of SUR events experienced per phone every day. Figure 4 depicts that on average as many as 338.28

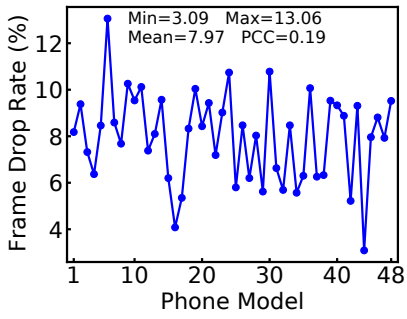


Figure 6: Frame drop rate of each phone model.

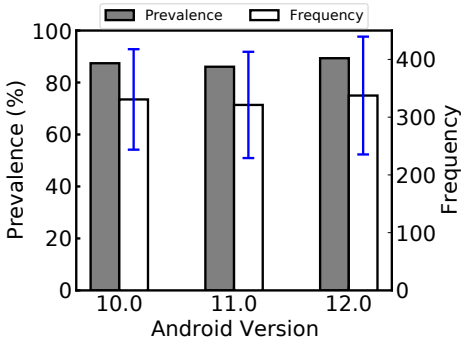


Figure 7: SUR prevalence and frequency of each Android version.

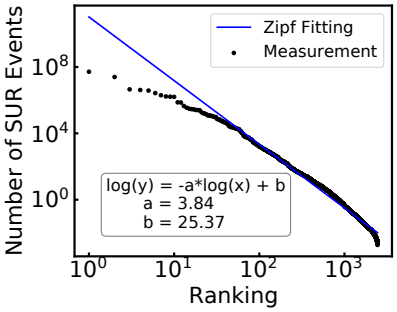


Figure 8: Ranking of apps by their number of SUR events per day.

SUR events occur on a phone per day, and the average number of SUR events happening to a specific model per day varies from 179.59 to 554.68. In addition, we are also concerned with the *frame drop rate* caused by SUR events per phone for each model. As demonstrated in Figure 6, on average, as many as 7.91% of the graphic frames rendered by an Android device suffer from SUR, and the frame drop rate of specific models varies from 3.09% to 14.12%.

4.2 Frustrating Android Ecosystem

We delve further into the internals of Android ecosystem with respect to the hardware, system, and applications.

Hardware Configurations. Common sense suggests that high-end smartphones should experience fewer SUR events. However, our measurement results suggest that better hardware configurations cannot effectively alleviate SUR events. We conduct the Pearson correlation analysis [54] to quantitatively characterize the correlation between the prevalence/frequency of SUR events and hardware configurations. As shown in Figure 3 and Figure 4, the Pearson correlation coefficient (PCC) of prevalence-hardware and frequency-hardware is 0.06 and 0.03, respectively, which indicates that simply enhancing hardware configuration cannot effectively reduce SUR events. Specifically, among the 48 models of phones we study, the ten oldest models (Model 1-10, released between Aug. 2019 and Jun. 2020) and the ten latest models (Model 39-48, released between Jul. 2020 and Dec. 2021) suffer almost the same number of SUR events per phone.

Android Versions. The Android versions of the studied 48 phone models consist of 10.0, 11.0, and 12.0, which were released in Sep. 2019, Sep. 2020, and Oct. 2021, respectively. As Android evolves from version 10.0 to 12.0, tremendous efforts have been made to optimize the responsiveness of the Android framework and OS kernel [13, 14, 29]. Therefore, smartphones with higher Android versions are expected to suffer fewer SUR events. As illustrated in Figure 7, Android 11.0 devices exhibit a 2.89% reduction in SUR events compared to their Android 10.0 counterparts. However, Android 12.0 devices have a 1.98% higher probability of suffering SUR events, and experience more 1.90% SUR events per phone. We primarily attribute this to the better stability and robustness of Android 10.0 and 11.0, as Android 12.0 was still undergoing constant patches and need apps to adapt to the newly-provided APIs [13] during our measurement.

Table 2: Top-10 apps ordered by the frequency (or simply likelihood) of SUR events after normalization.

ID	Alias	Category	Users	Time(s)	Likelihood
1	WeChat	Instant Messaging	47M	4344	10.81
2	Douyin	Video Streaming	40M	7547	10.21
3	Mobile QQ	Instant Messaging	24M	1566	6.93
4	Kwai	Video Streaming	16M	6666	6.80
5	Pinduoduo	E-commerce	41M	803	6.50
6	Taobao	E-commerce	38M	852	6.38
7	Alipay	Mobile Payment	34M	409	5.98
8	Toutiao	News Browsing	19M	4981	5.95
9	Jindong	E-commerce	19M	931	5.91
10	Bilibili	Video Streaming	10M	5975	5.55

Mobile Applications. Our analysis identifies a daily count of 13,029,922,803 SUR events, involving 1,037,261 Android apps in China. Upon ranking these apps based on their daily SUR event occurrences (in descending order), we observe a nearly-Zipf skewed distribution as depicted in Figure 8. Among all SUR events, 16.80% are attributed to the top-10 apps (<0.001%) in Table 2, while the remaining (83.20%) belong to the vast majority (>99.99%) of apps in the "long tail".

A detailed examination of the top-10 apps in Table 2 reveals their diverse functions: two for instant messaging, three for video streaming, three for e-commerce, and the last two are employed for mobile payment and news browsing. Our further analysis reveals that these apps exert substantial demands on system resources to manage their hefty workloads which encompass high-quality media content streaming, embedded WebView components [16], and complex UI functionalities. As depicted in Figure 9 and Figure 10, the memory consumption of these apps when running in the foreground and background shows a consistent rise. As of June 2022, the

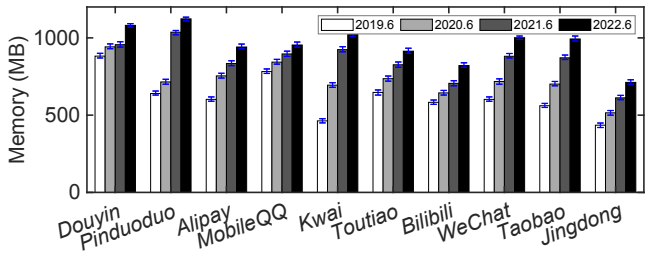


Figure 9: Memory occupation of the top-10 apps running in the foreground from Jun. 2019 to Jun. 2022.

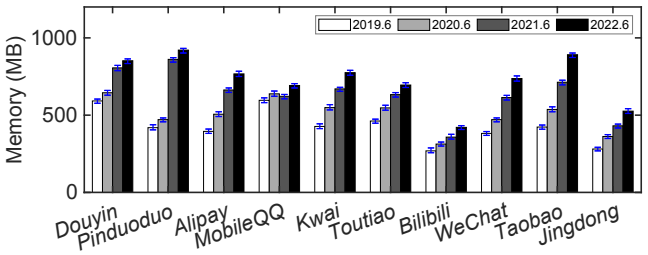


Figure 10: Memory occupation of the top-10 apps running in the background from Jun. 2019 to Jun. 2022.

average memory usage of these apps is recorded at 956.1 MB and 727.7 MB in the foreground and background, with an average annual growth rate of 19.78% and 15.58%, respectively.

4.3 Root Causes Analysis

App and system developers typically conduct analysis of corresponding logs by hand to identify the root cause of an individual SUR event. However, such manual analysis is not scalable and can not obtain representative conclusions. To address this, we propose a two-phase semi-automated analysis paradigm that combines online macro-statistical analysis and offline micro-reproduction analysis.

Online Macro-statistical Analysis. After cleaning the fine-grained cross-layer logs collected in Section 3.1, we aggregate these logs into feature vectors (V) upon which we conduct a clustering analysis to pinpoint the corresponding reasons. If the feature vector (V_i) of an SUR event i is similar to that (V_j) of another SUR event j, SUR events i and j will be classified into the same cluster. Specifically, the similarity between V_i and V_j (denoted as $S(i, j)$) is quantified through an off-the-shelf similarity metric named Jaccard index[31] which is particularly useful and efficient for clustering techniques that work with text data. The similarity score, $S(i, j)$, is calculated as $J(V_i, V_j) = \frac{|V_i \cap V_j|}{|V_i|+|V_j|-|V_i \cap V_j|}$. The SUR events i and j will be classified into the same cluster if $S(i, j)$ surpasses a threshold, which is empirically set to 0.95 based on our manual inspection of representative SUR samples.

Through this clustering process, we preliminarily acquire 1,310 clusters, among which three dominant clusters cover the majority (59%) of SUR logs. Then, we manually analyze the unbiased samples in each dominant cluster, and discover three major reasons of SUR events: long CPU scheduling delay (20.98%), slow I/O transactions (11.32%), and insufficient memory (26.70%). After manually checking the remaining non-dominant clusters, we find that these clusters primarily consist of cases caused by app-specific design defects (41%), which are merge into a single large cluster. Concrete characteristics of these reasons are detailed as follows.

First, we observe that SUR events caused by long CPU scheduling delay often arise when CPUs are occupied by other threads of background apps that should be of lower execution priority. Moreover, we find in 23% of these cases, only the less powerful `little` cores of the ARM CPUs are scheduled to the foreground apps' rendering threads and SurfaceFlinger, whereas the performant `big` cores are fully occupied by other threads of background apps.

Second, SUR events caused by slow I/O transactions are typically attributable to heightened resource contention when multiple processes concurrently strive for I/O operations. This occurs as all I/O requests from the upper framework layer are queued and handled in a hardware dispatch queue of I/O scheduler upon entering the kernel layer, leading to the I/O resource contention across different processes. In particularly, when background processes initiate a substantial number of read and write disk operations, they fully occupy the system's I/O resources, leading to the head-of-line (HoL) blocking of the foreground process's I/O requests.

Third, regarding insufficient memory, we observe that most of the related SUR events result from a usually time-consuming mechanism triggered by low available memory — `Low Memory Killer Daemon` (`lmkd`) [18] of Android, which aims to kill low-priority processes for freeing up memory in the userspace. Our analysis of the relevant logs reveals that low-priority processes in the `Empty` state, intended to be terminated by `lmkd`, are frequently reactivated by either their parent processes or other high-priority processes that "depend on" them, leading to a counterproductive cycle.

In terms of app-specific defects, our analysis identifies two primary causes: 1) Prolonged `doFrame` execution time on the UI thread due to overly complex UI components. When the UI design of apps is overly intricate (e.g., deep view hierarchies, heavy use of sophisticated animations, and resource-intensive rendering of elements), it keeps the UI thread running for extended periods. 2) Frequent garbage collection causing the suspension and blockage of the UI thread. In instances where an application creates a multitude of short-lived objects or suffers from memory leaks, it triggers the time-consuming garbage collection more frequently, subsequently causing the UI thread to be blocked and await.

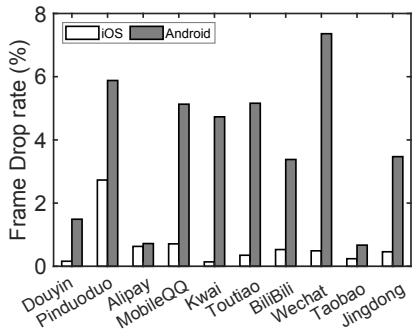


Figure 11: Frame drop rate of Top-10 apps in iOS and Android devices.

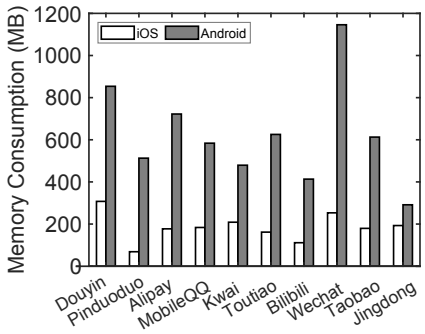


Figure 12: Memory consumption of Top-10 apps in iOS and Android.

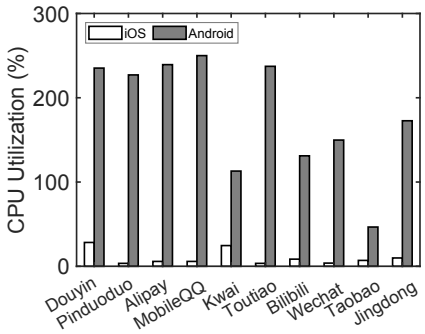


Figure 13: CPU occupation of Top-10 apps in iOS and Android devices.

Offline Micro-reproduction Analysis. To uncover the root cause of resource under-provisioning and contention, we conduct benchmark experiments to reproduce SUR events and record more detailed traces through the Android Debug Bridge (ADB) [17]. First, we enhance the `ActivityManager` (the system service for managing app activities in Android) for adding more information (e.g., states and lifespans of processes) to the output of `dumpsys` (a tool that provides system information), along with the existing details like the resource consumption. Our analysis involves a meticulous manual examination of the enhanced diagnostic outputs of `dumpsys`, coupled with an in-depth Timing Alignment Analysis which includes the strategic selection of SUR event window and a correlation analysis.

Specifically, once an SUR event is identified, we define a time window around it including a pre-event period (-1s), the event duration, and a post-event period (+1s). The choice of one-second time windows before and after each SUR event is inspired by Android-MOD's ANR trigger time which is set at two seconds. This event windowing strategy is helpful in isolating the temporal contexts pertinent to each SUR event, thereby facilitating a targeted analysis of process states in relation to SUR occurrences. After that, we employ the Pearson Correlation Coefficient (PCC) [60] to quantitatively assess the correlation between the occurrence time of SUR events and the lifespan of low-priority processes. The value of PCC turns out to be 0.91 which indicates a conspicuous correlation between SUR events and the persistent survival of numerous low-priority processes of hogging apps that should have been terminated to release system resources.

Delving deep, we find that such a strong correlation is rooted in these hogging apps' greedy strategy for keeping themselves alive. Unlike benign apps that pause/terminate operations and release resources when not needed according to the Android lifecycle policy, hogging apps exploit the optimistic design of Android process management (§2.2) and utilize various tricks to keep their processes alive. This easily trigger system resource contention and under-provisioning, leading to fast battery drain and frequent SUR events.

In our further investigation, for hogging apps in China, we identify two main exploited patterns for keeping their processes alive in the background. The first one is the abuse of foreground services which enable apps to perform continuous operations even when they are not perceivable by the user. While essential for functionalities requiring persistent running (e.g., audio services for music playback and GPS services for navigation), we observe that they are susceptible to be abused by hogging app developers. One typical example is the abuse of audio services where apps exploit the audio service for tasks unrelated to sound processing, such as sustaining app activities through silent audio playback.

The second exploited pattern is dual-process co-awakening behaviors through process binding. Since Android independently manages the lifecycle of each process of apps (see §2.2), one app can lock a file in one of its process (P_1), and meanwhile monitor the status of P_1 by attempting to lock the same file in the other in-app process (P_2). If P_1 is killed, P_2 will immediately lock the file and revive P_1, and vice versa.

Generalizability of Findings. To find whether our measurement results generalize to other countries where Google Play Store is not blocked, we conduct a benchmark experiment in the global context. Specifically, we select the top-100 hogging apps identified in our measurement study in China. We find that 56 of these top-100 hogging apps are also available on Google Play Store. More importantly, after applying our analysis methods in §4.3 to them, we observe that their hogging behaviors are consistent with those observed in China (i.e., keeping alive by abusing foreground services and dual-process co-awakening). This indicates that hogging apps cannot be fully eliminated despite Google's sophisticated vetting. Thus, we believe that the existence of hogging apps is a global problem, although the prevalence may vary with the policies in different regions.

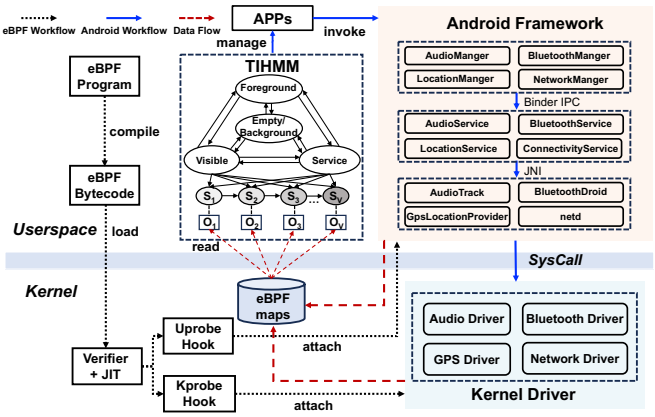


Figure 14: Architectural overview of our solution where S and O denote the hogging state and the corresponding observation, respectively.

4.4 Comparative Study with iOS

To discover the discrepancy of SUR events on iOS and Android, we conduct a comparative study. Specifically, we first install the iOS-version and Android-version of top-10 apps (listed in Table 2) that were released around the same timeframe and have similar functionalities on five mainstream iPhones (equipped with 4 GB RAM, 128 GB ROM, and iOS 15) and five Android devices (equipped with 4 GB RAM, 128 GB ROM, and Android 11). By leveraging the off-the-shell analysis tool, PerfDog [59], we measure the frame drop rate of these devices when running the above apps under the same workload. As depicted in Figure 11, we notice a discrepancy between Android devices and iPhones that the former tend to exhibit a substantially higher frame drop rate than the latter (by $\sim 10\times$). Particularly, the frame drop rate of Kwai in Android devices is 33.78 times higher than in iOS devices.

Delving deeper into the resource consumption of apps on iOS and Android in Figure 12 and Figure 13, we observe that these apps running on Android consistently exhibit much higher resource occupation compared to their counterparts on iOS. This discrepancy is primarily attributed to the stringent scrutiny policy of App Store [3]. In fact, hogging apps can also potentially exploit the APIs (e.g., registering background audio or location services) that iOS provides for legitimate apps (e.g., music playback and navigation apps) to keep them alive [1, 7, 53]. However, before being admitted to the Apple Store, they must undergo App Store's rigorous code inspection and manual verification. Thus, the exhibition of repeated resource-hogging behaviors are grounds for rejection or removal from the Apple Store, even with developer accounts being terminated.

5 REMODELING PROCESS MANAGEMENT

In §4.3, we reveal that given the deceptive behaviors of hogging apps for keeping alive, the existing optimistic process management model (§2.2) fails to effectively manage the real-world Android processes due to its deterministic state transitions and the incapability of sensing the actual behaviors of processes. To address this, our key insight is that the state changes during the lifecycle of the process can be deemed as a time series. Thus, we leverage the time-inhomogeneous Hidden Markov model [41] (TIHMM) to describe the real-world process states and their dynamic state transitions in a time-sensitive manner based on our dataset of the studied 47M devices in §4. Second, to efficiently sense the actual behaviors of processes to inform TIHMM, we develop a uniform authentic sensing layer in Android which continuously monitors the actual hardwareusage through eBPF-based probes. The architectural overview is demonstrated in Figure 14.

5.1 TIHMM-based Process State Modeling

At first glance, the state transitions of the process could be modeled as a Hidden Markov process [49]. Unfortunately, the traditional Hidden Markov process can only model a stationary process where the state transition probability is deterministic. To address this, we notice that changes in the process state can be regarded as a time series. Thus, we formalize the entire process state model as a `time-inhomogeneous` Hidden Markov process [41] (TIHMM), which is robust to the dynamic probability of the state transition over time.

We then attempt to incorporate the TIHMM into Android. To this end, we need to insert a new "hogging" state (S in Figure 14) to the original state machine of Android's process management, so as to describe the hogging processes. However, simply adding a "hogging" state is insufficient to accurately depict the varying degrees of the process's hogging behaviors (influenced by different levels of resource usage and the deceptive actions of the process itself). Therefore, based on the in-situ system observations (O in Figure 14), we adaptively insert one or multiple sub-hogging hidden states into existing process states on demand.

At this point, we can formalize all of these hidden state spaces as S_N. Meanwhile, the observation set O_M includes the system resource (CPU, I/O, and memory) usage and the perceptual service behaviors with audio, GPS, Bluetooth, and network. For continuous observations (like CPU utilization), we discretize them by segmenting [min, max] values into equal-length intervals. Apart from S_N and O_M, the TIHMM is defined by a time-inhomogeneous state transition probability matrix $Q(t)$ where element $q_{i,j}(t)$ indicates the probability of transitioning from state s_i to s_j at time t, the emission model $P(o|s)$ denoting the probability of observing o given state s, and an initial state probability distribution π.

When apps run in the background, we can get a fully observed TIHMM via the eBPF-based Uniform Authentic Sensing as described in §5.2, which contains several sequences

of information over time: the underlying state transition time points $T = (t_0, t_1, t_2, ..., t_V)$, the corresponding states $S = (s(t_0), ..., s(t_V))$ of the hidden Markov chain, and the observed data $O = (o_0, o_1, ..., o_V)$.

Then, we take Expectation Maximization algorithm [41, 43] to estimate Q from our measurement data. The parameters q_{ij} of Q is updated by: $\frac{\mathbb{E}\left[n_{ij}|O,T,\hat{Q}_0\right]}{\mathbb{E}\left[\tau_i|O,T,\hat{Q}_0\right]}$, where $\mathbb{E}\left[n_{ij}|O,T,\hat{Q}_0\right]$ and $\mathbb{E}\left[\tau_i|O,T,\hat{Q}_0\right]$ are the expected state transition count and the total duration given the current transition probability matrix $\hat{Q}_0$, respectively. n_{ij} is the number of transition from state s_i to s_j, and τ_i is the total amount of time when the Android process remains in state s_i. The expectation for n_{ij} and τ_{ij} is calculated by the successive pairs of observations:

$$\mathbb{E}\left[X|O,T,\hat{Q}_0\right] = \sum_{s(t_1)}^{s(t_V)} p\left(s(t_1),...,s(t_V)|O,T,\hat{Q}_0\right) \times \sum_{v=1}^{V-1} \mathbb{E}\left[X|s(t_v),s(t_{v+1}),\hat{Q}_0\right],$$

where X can be n_{ij} and τ_{ij}, respectively.

Based on the $q_{i,j}$ derived from the above model, we can obtain the probability of a process transiting from state s_i to any other states s_j. Among all potential transition paths, we assign the process with the new state following the path that maximizes the transition probability.

After identifying "hogging" processes, we need to prevent them from co-awakening each other while minimizing the impact on user QoE. To this end, we register a system service called `AppStateManager` for managing the global app state based on their internal processes' states. Specifically, in Android, all of the apps have their own unique ID (UID), which is assigned when an app is installed in the system and is different from Linux's process ID (PID). Thus, we first find a list of the Linux processes connected to a specific app by maintaining a UID-indexed table that is updated whenever a new process is created or terminated in the Linux kernel via the `sysfs` interface. Then, using UID as a key, we can quickly retrieve all of the app's processes. After that, we assign the app's state by inspecting the identified real state of its internal processes. For instance, if there is one process within an app that is identified as the hogging state, the global state of the app is consequently set to the hogging state.

For potential hogging apps, we utilize the kernel's freezing capability to temporarily suspend the execution of their all processes for mitigating the side effects of improper termination of processes on user QoE. If the user does not actively re-interact with the app after a certain period (empirically set to 30s), we then terminate all the processes associated with the app. In this way, the co-awakening behaviors among the processes of hogging apps can be effectively avoided. Note that for apps that run some legitimate interaction-free background tasks (e.g., file download and data processing), our method will not categorize them as hogging apps, but allow them to run normally with visual cues such as an icon in the status bar to indicate such ongoing tasks to users.

5.2 eBPF-based Uniform Authentic Sensing

Recalling in §4, we reveal that hogging apps abuse the perceptible foreground services to increase their priorities for keeping alive in the background and occupying substantial system resources. Thus, it is crucial to sense the actual hardware utilization. However, it is quite challenging due to the complex implementation of corresponding Android subsystems (i.e., audio, GPS, Bluetooth, and network), which covers from user-space components to kernel modules.

Fortunately, an emerging revolutionary technique called Extended Berkeley Packet Filter [9] (or eBPF for short), which has attracted increasing attention in the field of operating system, gives us opportunities to address this challenge. We leverage the eBPF-based kernel probes (kprobe) and user probes (uprobe) to sense the real user-perceptible hardware usage (i.e., audio, GPS, Bluetooth, and network) across different layers for identifying hogging apps and informing the TIHMM-based process model, as shown in Figure 14.

First, we develop specific eBPF programs for tracing the usage pattern of each target user-perceptible hardware in C/C++ language and then compile them into eBPF bytecodes through the LLVM compiler [48]. After that, we load the eBPF bytecode into the kernel using the `bpf` system call along with the metadata including the program types (i.e., uprobe and kprobe) and the target hooks. Upon receiving the bytecode, the kernel verifies it to ensure that the eBPF program adheres to the kernel's safety constraints. The eBPF bytecode is then translated into native machine code through just-in-time (JIT) compilation.

Once JIT-compiled, the eBPF kprobe and uprobe are attached to the relevant kernel and user-space functions regarding the audio, Bluetooth, GPS, and network subsystems of Android, respectively:

- For audio, we attach uprobes to `AudioTrack`'s write method and `AudioRecord`'s read method to detect active audio playbacks and recordings. We also use kprobes in the advanced Linux sound architecture's functions to analyze raw audio data, like `snd_pcm_writei` and `snd_pcm_readi`.
- In network monitoring, uprobes are used to instrument system calls in user-space network stack libraries, while eBPF's skfilter targets kernel functions related to network protocol operations for traffic pattern analysis.
- For Bluetooth, we instrument its user-space modules like `BluetoothDroid` and kernel functions like `hci_send_pkt`

and `hci_recv_frame` in the host controller interface (HCI) to analyze Bluetooth activities.

- In GPS tracking, we use uprobes on the location APIs (e.g., `requestLocationUpdates` and `getLastKnownLocation`) and kprobes on the driver functions (such as `gps_start`, `gps_request_location`, and `gps_stop`) to analyze the GPS usage pattern.

The collected sensing data are then stored in the eBPF `maps` (a generic storage of different data types for sharing data between kernel and userspace) [34], from which our user-space TIHMM module retrieves them. Between the user-space TIHMM module and the kernel, we design an interface for high-throughput and low-latency data transfer. Specifically, we utilize the `perf_event_output` system API to efficiently push data from kernel to user space in a ring buffer format, which allows asynchronous data transfer. After that, the data interface converts the raw data into feature vectors, encapsulating key pattern characteristics like frequency, duration, and intensity of hardware usage. Through such data normalization and feature extraction, we can incorporate the sensing data into the TIHMM module as its observations.

6 DEPLOYMENT AND EVALUATION

To validate the real-world effect of our design, we integrate the optimizations to Android-MOD and invite the original 47M opt-in users in late Dec. 2022 to participate in our evaluation of the optimization mechanism (§5). On this occasion, 60% of the 47M users opted in and upgraded to our patched system, covering all the studied phone models. The evaluation spanned two months (Jan.–Feb. 2023). Note that even though the measurement study and evaluation are conducted during two disjoint time periods, the users targeted in the evaluation are from the user pool of the measurement study. Thus, we believe that their usage patterns of mobile phones remained consistent before and after the optimization.

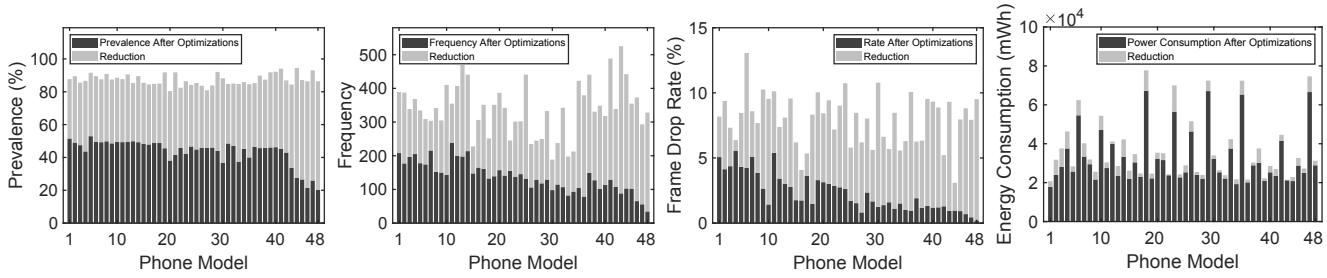
In addition, we do not limit ourselves to pure system solutions, but also coordinate app developers in the mobile industry. In detail, to address the app-specific SUR events, we have reported 415 detailed problem statements to corresponding app developers, among which 278 defects have been confirmed and officially fixed (as in Dec. 2022). The rest are either under beta tests or under code review. We present the evaluation results of the four metrics: *prevalence, frequency, frame drop rate, and energy consumption.*

Figure 15a and Figure 15b present the prevalence and frequency of SUR events for each phone model before and after the integration of the optimization mechanism in §5, respectively. As demonstrated in Figure 15a, overall, the proportion of phones experiencing SUR events per day has decreased by half (dropped from 87.37% to 43.72% across the phone models). As indicated in Figure 15b, the frequency of SUR events occurring daily on each phone has decreased by 59.90% (from an average of 339.40 times to 136.10 times). Particularly, for each individual phone model, the daily frequency of SUR events has decreased by 36.51%—82.12%.

In addition, we utilize the lightweight frame drop monitoring system described in §3.1 to record the frame drop rate of the devices after our patch was deployed. As shown in Figure 15c, on average, we have avoided 70.11% of frame drops per device per day (the frame drop rate decreased from 7.97% to 2.25%). Among them, the most significant improvement is observed in a high-end phone model, with a 97.79% decrease in frame drop events, reducing the frame drop rate from 9.52% to 0.21%. On the contrary, the least effective improvement is seen in a low-end model, where only 12.71% of frame drop events were reduced, with the frame drop rate dropping from 6.37% to 5.56%. In general, we observe a correlation between models and frame drop rates that more recent (and thus powerful) models exhibit lower frame drop rates. This indicates that the low-end devices benefit less from our optimizations than the high-end ones. This is understandable since low-end models have lower hardware configurations (e.g., memory capacity and CPU cycles). Thus, although our solution can prevent system resource under-provisioning to a certain extent, compared to high-end models, the low-end devices are more likely in a state of resource deficiency.

In the meantime, to evaluate the energy consumption of devices before and after integrating our optimizations, we leverage the existing `BatteryManager` APIs of Android which allows querying battery and charging properties. As shown in Figure 15d, the energy consumption of the entire device is reduced by an average of 10.69% (from 37420.76 mWh to 33420.47 mWh). The most significant improvement is a 25.07% reduction (from 37582.38 mWh to 28160.26 mWh) of phone model 3. The reason why the energy consumption of certain models (e.g., model 47) is twice that of some others can be attributed to model-specific features, such as a more coarse-grained screen-off power management strategy. In conclusion, these results suggest that our mechanism saves device energy by imposing restrictions on the unscrupulous system resource consumption of hogging apps.

To understand the probable false positives, we randomly sample 1200 user devices for manual examination and calculate the false positive (FP) rate and false negative (FN) rate of identified hogging apps. Our experiment results reveal that the FP and FN rates are 0.08% and 0.13%, respectively, both of which are very low. FPs occur when a benign app inadvertently acts like a hogging app. For example, when a video streaming app experiences frequent network disconnections and reconnections under poor network conditions, it exhibits behaviors similar to a hogging app that periodically sends garbage data for keeping alive. FNs are primarily from "utility" hogging apps. They reside in the background



(a) (b) (c) (d)

Figure 15: (a) SUR prevalence, (b) SUR frequency, (c) frame drop rate, and (d) energy consumption of each phone model per day before and after optimizations, with the total bar height indicating pre-optimization performance.

and stealthily push advertisements while keeping their resource usage low. We allow users to report these apps for our manual examination.

Meanwhile, to evaluate the overhead of our solution, we perform benchmark experiments by randomly sampling 480 user devices and recording the corresponding overhead when our solution is running on them. The results demonstrate that even in the worst case when the daily SUR occurrence reaches 600+, our solution induces little (and thus acceptable) overhead to user devices: <8% CPU utilization, <5 MB memory usage, <3 MB storage, and <100 KB traffic per day.

Robustness to Evasion Attempts. Recall from §1 that our design of vetting hogging apps is based on the observation from large-scale, state-of-the-art hogging behaviors. It targets non-malicious hogging apps intending to gain resource advantages. Thus, a dedicated/malicious developer who well knows our design may still succeed in evading our detection by adapting the app's behaviors using sophisticated techniques. For example, (1) a hogging app may intermittently produce sounds at a low volume or beyond human-audible frequency to evade detection; (2) a hogging app may actually turn on GPS, Bluetooth, or other sensors to sidestep our vetting; (3) an even more dedicated hogging app may perform additional work such as transferring garbage data and performing actual localization before discarding the result.

Some of the above behaviors such as (1) and (2) can be picked up by our authentic sensing layer. To alert the user, we also add visual cues such as an icon in the status bar to indicate an ongoing audio playback. Also note that in Android, using many I/O devices such as audio and GPS requires explicit user consent. For more stealthy behaviors such as (3), they could be regarded as malicious and are thus beyond the scope of this work. In all cases, our design significantly raises the bar for hogging app developers. For example, almost all the mainstream phone vendors offer a centralized system push service to deal with background network traffic, instead of letting background apps handle network connections by themselves. To bypass this and achieve (3) thus requires considerable hacking efforts with high risks of being flagged by the app market and/or anti-malware services.

Besides the above-mentioned attempts, malicious apps may employ crafty tactics such as constraining their resource consumption to stay below TIHMM's prediction thresholds and regulating their hogging behaviors to minimize the probability of being detected. These evasion tactics require deep knowledge of TIHMM and significant engineering efforts. Moreover, TIHMM is updated daily based on users' data, making evasions even more difficult.

Lessons Learned. Beyond tackling hogging apps to improve responsiveness, our study offers insights into behavior profiling and performance modeling of interactive mobile systems. First, to understand apps' real behaviors, one needs to capture actual user perceptions rather than relying on system APIs. Second, static performance modeling based on a predefined state machine is insufficient for today's mobile apps; instead, it's beneficial to model process management by considering dynamic time-sensitive factors, hidden process states, and complex interaction scenarios, etc.

7 CONCLUSION

This paper presents our experiences of understanding and mitigating slow UI responsiveness (SUR) in Android, a representative interactive mobile system. Despite its disruptions to mobile user experiences, SUR has never been well addressed due to the challenges of comprehensive measurements on massive devices. Our study fills the critical gap by conducting a large-scale crowd-sourced study with 47M Android phones. We deploy a continuous monitoring infrastructure to collect detailed data of every SUR event on users' devices. We then analyze the data and uncover multi-fold root causes of SUR. To address the major root cause, we develop an effective solution by remodeling the process management of Android, which has actually benefited tens of millions of users in terms of both SUR occurrence and battery consumption.

ACKNOWLEDGEMENT

We are grateful to our shepherd and the anonymous reviewers for their insightful and detailed comments. This work is supported in part by National Key R&D Program of China under grant 2022YFB4500703 and National Natural Science Foundation of China under grant 62332012.

REFERENCES

[1] AndroidSoft. 2023. How To Keep Apps Running In The Background IPhone? https://www.androidphonesoft.com/blog/how-to-keep-apps-running-in-the-background-iphone/.

[2] Anmol Sehgal. 2020. Android-Handlers, Loopers, MessageQueue Basics. https://anmolsehgal.medium.com/android-handlers-loopers-messagequeue-basics-d56a750df2cc.

[3] apple.com. 2023. App Store Review Guidelines. https://developer.apple.com/app-store/review/guidelines/.

[4] apple.com. 2023. Optimizing ProMotion refresh rates. https://developer.apple.com/documentation/quartzcore/optimizing_promotion_refresh_rates_for_iphone_13_pro_and_ipad_pro.

[5] Baiqin Wang. 2021. Binder Introduction. https://medium.com/swlh/binder-introduction-54fb90feeecb.

[6] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. Clonecloud: Elastic Execution Between Mobile Device and Cloud. In *Proc. of EuroSys*.

[7] csdn.net. 2023. Keeping Apps Alive in iOS. https://blog.csdn.net/WangQingLei0307/article/details/112002904.

[8] Pradip Kumar Das, Sayantan Shome, and Abhishek Kumar Sarkar. 2016. Apps: Accelerating Performance and Power Saving in Smartphones Using Code Offload. In *Proc. of IACC*.

[9] eBPF.io. 2023. eBPF Documentation. https://ebpf.io/what-is-ebpf/.

[10] Eric Silverberg. 2021. Android Activity Lifecycle Considered Harmful. https://proandroiddev.com/android-activity-lifecycle-considered-harmful-98a5b00d287.

[11] Liangyi Gong, Zhenhua Li, Hongyi Wang, Hao Lin, Xiaobo Ma, and Yunhao Liu. 2022. Overlay-Based Android Malware Detection at Market Scales: Systematically Adapting to the New Technological Landscape. *Transactions on Mobile Computing* 21 (2022).

[12] Google.com. 2020. High Refresh Rate Rendering on Android. https://android-developers.googleblog.com/2020/04/high-refresh-rate-rendering-on-android.html.

[13] Google.com. 2022. Android 12 Features and Changes List. https://developer.android.com/about/versions/12/summary.

[14] Google.com. 2022. Google I/O 2022: What's new in Jetpack. https://android-developers.googleblog.com/2022/05/whats-new-in-jetpack.html.

[15] Google.com. 2022. Multiple Refresh Rate. https://source.android.com/docs/core/graphics/multiple-refresh-rate.

[16] Google.com. 2022. WebView. https://developer.android.com/reference/android/webkit/WebView.

[17] Google.com. 2023. Android Debug Bridge (ADB). https://developer.android.com/tools/adb.

[18] Google.com. 2023. Android Low Memory Killer Daemon. https://android.googlesource.com/platform/system/memory/lmkd/+/refs/heads/master/README.md.

[19] Google.com. 2023. Android Processes and App Lifecycle. https://developer.android.com/guide/components/activities/process-lifecycle.

[20] Google.com. 2023. Application fundamentals. https://developer.android.com/guide/components/fundamentals.

[21] Google.com. 2023. Looper of Android. https://developer.android.com/reference/android/os/Looper.

[22] Google.com. 2023. Overview of system tracing. https://source.android.com/docs/core/tests/debug/read-bug-reports.

[23] Google.com. 2023. Performance of Rendering. https://developer.android.com/topic/performance/rendering.

[24] Google.com. 2023. Reading Bug Reports. https://source.android.com/docs/core/tests/debug/read-bug-reports.

[25] Google.com. 2023. SimplePerf on Android. https://developer.android.com/ndk/guides/simpleperf.

[26] Google.com. 2023. The Slow Rendering of Android. https://developer.android.com/topic/performance/vitals/render.

[27] Google.com. 2023. VSYNC. https://source.android.com/docs/core/graphics/implement-vsync.

[28] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. 2012. COMET: Code Offload by Migrating Execution Transparently. In *Proc. of OSDI*.

[29] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. 2018. FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones. In *Proc. of ATC*.

[30] https://perfetto.dev/. 2023. Quickstart: Record traces on Android. https://perfetto.dev/docs/quickstart/android-tracing.

[31] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New phytologist* 11, 2 (1912), 37–50.

[32] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proc. of ATC*.

[33] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. 2012. Cuckoo: A Computation Offloading Framework for Smartphones. In *Proc. of MobiCASE*.

[34] kernel.org. 2023. eBPF maps. https://www.kernel.org/doc/html/v5.18/bpf/maps.html.

[35] Ron Kohavi, Diane Tang, and Ya Xu. 2020. *Trustworthy Online Controlled Experiments: A Practical Guide to A/B Testing*. Cambridge University Press.

[36] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. Thinkair: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading. In *Proc. of Infocom*.

[37] Mingliang Li, Hao Lin, Cai Liu, Zhenhua Li, Feng Qian, Yunhao Liu, Nian Sun, and Tianyin Xu. 2020. Experience: Aging or Glitching? Why Does Android Stop Responding and What Can We Do About It?. In *Proc. of MobiCom*.

[38] Hao Lin, Cai Liu, Zhenhua Li, Feng Qian, Mingliang Li, Ping Xiong, and Yunhao Liu. 2021. Aging or Glitching? What Leads to Poor Android Responsiveness and What Can We Do About It? *IEEE Transactions on Mobile Computing* 32 (2021).

[39] Yu Lin, Semih Okur, and Danny Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *Proc. of ASE*.

[40] Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting Concurrency for Android Applications Through Refactoring. In *Proc. of FSE*.

[41] Yu-Ying Liu, Alexander Moreno, Shuang Li, Fuxin Li, Le Song, and James M Rehg. 2017. Learning Continuous-Time Hidden Markov Models for Event Data. *Mobile Health: Sensors, Analytic Methods, and Applications* (2017), 361–387.

[42] Yu Luo, Kirk Rodrigues, Cuiqin Li, Feng Zhang, Lijin Jiang, Bing Xia, David Lion, and Ding Yuan. 2022. Hubble: Performance Debugging with In-Production, Just-In-Time Method Tracing on Android. In *Proc. of OSDI*.

[43] Todd K Moon. 1996. The Expectation-Maximization Algorithm. *IEEE Signal Processing Magazine* 13, 6 (1996), 47–60.

[44] David T Nguyen. 2014. Improving Smartphone Responsiveness Through I/O Optimizations. In *Proc. of UbiComp*.

[45] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns.

In *Proc. of ICSE*.

[46] Thanaporn Ongkosit and Shingo Takada. 2014. Responsiveness Analysis Tool for Android Application. In *Proc. of DeMobile*.

[47] Perfetto.dev. 2023. ATrace: Android System and App Trace Events. https://perfetto.dev/docs/data-sources/atrace.

[48] Quentin Monnet. 2020. eBPF assembly with LLVM. https://qmonnet.github.io/whirl-offload/2020/04/12/llvm-ebpf-asm/.

[49] Lawrence Rabiner and Biinghwang Juang. 1986. An introduction to Hidden Markov Models. *ASSP Magazine* 3, 1 (1986), 4–16.

[50] Rafael J. Wysocki. 2007. Freezing of Tasks. https://www.kernel.org/doc/html/next/power/freezing-of-tasks.html.

[51] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proc. of ICSE*.

[52] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in The Wild. In *Proc. of OSDI*.

[53] reddit.com. 2023. Is It Illegal to Continuosly Play Silent Audio File in the Background? https://www.reddit.com/r/swift/comments/10o11ap/is_it_illegal_to_continuosly_play_silent_audio/.

[54] Scribbr. 2023. Guide and Examples of Pearson Correlation Coefficient. https://www.scribbr.com/statistics/pearson-correlation-coefficient/.

[55] Surachai Thongkaew, Tsuyoshi Isshiki, Dongju Li, and Hiroaki Kunieda. 2015. Dalvik Bytecode Acceleration Using Fetch/Decode Hardware Extension. *Journal of Information Processing* 23 (2015).

[56] Uber. 2018. Introducing Nanoscope: An Extremely Accurate Method Tracing Tool for Android. https://www.uber.com/en-JP/blog/nanoscope/.

[57] Cheng Wang, Youfeng Wu, and Marcelo Cintra. 2013. Acceldroid: Co-designed Acceleration of Android Bytecode. In *Proc. of CGO*.

[58] Yan Wang and Atanas Rountev. 2016. Profiling The Responsiveness of Android Applications via Automated Resource Amplification. In *Proc. of MOBILESoft*.

[59] wetest.net. 2023. PerfDog: Full Platform Performance Test and Analysis Tool. https://perfdog.wetest.net/.

[60] Wikipedia. 2023. Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[61] Dacong Yan, Shengqian Yang, and Atanas Rountev. 2013. Systematic Testing for Resource Leaks in Android Applications. In *Proc. of ISSRE*.

[62] Yuxuan Yan, Zhenhua Li, Qi Alfred Chen, Christo Wilson, Tianyin Xu, Ennan Zhai, Yong Li, and Yunhao Liu. 2019. Understanding and Detecting Overlay-Based Android Malware at Market Scales. In *Proc. of MobiSys*.

[63] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *Proc. of ATC*.