Producing Software Obsolescence: the case of Android OS

Journal:	*ACM Journal on Responsible Computing*
Manuscript ID	Draft
Manuscript Type:	Research Article
Computing Classification Systems:	Software and its engineering~Software creation and management~Collaboration in software development~Open source model, Software and its engineering~Software organization and properties~Contextual software domains~Operating systems, Software and its engineering~Software creation and management~Software post-development issues~Maintaining software, Social and professional topics~Professional topics~Computing profession~Computing organizations, Social and professional topics~Computing / technology policy, Social and professional topics~Professional topics~Management of computing and information systems~Software management~Software, Social and professional topics~Professional topics~Management of computing and information systems~System management, Human-centered computing~Collaborative and social computing~Empirical studies in collaborative and social computing

Producing Software Obsolescence: the case of Android OS

In the Android ecosystem, the most widely used operating system (OS) in the world, devices are rarely updated more than two years after their release. We investigate what hinders Android maintenance, and the deployment of OS updates. We conducted twelve interviews with key players in the ecosystem, supplemented with conference ethnography and analysis of technical literature. Based on this corpus, we clarify the structure of the Android ecosystem and show that there is not just one Android OS, but rather a specific Android build for each smartphone model. We show that the way code flows are organized across the various ecosystem actors inhibits updates, and we outline how these actors locate their maintenance efforts in different places to serve their strategic interests.

The lack of updates appear at the kernel level, i.e, at the core of Android builds, as the code from phone vendors and system on chip manufacturers increasingly diverges from the original Linux kernel code. We show that Google, the main actor governing the Android ecosystem, addresses maintenance issues by shifting the responsibility towards phone vendors. However, as vendors are the least inclined actors to maintain their code, the problem remains, leading to premature end-of-life for devices and, consequently, their obsolescence. In parallel, we analyze how, driven by a concern on longevity, some phone vendors and alternative free and open-source mobile actors are implementing remediation strategies to maintain devices.

Reflecting on the Android ecosystem, we discuss the various maintenance strategies we have observed and the play between openness and closure in software development and maintenance.

CCS Concepts: • **Software and its engineering** → **Open source model**; **Operating systems**; **Maintaining software**; • **Social and professional topics** → *Computing / technology policy*; **Computing organizations**; **Software maintenance**; *System management*; *Governmental regulations*; • **Human-centered computing** → **Empirical studies in collaborative and social computing**.

Additional Key Words and Phrases: Obsolescence, Smartphone, Android, Update, Fragmentation, Frankenkernel, Linux, Google, FOSS

ACM Reference Format:

. 2026. Producing Software Obsolescence: the case of Android OS. 1, 1 (February 2026), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Smartphones appear to be the poster child of "disposable technology", i.e., devices that are neither maintained, nor meant to be repairable or recyclable, but rather replaced [34, 55]. By way of illustration, more than 1.2 billion smartphones were sold worldwide in 2024, with an average lifespan around 3.5 years. By comparison, embedded software typically have lifespans of 7 to 15 years. And the software on the Voyager 1 space probe software is still running nearly 50 years after its launch, with software updates performed in 2023.

The lack of updates is one of many reasons why consumers renew their smartphones. Although social and psychological factors are at play in renewal decisions, software factors do have a significant role [39]. Without software maintenance, devices tend to become slower and less reliable, but also less secure.

Author's Contact Information:

Manuscript submitted to ACM

In the Android ecosystem, devices are rarely updated, despite the release of a new version of the Android operating system (OS) every year. The proliferation of Android phone vendors, device models, and active OS versions, known as the Android fragmentation problem, generates maintenance issues at both the OS and application levels [59].

Recent studies define the lifespan of devices in terms of obsolescence [7, 8], viewing it not only as a technical or engineering quality, but also as being influenced by socio-cultural and socio-economic factors. As Cohn argues in her work on software maintenance [6], we hypothesize that the Android update process should also be investigated through a social and organizational lens. Rather than a simple OS, Android can be understood as an ecosystem of organizations involving actors with different values, who interact through code exchanges, shared libraries, documentation, test benches, commercial partnerships, competing or complementary business models, to name but a few.

In order to study the socio-technical challenges associated with the development and maintenance of Android, we carried out a multifaceted fieldwork. We conducted 12 interviews with key players in the Android mobile ecosystem (developers of the Linux kernel, Google, Fairphone, Commown, and alternative mobile OSes or libraries). We supplemented these interviews with conference ethnography, and an analysis of technical literature.

Building upon our corpus, we seek to answer the following questions:

- How is Android structured? Who are the actors involved, and what is the Android building process?
- What inhibits Android updates? Where does software obsolescence manifest itself?
- What are the strategies of actors in Android or non-Android ecosystems in tackling maintenance issues?

After presenting related work on obsolescence, Android and maintenance, we detail our methodological approach. We then present our results as follows: first, we show how the Android operating system is structured into different software layers, the organizations involved in developing each of these layers, and the detailed pipeline for building the Android system. We then show what inhibits updates, and where obsolescence occurs in practice: which actors are involved and how. We analyze the development flow of the Android ecosystem, how actors interact in building and maintaining (or not). This allows us to identify maintenance breakpoints, that lead to premature end-of-life of devices and, consequently, their obsolescence.

Our findings show that there is not one Android OS, but rather one specific Android build for each smartphone model. The lack of updates appears at the kernel level, i.e, at the core of Android builds, as the code from phone vendors and system on chip manufacturers increasingly diverges from the original Linux kernel code. Google, the main actor governing the Android ecosystem, addresses maintenance issues by seeking to create a dedicated private space within the OS for industrial actors (phone vendors, system on chip manufacturers), whereas the open-source community would prefer these same actors to contribute and share their knowledge. However, as vendors are the least inclined actors to maintain their code or share their knowledge, the problem remains. Given this tension, we observe how the remediation and maintenance strategies diverge, how the maintenance responsibility shifts from one actor to another, and how, driven by a concern for longevity, some phone vendors and alternative free and open-source mobile projects implement remediation strategies to maintain devices.

Drawing on the Android ecosystem and existing literature analyzing complex socio-technical ecosystems involving open-source actors, we discuss the values and interests of actors, how they enact power, and the interplay between openness and closure in software development and maintenance.

2 Background and related work

Android is the leading operating system worldwide. According to Google, it held 70% of the mobile OS market share in 2023, with approximately 3 billion active Android devices [1], with the remaining 30% mainly held by Apple iOS.

A specificity of Android and its ecosystem is that it is structured around a major technology company, Google, but relies at its core on the Linux kernel, a free and open-source software (FOSS). It also involves industrial smartphone assemblers and manufacturers (known as OEMs, for Original Equipment Manufacturers, sometimes also referred to as "vendors"). This ecosystem is a unique mix of communities and organizations that interact with each other in building and maintaining the system.

In this article, we focus on the specific challenges related to maintaining and ensuring the evolution of the Android operating system on smartphones, both old and new. What interests us here is the "dumpster fire" [17] of socio-technical problems at the interface of the operating system, firmware (software embedded in hardware) and hardware, and how the various actors in the Android ecosystem engage (or not) in maintenance efforts.

2.1 Managing software obsolescence for industry needs

Software maintenance has been a long-standing concern in software engineering research and industry. While early methodologies attempting to deal with commercial-off-the-shelf obsolescence are related to hardware and spare-parts, software obsolescence became a topic of concern in the early 2000s [2, 37, 42, 48]. This led to a definition of software obsolescence centered around prediction for industry-specific needs [47]. Industries such as the military, aviation, rail, automotive or industrial machinery manufacturing, which are concerned with ensuring the longevity of the systems they integrate in their organizations, are often cited and taken as examples of organizations managing obsolescence, through forensics and prevention methods [16, 46].

2.2 Software maintenance and evolution

In the software industry at large, despite the emphasis on innovation in public discourse, much of the software work is centered on maintenance. As Webster et al. point out, *"maintenance is an unavoidable activity required to keep systems synchronized with the reality they are modeling, a reality that changes continuously"* [58]. As the environment in which the code evolves, the functioning code "decays" [12]. Synthesizing surveys in the literature, Canfora and Cimitile estimate that software maintenance consumes between 60% and 80% of the total life cycle cost of software projects [4]. But they also note that, according to these same surveys, a large share of these maintenance costs (75% to 80%) relates to enhancements rather than corrections. In this sense, maintenance goes beyond correcting bugs, with a large portion related to evolutive maintenance.

In FOSS systems, maintenance is central to the work of the communities that develop them. In their work on the labor of maintaining and scaling FOSS systems, Geiger, Howard and Irani detail how maintainers in the community have a central role, and how *"maintenance is not only about repairing and fixing. It is crucially about updating and changing to stay relevant"* [18].

2.3 Maintaining Android OS and Android applications in a fragmented ecosystem

A major version of Android is released every year, with the latest being Android 16 in June 2025 [2]. These new versions of the Android OS are not distributed or installed on all active devices: at any given time, several Android OS versions are

[1] Google blog - Android Updates 2023, accessed on Jan. 29 2025.
[2] Wikipedia - Android version history, last accessed in Sept. 2025.

active simultaneously among them. Figure 1 shows the distribution of Android OS versions on active devices in April 2025, one month before the release of the new version 16. Android 15 was present on only 4.5% of the devices, versions 10 to 14 (six to two years old, respectively) were all widely used[3], and more than 50% of the market was running an OS that was at least four years old OS (Android 12 and earlier version).

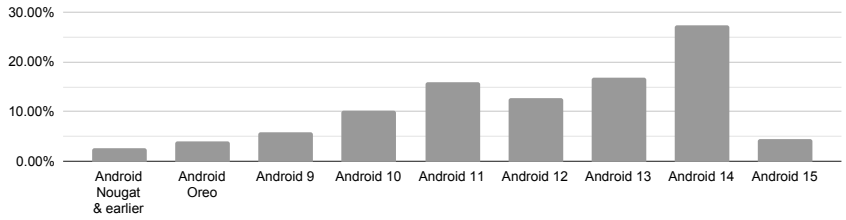

Fig. 1. Marketshare of Android OS distributions in April 2025

This proliferation of Android OS versions, mostly old ones, installed on many devices at any given time, has been called the Android fragmentation problem. It began being discussed and analyzed in specialized media[4] or business reports[5] as soon as 2010. Over time, fragmentation is discussed as being at the core of maintenance concerns of both the Android OS and of the applications running on top of it. It is both talked about by Android developers[6] and by scholars [23, 59, 60].

Fragmentation is then defined as a two-sided problem: a hardware-based and a software-based fragmentation[7]. Hardware fragmentation refers to the lack of hardware uniformity on Android (with Android devices holding different System on Chips (SoC), screen types, modems, etc.), which generates a lot of edge cases and bugs for developers to handle [23]. Software fragmentation refers to end-user devices running on different versions of Android OS.

At the software level, Android application developers face portability and compatibility issues, meaning that their code does not readily support multiple devices or Android versions (and its underlying libraries) [23]. Wei et al. show that app developers have to cope with the ways in which various smartphone vendors defined default values, the way they expose some of their Application Programming Interfaces (API) in idiosyncratic ways, etc. When updated to fit more recent OS versions and to follow the newest Android Software Development Kit (SDK), apps tend to be optimized for newer, more powerful devices. This can lead to performance issues on older phones such as taking a long time to respond, needing more resources, therefore leading to battery use and slow-downs [59], which in turn lead to faster replacement of devices by users [39]. Linares-Vasquez et al. also note that apps leveraging changing APIs tend to receive lower user ratings [32]. Because App developers are concerned with ratings [31], worsening ratings could lead them to artificially deprecate old applications, to avoid maintenance efforts and bad ratings.

Last but not least, by running old OS versions, phones are exposed to security risks, leading to a high rate of device renewal in work environments, but also in communities and contexts that emphasize software security [29, 53]. This narrative is also pushed by industrial actors who often emphasize security as a reason for OS upgrades, or for changing hardware. Recent scholarship on security brings some nuance, Korn and Wagenknecht propose to consider 'security research' as an ambivalent form of repair and maintenance [27]. They examine how frictions arise in what they call

[3] Android distribution data from Google, April 2025, last accessed in Oct. 2025.
[4] See V. Madhav, Fragmentation in Android: Boon or Bane, 2010, last accessed in Feb. 26, 2025.
[5] Opensignal report on Android fragmentation in 2012, and then yearly until 2015, accessed on Feb. 26, 2025.
[6] Linux World News discussion on fragmentation, accessed in Feb. 2025.
[7] See S. Singh, An Analysis of Android Fragmentation, 2012, last accessed in Feb. 2025.

"*social arena of repair*" between industrial actors and hacker and security activists. Kocksch notes that computer security is situated and should be seen as living with fragility [26].

2.4 Smartphones, SoCs and new maintenance vulnerabilities

Android OS has several specific features as an operating system developed for mobile devices. The main component of a smartphone is the system on chip (SoC). A SoC is a single chip containing all the key components of a system soldered side by side: one or more microprocessors, the memory, one or more graphics processors to manage display or AI calculations, modems (for mobile networks or WiFi) and sensors (Bluetooth, infrared, biometrics, etc.). The very fine soldering of these components in a single chip saves space, uses less energy, and allows the creation of portable battery-powered devices such as smartphones. Their design and manufacturing in a single piece allows mass industrial production at lower costs [5]. SoCs are complete embedded systems that began being commercialized in the early 2000. Their technical development has led to the emergence of IoT devices and smartphones as well as companies dedicated to SoC and smartphone production such as Qualcomm, MediaTek, Samsung Exynos.

SoCs come with embedded software which is code that provides low-level control of its components and manages peripheral hardware. When a smartphone is in use, SoC's embedded software manages all the interactions that happen with phone components (touchscreen, cameras, speaker, microphones, etc). A SoC carries both hardware and software architectures, that are co-developed simultaneously during the design flow [25]. This creates new intertwined hardware and software maintenance vulnerabilities: replacing a faulty component in the finely soldered SoC is difficult, even for professional repairers, while embedded software adds a layer of dependency[20] that, according to Greengard *"makes it easier for manufacturers and rights holders to block repairs and control the aftermarket*". While civil society organizations engage more and more in Right To Repair campaigns[8] insisting on the fact that reclaiming full ownership of hardware and software in order to control devices is an important strategy for addressing obsolescence, there is a need to better study the role that SoCs play in software maintenance of smartphones or other embedded systems.

2.5 Social practices around maintenance

Beyond the software industry, scholarship on "maintenance and repair studies" developed at the intersection of various fields, from geography [19], to sociology [9]. This work broadly attempts to tie the literature on Care to the material consideration related to maintenance and repair [10]. It draws notably on Anne Marie Mol [38] and Maria Puig de la Bellacasa work on articulating *"maintenance and repair as processes dedicated to restoring order*" [10]. This approach invites to consider tools and technical artifacts not as solid and permanent but as fragile objects, with a constant need for care to remain functional. This scholarship has studied infrastructures, cars, homes, and more, but is much more limited when it comes to software and ICT. Marisa Leavitt Cohn's study of maintenance at NASA [6] is an exception.

Daniela Rosner and Morgan G. Ames, building upon field studies of repair practices, also note that *"breakdown and repair are not processes that designers can effectively script ahead of time; instead, they emerge in everyday practice*" [45]. This relates to the fragile nature of digital technologies emphasized by Steven Jackson: breakdowns are constitutive of technological systems and happen all the time [24]. But their identification and the worthiness of maintenance is constantly negotiated. Den Hollander [7, 8] also suggests that obsolescence and product lifetime are not just technical qualities, but affected by social, economical and interaction factors that can be influenced and changed over time.

[8]See e.g. Right to Repair Europe, PIRG USA, or organizations on the iFixit worldwide map.

By focusing on the relationships and the social arrangements taking place in software production and the technical constraints shaping development, scholars in science and technology studies contextualize software production not only as a technical issue, but as a socio-technical one, in which technical constraints will shape social order, as much as social and organizational forms will define how software is produced. This can be seen in Kocksch and Jensen LM: 's recent study of cybersecurity practices in small and medium enterprises [26], places in which fragility must be managed and risk constantly negotiated.

At a more macro level, Maldini et al. discuss how research literature and current regulations on product lifetime extensions are often based on the unproved assumption that industrial production does not come as an adaptation to user demand, but elements tend to suggest that overproduction occurs frequently [35]. By studying the market and social history of everyday objects, including smartphones, Guien [21] argues that obsolescence is at the very heart of the business model for consumer goods and has been theorized as such since early management studies in the 1930's. From this historical and economic perspective, obsolescence and longevity should not be understood as states, but rather as being produced and managed, in a society structured around the push towards consumption of new products.

3 Methods

This work is part of a broader effort of the authors to understand obsolescence of mobile devices, which involved prior field work on the perception of obsolescence, participatory workshops on smartphones upgrades, and active monitoring of public debates on smartphone longevity. This led us to wonder what makes Android smartphones especially challenging to update and maintain.

Our investigation was carried out on three complementary levels. We conducted 12 interviews with actors of the Android OS ecosystem. We also carried out conference ethnography at both in-person and online conferences and developer gatherings of Android ecosystem actors. We also analyzed historical and technical documents on specialized websites, media and developer online spaces.

3.1 Interviews

We conducted semi-structured interviews with twelve informants from September 2023 to November 2024. Some of them took place in person during conferences or gatherings, while others were conducted online. For each interview, we presented the purpose of the interview and explained how the information would be used. In the interview consent form, informants who agreed to talk to us could choose to appear anonymously or with their real name and professional activities. One participant preferred anonymity. Furthermore, every participant received a copy of the paper that was submitted, with an invitation to review and discuss what they would consider as misrepresentations or errors.

3.1.1 Informants. We interviewed 12 main informants presented in Table 1. A significant part of our work consisted of identifying informants with deep knowledge of Android and its ecosystem. Given Google's centrality, we contacted several developers working at the company while being transparent about our research topics and questions. After several months we received a final email stating that the corporate position was not to communicate with researchers about Android on these sensitive topics. Nevertheless, we interviewed an informant working at the company before we received the "official" position.

At the smartphone manufacturer and vendor level, we conducted interviews with Fairphone employees, a company that builds and markets a "fair and durable" device, as well as Commown, a company promoting smartphone longevity through an original business model.

#	Name	Organization/Project	Function	Date	Duration
1	Agnès Crepet	Fairphone	Head of Software Longevity & Information Technology	2023-09-14	1h
2	P2 (anonymized)	Google	Developer	Fall 2023	1.5h
3	Marvin Wissfeld	MicroG & LineageOS for MicroG	Main developper of MicroG	2023-12-22	1h
4	Emanuele Rocca	Debian & ARM	Debian developer and maintainer, ARM developer	2023-11-25	1h
5	Ben Hutchings	Debian & Linux kernel	Maintainer for Debian Linux kernel packages and linux-firmware repository	2024-05-20	1h
6	Arnaud Ferraris	Mobian	Linux on mobile and Mobian team leader	2024-11-19	1.5h
7	Federico Ceratto & Jochen Sprickerhof	Mobian	Developers	2024-05-16	1h
8	Denis Carikli & David Ludovino	Replicant OS	Main developers	2024-07-01	2.5h
9	Johannes Schauer Marin Rodrigues	Debian & MNT reform	Developer for MNT Reform port in Debian	2024-05-19	1.5h
10	Elie Assémat	Commown	Cofounder	2023-09-20	1.5h
11	Adrien Montagut-Romans	Commown	In charge of advocacy towards France and EU administrations	2024-09-16	1.5h
12	Simon Gougeon	UnifiedPush	Main developer	2025-12-05	1.5h

Table 1. List of interviews

At the OS level, we interviewed developers working on the Linux kernel upon which Android is built. We also interviewed Debian OS developers. Both clarified distributed development practices related to maintenance.

Notably we interviewed two developers with experiences on adapting the Linux kernel on a new System on Chip (SoC), a process called porting. These interviews clarified how the kernel communicates with hardware embedded software (also called firmware) at the SoC level, in order to implement full device functionality at the OS level (through drivers).

We interviewed several developers and community members of Android-based alternative OSes, such as LineageOS, LineageOS for MicroG and Replicant. We interviewed the main developers of two important tools for the Android OS: UnifiedPush, a decentralized open-source protocol and libraries for push notifications in Android that follows the IETF WebPush standards; and micro-G, a free open-source implementation of the Google Play Services. We also interviewed non Android Linux-based OS developers from PostmarketOS and Mobian. These helped us understand community coding practices and policies implemented over time, to facilitate development and maintenance in smartphone OSes that also derive from Linux but differ from Android's industrial dominant ecosystem both in coding values and practices.

3.1.2 Analysis process. The interviews were crucial in understanding the Android development ecosystem, actors, interactions and identifying friction points, or points of interest for our research questions. Because they took place at different stages of our research, over an extended period of one year, they played different roles. Some were decisive in that they brought to our attention an unexpected issue that proved important in our understanding. These key issues then guided our next interviews: when something caught our attention during an interview or the analysis process, we

organized new interviews and continued conference and desktop research to explore these specific issues and deepen our understanding.

The interviews were audio-recorded with the consent of the interviewees. Most of them were transcribed to text by using the Vosk offline open-source speech recognition toolkit[9], followed by a human transcription. The transcripts served our study in several ways: deepening our understanding by discovery of key points, highlighting uncertainties and questions to be clarified, triggering new interviews or research on them. We kept a text journal of the most important quote excerpts from these transcripts, and what they triggered or highlighted. Many of them also appear in this article. Further quote excerpts were added from conferences that we attended in person or watched online, all being freely available in audio, video or text-transcribed versions. This selection was motivated by the relevance to our mapping of the ecosystem, to the interactions between actors, and finally, to our findings and discussion points. During the writing process, the quotes were carefully checked within their context before being used.

Mappings of the interactions within the ecosystem, of the Android stack, and of the development pipelines played an important role in our analysis. The iterative process of creating, discussing, and fine-tuning the resulting figures among co-authors, enabled us to identify limitations in our understanding of Android, map interactions among actors and organizations, and instantiate abstract discourse to specific development activities. These diagrams also enabled us to confront our understanding with external informants familiar with the Android ecosystem. They could (in)validate our understanding of Android, signal elements they discovered thanks to us, or direct our attention to shortcuts or missing elements in our mappings.

3.2 Conference ethnography

The first author participated in person in conferences, multiple-day community gatherings and followed numerous online conferences on the technical aspects of Android OS development and update process, including: the annual Linux plumber conference (online), the annual Linux kernel recipes conference (online), the Open Firmware conference (online), Capitole du Libre (online in 2023, in person in 2024), the Free Silicon conference (in person) about free and open-source design and manufacturing of chips, the Debian OS annual conference (online) two European Debian OS community gatherings (in person), as well as the FOSS on Mobile Devices conference day at FOSDEM in 2024 and 2025 (online). This conference immersion also enabled the first author to conduct interviews, get recommendations, and learn through discussions with developers from the Linux, Android and alternative OS communities, some dynamics that did not surface in more formal communications.

3.3 Technical documentation immersion

To complement the ethnographic work, we analyzed the official Android developer documentation by Google, technical documentation on alternative Android OS based systems such as LineageOS, LineageOS for microG, e/OS, Replicant, GrapheneOS and on alternative non Android mobile OSes based on Linux, such as PostmarketOS and Mobian. These helped us better understand specific problems or techniques such as fragmentation, porting, upstreaming, mainlining, backporting. We also followed specialized news and analysis media such as lwn.net (a Linux news and information website), 9to5google.com (news about Google) Android Authority (news about Android) or OS News (news about operating systems), and selected articles from Ars Technica, The Verge or Wired magazines. We also leveraged developer

[9] The Vosk Speech Recognition Toolkit repository

community forums and mailing lists for specific information about OS releases (e.g. xda-developers, Reddit, Fairphone or e/OS community forums).

Reports and research work by and on the Right to repair movements in different continents [33, 50, 54, 56, 61] together with our interview with one of Commown's co-founders working in advocacy, were important to understand software-hardware repair and maintenance issues worldwide, as well as the state of regulations in the US, where the movement for Right to Repair originates, in France, and at the European Union (EU) level. At the economical and legal level, the reports on the Android antitrust infringement cases against Google in Europe [49][14], in the US [44], UK, recently in Japan[10]. In particular, the detailed investigations on Google that some of these reports [40, 41] detail, helped us to understand Android practices, to confirm or mitigate some of our findings, and to learn more about contractual relations between Google and vendors.

3.4 Positionality statement

The first author has been continuously installing and testing mobile OSes for the past 10 years, and discussing within the community of hackers problems and solutions related to software on old smartphones. The first author is also an active member of La Quadrature du Net, a French nonprofit organization defending digital rights, as well as of April, a French nonprofit defending and promoting free software technology and its ethical and social values[11].

The co-authors have backgrounds in design and computer science. They are part of a broader project studying digital obsolescence and longevity from a technical and social point of view. As such, all authors have followed and organized numerous exploratory workshops and discussions both with developers, hackers and users on smartphone usage, settings, tweaking, or hacking.

4 Results

We now outline the socio-technical challenges of developing and maintaining Android. We first present an overview of Android software layers, focusing on the ones that are relevant to upgrading and maintaining Android. We present the organizations involved in developing these layers, and the Android build pipeline (4.1). Building upon this technical picture, we detail how obsolescence happens in practice (4.2) and what inhibits maintenance (4.3). Simultaneously, we present how the actors involved tackle (or not) maintenance problems, the strategies they develop to enable software maintenance on aging smartphones (4.3) and their limits (4.4).

4.1 An overview of Android

As the most widely used operating system in the world, Android should not be seen only in the technical sense assigned to operating systems in the Computer Science literature, i.e. a tool to abstract device hardware and manage resources for its users and their applications (the OS kernel). We study Android as a complex ecosystem, involving hardware manufacturers, phone vendors, app developers, and end-users, all interacting with each other in technical but also commercial, industrial, legal and social ways.

4.1.1 Android software layers and actors. To give a sense of the Android architecture, in Figure 2 (left) we offer a simplified overview of its software layers, and for each of them we explain their role and the actors involved in their development, maintenance and governance.

[10] On the Japan Fair Trade Commission's Google Decision: Some Early Reflections, Sangyun Lee, Kyoto University, April 2025, last accessed in Nov. 2025.
[11] See La Quadrature du Net's and April's websites.

The **Hardware and Firmware layer** (bottom) represents the code embedded into the hardware of smartphone devices. Every piece of smartphone hardware, be it modems, cameras, sensors, touchscreens or System on Chips (SoC), comes with embedded software, often called firmware. Embedded software is necessary to ensure the functioning of the hardware and to provide low-level control of the hardware to higher-level software such as the operating system. This software is developed and maintained by the hardware manufacturing companies for the Android ecosystem. They are responsible for providing fixes when bugs or security issues are discovered, or for updating the software when a new version of the Android operating system requires it.

The **Operating System (OS)** abstracts device hardware and manages the device resources for its users and their applications [51]. The OS communicates with the hardware via dedicated software libraries often called drivers, which interact directly with their firmware. Google is the main developer of the Android OS, defining a general scheme for these drivers and the system to interact with hardware components. The drivers and OS modifications are provided by phone vendors and phone hardware manufacturers in order to ensure that the specific cameras, modems or other components are functional in the OS.

Finally, the **Application layer** is where anyone can introduce new software, with the aim of developing applications (apps) that make the phone directly usable to its users.

On top of these three software layers, we identified two cross-layers. First, the **Background system-wide services and apps** containing applications and libraries that often run in the background (are not immediately visible to the user), are pre-installed in the phone, and have special privileged access to the whole Android system. These can be created by Google, phone vendors or network operators and include services or applications such as localization, data tracking, advertisement injection, push messaging, permissions management, network services, etc. Most of the time they cannot be uninstalled by users, and if blocked or uninstalled by experienced users using special techniques there is a risk of altering phone functionalities. They are most of the time proprietary code software of Google or phone vendors, as for example Google Play Services (a Google system-level package of services that almost all Android apps depend on), Samsung Push Services (a Samsung system-level package of services that Samsung apps depend on), etc.

The second transverse layer is the **System on Chip & hardware integration** one, which we identified as a collection of software code, firmware and drivers, hardware documentation and schematics, or "hacks", that are needed in a mobile phone to support its particular SoC, and all other hardware such as touchscreen, battery, etc. They can be located both in the OS layer for their core functionalities (for example, a generic driver making a camera work), but can also be very

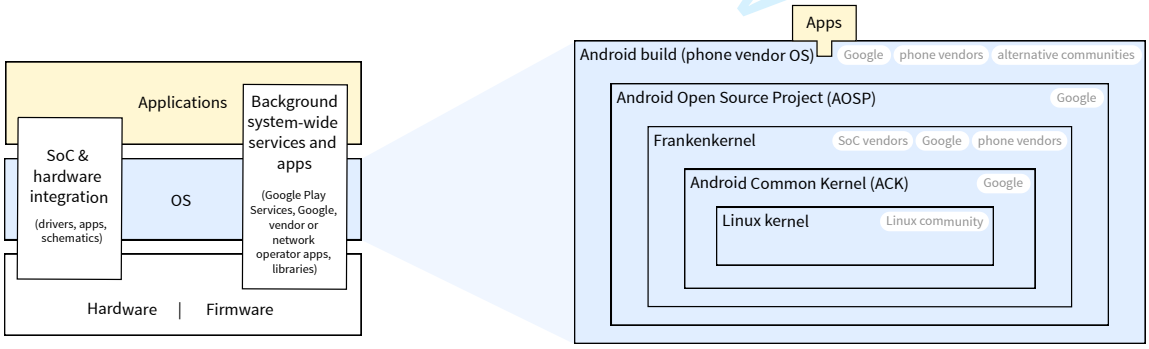

Fig. 2. Android software layers (left) and Zoom into the Android OS composition (right)

specific to the device, or in the application layer (to make the camera perform in a device-specific way, make brighter pictures, have special features, etc.). Without them, the device cannot function properly.

4.1.2 From a Linux kernel to an Android build. The Android installed on a smartphone is the result of multiple actors: from the Linux community to Google, SoC manufacturers, phone vendors, and network operators. In the following, we detail the various software artifacts involved in the production of Android as it is deployed (see Figure 2, right).

An Android build corresponds to a version of Android created for a specific device model. For each smartphone model, a specific Android is built and installed on it, which is ultimately the Android build experienced by end-users. These builds are all based on Android Open Source (AOSP) developed by Google, but diverge in what Marvin Wissfeld (participant 3) explained as follows:

> "*Each phone has its own OS, there is no universal installable image for Android like we are used to in desktop computing. The AOSP is open source, but we cannot deploy it on physical hardware, for that we need additional drivers or firmware, that are available only for the specific hardware the phone is made of, and are not open source in general. A custom ROM, an Android build for a device, is what you put on top of the AOSP plus drivers and firmware strictly needed to get the hardware running, and many of the services and applications that Google and manufacturers put on top of it. AOSP itself is not for real hardware, is not relevant to end users, it is running only on emulators.*"

At its core, Android is based on the free and open-source Linux kernel. A kernel is a basic fundamental software component at the core of an OS, providing important hardware functionalities and facilitating hardware and software interactions. Over 1000 contributors forming the Linux community contribute to each release, which involves over 8 million lines of code. Every 9 to 10 weeks a new stable mainline kernel is released and published on kernel.org. Usually once a year, a stable kernel is picked and designated as a long term support (LTS) kernel.

Google develops and releases a new version of AOSP every year. To do this, an LTS Linux kernel will be used by Google to create the Android Common Kernel (ACK) at the core of every new AOSP version to be released. An ACK is an LTS Linux kernel with extra code from other branches of the Linux kernel such as *"new Android features under development in the Linux community*", and *"Vendor/OEM*[12] *features that are useful for other ecosystem partners*"[13]. Any version of AOSP contains an ACK and all the necessary functionalities of an OS that make its specificity: in the case of Android the way the system interacts with hardware components (e.g. the touchscreen or the camera), how hardware components interact with each other (e.g. the battery with the CPU), how the user interacts with the system (e.g. through a custom UI layer).

Luca Weiss, an Android developer at Fairphone, describes how the ACK is then used[14]: "*Based on [an] ACK branch, the SoC manufacturers take it, add some support for their SoC on top, and then finally, device manufacturers get this code base and put their device-specific changes on top*". First SoC manufacturers develop their kernel by adding a large amount of SoC specific code on top of an ACK kernel, this ACK corresponding to a given Android AOSP version and a given LTS Linux kernel. For every device manufactured with a given SoC, vendors take this SoC's specific kernel and add to it the extra hardware-specific code and drivers (for the touchscreen, battery, camera, etc.). The resulting kernel is what is sometimes called a *vendor kernel*, or sometimes, "*frankenkernel*"—a term that we will discuss further below.

[12] OEM: Original Equipment Manufacturer
[13] From the AOSP official documentation, accessed on Jan. 13, 2025.
[14] Mainline Linux on Fairphone? Yes, please!, Capitole du Libre, Toulouse, Nov. 2023, last accessed in Dec. 2025.

On top of the vendor kernel, vendors take the corresponding AOSP version released by Google, usually add to it drivers needed by the phone components, creating an Android OS build able to run only on this specific phone device. In this phase, vendors also customize Android by adding their own user interfaces, system-wide services or apps as part of the transversal Background layer. When these phones are marketed by network providers, the latter also customize the Android system by adding their own user interface, system-wide services or applications. most of the time these changes (drivers, services and applications) are not open-source code.

The specific nature of the Android OS, consisting of one build per device, and how these builds accumulate specific software from various actors, is one of the first main findings of our study. We noticed that this is not something that is widely known or understood, even among developers or on specialized media. From a software perspective, smartphones are indeed quite different from personal computers, where the same generic OS (whether based on Linux or Windows) can be installed by the user regardless of specific key hardware components such as the motherboard, the processors, or the memory they hold. In Android smartphones, the SoCs, holding both key hardware and software components, seem to play a central role in the specificity of the OS builds, on top of which actors each add their own software layer of specific hardware features, services or applications.

4.2 Obsolescence in action

Given the Android build pipeline described in the previous section, we will now clarify how specific Android builds pose maintenance issues and can become obsolete in the ecosystem. In Figure 3, we illustrate this build process, from the Linux kernel to Android builds, for two specific smartphones: the Motorola Moto G7 and the Fairphone 3.

4.2.1 How lack of updates appears. Every year, Google releases a new version of Android based on two or three of the latest Linux LTS kernels and creates an ACK for each LTS kernel used. Each ACK will be the core of the new Android versions specific to a device. For example, Google Android OS version 9 was released both as version 9-4.4 (based on the ACK coming from LTS 4.4), and as version 9-4.9 (based on LTS 4.9). As we will explain later, maintaining or disrupting maintenance for one of these versions has an impact on the obsolescence of the devices that were put on the market with these Android versions. In Figure 3 we only detail the Android development process based on LTS 4.9 (marker A).

The Motorola Moto G7 and the Fairphone 3 were both put on the market in 2019, and were both shipped with the same Qualcomm Snapdragon 632 SoC. Both smartphones were running on Google's Android version 9-4.9 (marker B), and contained a kernel built by Qualcomm specifically for the Snapdragon 632 SoC in 2019 (marker C). The LTS kernel 4.9 of these Android builds, was released in 2016 by the Linux kernel development community.

When they were released, in 2019, the Android builds on both smartphones were based on a 3 years old Linux kernel. The LTS 4.9 had been released in 2016, and the Linux community defined its "end-of-life" (end of official support) in January 2023. According to A. Ferraris (participant 6), Google needs time to develop an ACK based on a given LTS kernel (one or two years), after which SoC manufacturers in turn need extra time to develop the SoC's specific kernel on top of this ACK (another one or two years).

Qualcomm, **the SoC manufacturer, never upgraded its kernel after the 4.9 LTS kernel**. Qualcomm only applied some of the security updates that LTS 4.9 received from the Linux community. Qualcomm applied these updates to its SoC kernel for only two years before stopping updates altogether in 2021. Motorola then applied these updates to its Moto G7 phone before stopping in 2021, two years after the phone's release. At that time, the Qualcomm frankenkernel was based on a five-year-old Linux kernel that had been upgraded five times to newer versions, each of which had been updated many times with bug fixes or security patches.

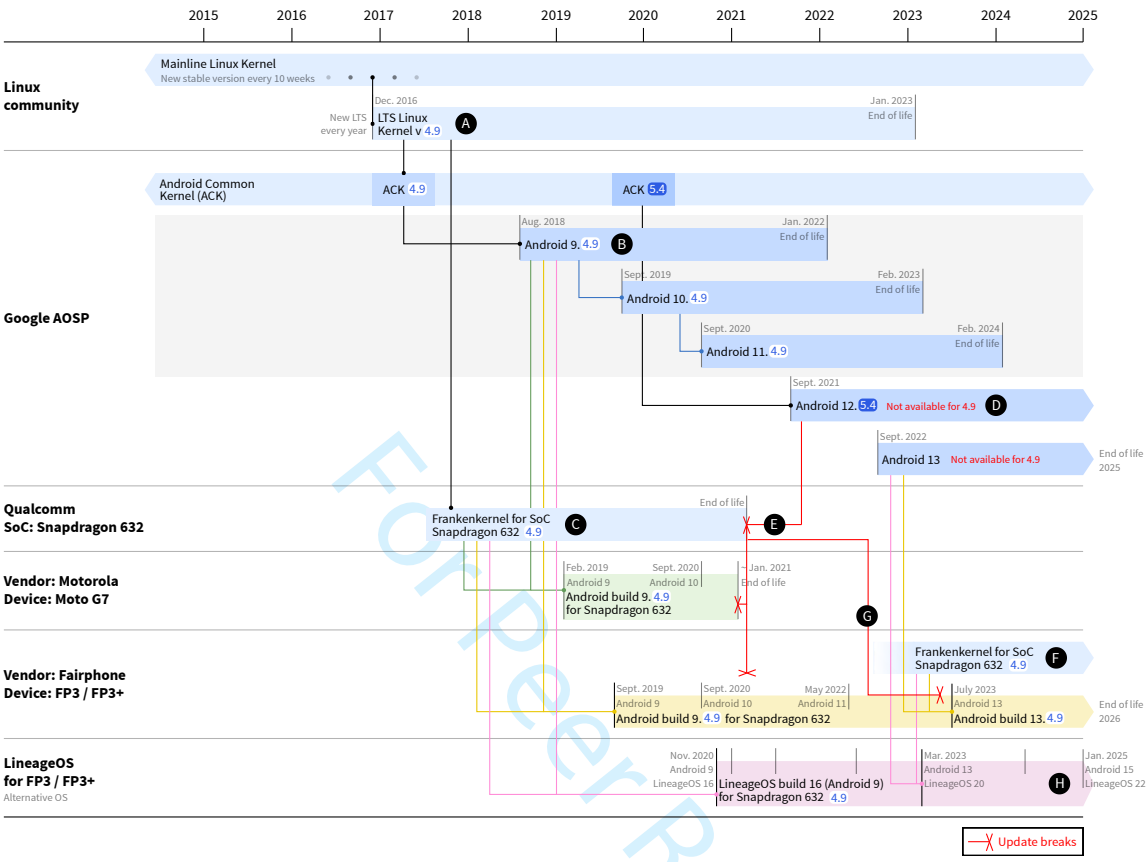

Fig. 3. Android builds for Fairphone 3 and Moto G7 phones with update breaks appearing at different points

These two observations reveal poor maintenance practices in the Android ecosystem: few software updates and early end of support. This is quite unusual when compared to development and maintenance practices of some of the most well-known operating systems derived from the Linux kernel in the Desktop world, such as Debian, Ubuntu, Fedora, Red Hat OS, etc. They generally closely follow the updates and upgrades of the latest LTS kernels, by implementing all bug fixes and security patches, as well as new features and developments contributed to the Linux kernel as hardware evolves. Study participants emphasized how alignment with LTS kernel releases facilitates frequent but small changes, as opposed to larger, more complicated changes, when updates are infrequent and occur after long periods. Luca Weiss from Fairphone explains in 14 when talking about Qualcomm's frankenkernel 4.9 for Snapdragon 632:

> " *The code that we got from Qualcomm [of frankenkernel 4.9] is about 2.5 million lines of difference compared to the 4.9 LTS kernel that is being released upstream. And it is about 18000 commits. This also shows why it is not really possible for a device manufacturer to rebase all of this 2.5 million lines and make them work.*"

"Rebasing" is a development practice consisting in applying changes made to a derived code (in this case the frankenkernels), to the original code from which the derivation occurs (here the LTS Linux kernel) by resolving code

conflicts, dis-functionalities and making sure everything works how it should. The biggest the changes between the derived code and the origin are, the more difficult the rebase becomes.

The changes made by Google to the LTS Linux kernel are also significant and not aligned with the LTS kernel, as explained by Arnaud Ferraris from the Debian and the Mobian mobile project:

> "*When Google makes the kernel for Android [ACK], it modifies the whole core part of the Linux kernel with the scheduler, memory management, power management, to make it compatible with the Android OS. Google tries to upstream this code so that it is available in the Linux kernel, so when the Linux kernel gets updated Google does not have to deal with it anymore, unless Google changes these parts. But the process of upstreaming patches into the kernel takes time: there will always be a first version, comments from the Linux kernel team, things that the Google developers had not thought of—the kernel maintainer will have their own Linux logic—and code will generally need to be modified several times before we gradually arrive at a final version. So, despite a genuine desire to do things upstream, we still have an ACK from Google that has a lot of downstream code when it is passed on to the SoC manufacturer. But, downstream code is code maintained only by Google. Every time the Linux kernel is updated, the downstream code must also be updated from Google, and then passed to SoC manufacturers once again. This code may become upstream at some point in the future. But at a given time, it is downstream code. Why? Because upstreaming takes time, and Google's pace is not that of the Linux community, they are not completely in sync, they have their own Android pace.*"

4.2.2 Upgrades lacking backward compatibility. Beside the lack of software update previously observed, we observe new breaking points that appear when Google upgrades, on a yearly basis, from one version of Android to another.

As illustrated in Figure 3, in 2021, two years after the release of the Moto G7 and Fairphone 3, **Google introduced kernel backward incompatibility in the new Android version 12, which was based on kernel ACK 5.4, and no longer on ACK 4.9.** (marker D). Android OS upgrades were no longer available for 4.9 kernels. This marked the end of the support from Qualcomm of its 4.9 frankenkernel for the Moto G7, the Fairphone 3 and all other phones holding this SoC (which was confirmed by P1). In 2021, Motorola had upgraded the Moto G7 only once, from Android 9 to version 10[15]. When Google stopped providing the 4.9 version when releasing Android 12, Qualcomm stopped updating the phone's frankenkernel to the ACK of Android 12 and Motorola stopped upgrading the phone to newer Android versions (marker E).

But this backward incompatibility did not make upgrades impossible. As Agnès Crepet (participant 1), head of the *Software longevity and IT* team at Fairphone told us, they managed to upgrade its OS on Fairphone 3, despite the lack of support. When Google released Android 12, and Qualcomm stopped updating the SoC's frankenkernel, Fairphone maintainers took over. Luca Weiss, who was then a volunteer developer on the LineageOS and PostmarketOS projects, two alternative community-driven mobile OSes, was hired in the team because of his knowledge of this specific SoC. The updating process of the abandoned Qualcomm SoC frankenkernel took time and was finalized after Android 13 was released, so the company decided to release an upgrade of Fairphone 3 directly to this new Android 13 in 2022 (marker F). This update was possible in great part because of the community work from the alternative OSes—PostmarketOS and LineageOS mostly—that had taken over maintenance work on Qualcomm's abandoned frankenkernel by applying security patches and functionalities needed to keep compatibility to the Linux kernel or Android newer versions whenever possible. Their work is made available to the community in open-source repositories, while device trees—data

[15] See Motorola's update announcement followed by specialized media Techradar analysis, December 2020, archived, last accessed in Dec. 2025.

structures describing all the device components needed by the kernel to use and manage those components—are shared together with available documentation. Various discussion spaces for users and developers to interact also exist[16]. Based on this work, the newly formed Software Longevity team of seven people at Fairphone succeeded in updating an Android build after both Google and the SoC manufacturer had stopped doing so.

4.2.3 Upgrades introducing new features that break compatibility. One year later, the release of Android 14 brought new breakdowns in updates. As Weiss explains in 14: *"With Android 14 released in 2023, Google is introducing some new features that are not present at all in kernel 4.9. AOSP is dropping support for these kernel versions sort of aggressively, as they do not support any of the new features that [Google] wants to use in Android 14*". In July 2024, Fairphone announced to its users [17] that the FP3 phone would not be upgraded to Android 14 and beyond anymore (marker G):

> "*We invested considerable time and resources into exploring ways to integrate the new Android system with the existing kernel, and even contemplated upgrading the Linux kernel itself. We also engaged in discussions with Google Android Engineering. The legacy Linux kernel (4.9) used in the Fairphone 3 [...] would not support Android 14 at all.*"

FP3 would thus end software support in 2026, but would nevertheless continue to receive security updates on Android 13 until then. With 7 years OS support, this makes the Fairphone 3 one of the longest supported Google certified Android OS phones, and as some specialized media say, the only phone manufacturer that maintains the OS after SoC support shut down[18].

We noticed that the LineageOS community managed to circumvent the above incompatibilities and keep the Fairphone 3 updated after Qualcomm, Google, and even Fairphone stopped supporting it. In November 2025 LineageOS with Android 15 could still run on Fairphone 3 (marker H). We discuss the circumvention strategies allowing longer maintenance from both Fairphone and LineageOS, but also their limits, in section 4.4 below.

4.3 Disruptions to maintenance practices in the Android code flow

Based on our interviews, conferences and documentation analysis, we identified issues related to how changes to code developed by each actor are propagated within the Android ecosystem. The layers in the Android operating system shown previously in Figure 2, appear as code flows circulating from one actor to another. The breaking points, which we described previously, occur at different levels and affect code maintenance differently. Figure 4 illustrates these code flows, or lack thereof, and the resulting maintenance breakpoints.

4.3.1 Divergent maintenance practices inhibiting updates and creating technical debt. In our interview, Arnaud Ferraris (participant 6) from the Mobian project, a Linux derived OS for mobile, talks about the great amount of added code that makes maintaining difficult in Android:

> "*This ACK by Google with downstream code in it passes into the hands of the SoC manufacturer, they add their own system drivers to manage hardware, then it is passed on to the phone manufacturer who will add other drivers. So we already have three levels of forked kernels, and when we finally look at the cumulative amount of changes that this represents, it's huge. If I take the example of the OnePlus 6 phone that we maintain for Mobian, this was 5 million lines of code compared to the Linux kernel used.*"

[16] See the LineageOS user and developer wiki, the blog that announces the new development releases and discusses changes. The same can be found for PostmarketOS, and LineageOS based OSes that maintain Fairphone 3 such as LineageOS for microG or e/OS.
[17] Announcement of end of support for Fairphone 3 in July 2024, last accessed in Dec. 2025.
[18] See *Fairphone 3 gets seven years of updates besting every other OEM*, Ars Technica, July 2023, last accessed in Jan. 2025.

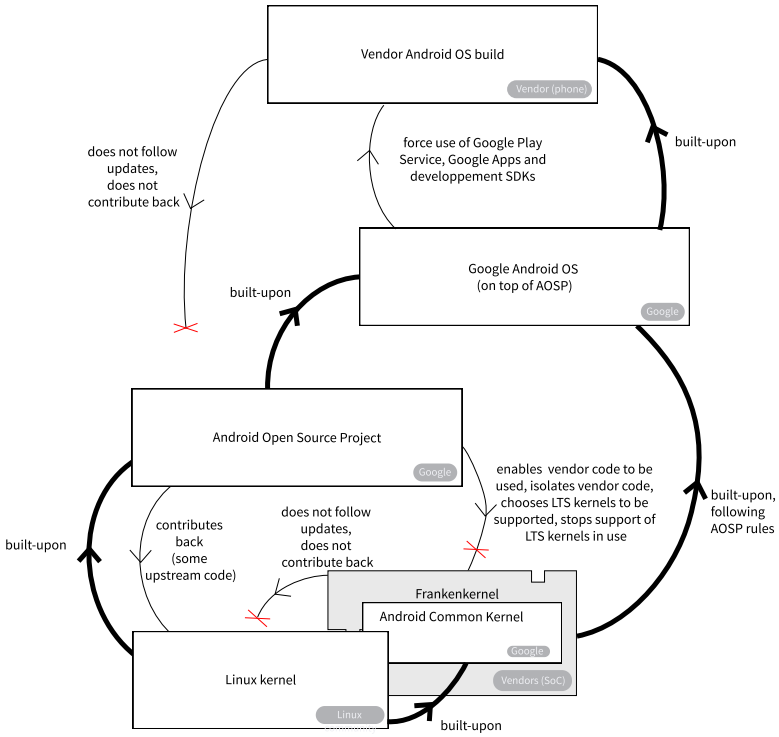

Fig. 4. The development flow of an Android build, red crosses indicates a lack of contribution to the original code-base.

In free and open source software development, *mainlining* is an important wide-spread coding and maintenance practice that refers to the integration of new developments into the main *mainline* source code branch in the project's repository. When developing upon the Linux kernel, mainlining is a two sided development flow: (1) systems derived from Linux follow the mainline Linux kernel development, and also (2) contribute back code to the Linux kernel as often as possible, through a process known as *upstreaming*.

Ferraris further explains the different development practices between phone manufacturers and free software communities:

> "*There is a big difference most of the time between the kernels from phone or SoC manufacturers and kernels maintained by a free software community: manufacturers will create a software mess that suits their needs, while free software communities will make sure to make the minimum changes necessary for it to work and be accepted into the upstream Linux kernel. When developing we always try to keep this notion of upstreaming in mind [...] for the sake of sustainability and for the comfort of everyone, both users and developers, it's better for the changes to be upstream, and we try to think in advance how to make the changes we need so that they can be accepted in the kernel.*"

Upstreaming enables mobile OS actors following the mainline Linux kernel, such as PostmarketOS and Mobian, to directly benefit from the various contributions to the Linux kernel. It also enables kernel maintainers to handle

upstreamed contributions in manageable bits, isolating problems, using simpler tests and dealing with them on a daily basis. Other Android based OS actors, such as LineageOS, GrapheneOS, Replicant, even if they do not mainline the Linux kernel, follow upstreaming practices as much as they can, and collaborate in understanding and developing free drivers for phone components for which the code is neither public nor documented.

As illustrated in 4.2.1, in the Android ecosystem, Google creates the ACK from the Linux kernel without mainlining it and only partially upstreaming code into it. SoC and phone manufacturers use these ACK to create the frankenkernels by adding code that is not designed to be contributed back to the Linux community: assembled from various internal codes, designed to meet their internal development needs and undocumented for external use, hence the prefix "franken". As the Linux mainline kernel evolves, the difference with SoC frankenkernels continues to grow, making it increasingly challenging to share and propagate code between them over time, as Ferraris explains:

> " *It is code that is written internally, never submitted to the community. A manufacturer will have dozens of very different hardware references and generally, almost always, wants to have a single code base for all of them. As a result, they develop extremely complicated code filled with switches, if statements, internal compilation options used to enable or disable certain parts depending on their different hardware. What's more, there is no public documentation for any of this code. Sometimes things seem magical when you read the code. It makes sense to manufacturers, but when we read it, because we don't have any documentation, we don't understand at all why they're doing it, and we wouldn't be able to explain it to the Linux kernel maintainer, or make it work, let alone provide a Linux kernel driver with it. This also makes reverse engineering complex.* "

The use of nested if statements and switches is a known cause of what in computer programming is called *spaghetti code*, code that is fragmented, entangled, and thus difficult to understand and to maintain [43].

The divergence in development strategies outlined above, has created historical tensions in maintenance practices between FOSS communities such as the Linux kernel community, and hardware manufacturers such as phone vendors in our case. The Linux community aims to build a kernel that runs on as much hardware as possible and for as long as possible, whether that hardware is old or new. Manufacturers focus on developing code for new hardware by adding to their own existing code base. By using the Linux kernel, which implements the core functionalities of operating systems, they reduce the costs and time required for launching them [30]. But they show little interest in maintaining this code or contributing code back to the kernel in order to keep their hardware updated over time. The responsibility for code maintenance is transferred from manufacturers to the Linux kernel community or to the FOSS mobile communities maintaining old hardware. This shifted maintenance constitutes what is called technical debt in software development [15]. In software using the Linux kernel, the lack or delay of upstreaming, as well as obfuscated and undocumented code, are common factor in technical debt [22].

4.3.2 Maintenance barriers: software obfuscation and circumvention strategies. As maintenance shifts from vendors to the FOSS communities, access to hardware specifications and documentation becomes important. Luca Weiss, an Android developer at Fairphone, explained during a public conference, how access to private documentation shared by Qualcomm exclusively with phone vendors, enabled him to update the Fairphone/Spapdragon 632 frankenkernel from Android 11 to Android 13, and share it with the open source community:

> " *Having access to secret documents now that I am working at Fairphone definitely makes things easier. Schematics, data sheets and documentation are confidential material. Schematics make it easier to build*

> *the device trees, or hardware descriptions that the kernel needs, because some things are not obvious from the way the vendor-provided kernel has this written. Data sheets make it easier to write new drivers in case it's necessary.* "

Additionally, hardware and smartphone vendors place a great emphasis on keeping code private whenever possible, by using intellectual property. The copyleft Gnu Public Licence (GPL) of the Linux kernel requires manufacturers to provide the code from the frankenkernels under GPL-compliant licences. As mentioned previously, this code is provided by vendors, but being too chunky, complex and undocumented, it is of little or no help for the open-source community in maintaining it. For phone components source code is never available, vendor drivers being almost always proprietary closed code. As articulated by participant 2, cameras for instance play an important role in vendor marketing strategies: differentiating devices on camera performance and features are key selling points in the smartphone market. Since software plays a key role in camera performance and their distinguishing features, vendors keep it private and well protected. Ferraris confirms:

> " *Beyond the SoC in every phone there will be a screen that will need a very specific driver, the same goes for touch panels, and cameras are yet another problem because there is a lot of intellectual property involved in them. In the manufacturers kernels there is nothing to be found about these, everything is in the Android user space or elsewhere, in proprietary binary code form. How it works is well hidden.* "

Because of this opaque nature, the binary code and drivers coming from manufacturers is often called *blobs* among Linux kernel developers and open-source communities. Trying to have the less blobs possible, by replacing them with functional open-source code, is an important issue, as these blobs do complicate maintenance when changes occur in the OS or in the kernels, or when security issues involving them arise.

A tension exists between Google and the Linux kernel community on these code blobs. Google attempts to create safe spaces for vendors directly into the Linux kernel structure, and to remove part or all of them from its Android layers. Linux kernel developers that place great emphasis on open source code are reluctant to create these safe spaces for vendors to put their binary code. They fear that instead of pushing vendors to open-up their code or at least contribute to the kernel community with documentation, these safe-spaces will push them to continue non-transparent development practices, which are precisely what complicates their maintenance and updating work.

This tension is clearly illustrated in a meeting report from a two-days workshop within the Linux kernel team dealing with camera hardware [52] involving kernel developers and Google employees. The report shows that Google contributes back, upstreaming code to the Linux kernel camera modules. This is useful both for the Linux kernel and for Google in the development of Android and Chrome OS, both based on Linux. During the meeting Google tries to push its idea to create a vendor specific code space inside the Linux kernel camera framework, where vendors can upstream blobs and keep the intellectual property of their devices safe. The reaction of the Linux kernel team maintainers clearly illustrates the tension: [52]: *"Upstreaming a driver requires opening up the driver interfaces [in Linux kernel practices]. There appears to be near-unanimous consensus on this apart from Google. It is not an option [for the Linux community] to upstream a driver that has support for undocumented closed features. Basically [Linux kernel] maintainers can not put their name on something that contains unverifiable (for them) and unusable (by all except the vendor) features."*

Extremely large commits, spaghetti code and blobs are considered anti-patterns that negatively impact program comprehension [43], specifically during maintaining tasks [1], while program comprehension and documentation are considered central to effective software evolution and maintenance practices in software engineering studies [3, 57]. As illustrated above, all of these patterns appear in the Android OS ecosystem.

4.3.3 Google's two-tier Android development strategy: reinforcing fragmentation when trying to address it. Google has recognized the persistent Android OS fragmentation problem, (i.e. multiple versions of the Android operating system present on active devices at any given moment), with multiple active Android devices not being upgraded to the latest Android OS version. The company has implemented several strategies to facilitate updates in order to reduce this fragmentation. In 2017, when launching project Treble[19], Google Android development team declares: *"we've consistently heard from our device-maker partners that updating existing devices to a new version of Android is incredibly time consuming and costly*". Treble consists in implementing a modular base for the low-level architecture of the OS in order to better isolate vendor code. Later in 2020, Google also launches the Generic Kernel Image (GKI) project to address kernel fragmentation by *"moving SoC and board support out of the core kernel into loadable vendor modules so they can be updated independently."*[20].

By isolating vendor code both in the ACK and in AOSP, Treble and GKI attempt to help Android actors offer faster roll-out updates. This technical isolation is welcomed from vendors willing to maintain their phones, as Agnès Crepet from Fairphone (participant 1) told us. But it does not enforce changes to their updating policies and coding practices. Regardless of the underlying technical stack improvement, the obfuscated vendor code remains, and updates still depend on the willingness of vendors in maintaining their code and devices. Data from 2025 attests that fragmentation is still pervasive (see Figure 1) and Android OS updates have not significantly improved on user devices since 2017.

The vendor code isolation also facilitates Google's own work of developing and maintaining AOSP as vendor blobs interfere less with the rest of Android. But the approach here differs from that of the Linux kernel community on blobs that consists in integrating them as much as possible in the system, in order to facilitate maintenance. Google has no need for vendor blobs to be completely open-source, documented or intelligible. By isolating vendor space, Google can continue developing AOSP while leaving its partner vendors free to "blobify" and maintain secrecy over their specific hardware code.

This allows for another shift in maintenance responsibility, from Google to vendors. But vendors' lack of maintenance remains unaddressed. And it is precisely the lack of maintenance of vendor code—Android OS builds specific to every device that are being little or not maintained—that leads to the Android fragmentation situation.

4.3.4 Google's double-standard Android updates strategy: from deprecation to forced dependencies. In the Applications and Background services of Android (Figure 2), updates are fine-tuned to serve Google's business model and interests. As Marvin Wissfeld (participant 3), that develops microG, a free open-source implementation of Google Play Services, notes:

> "*[Google] stopped updating apps on AOSP, the clock app, the messaging app. They update only in the sense that they make sure they are still compatible, but do not add new features. They fully update only the proprietary versions coming with the Google-licensed Android. In the Google messaging app they integrated their own messaging proprietary system. They leave AOSP behind, add small features to the Google versions of the apps, and put on top of them a dependency on Google Play Services. ... [Manufacturers] take the proprietary Google apps requiring Google Play Services.*"

As an AOSP derived OS, LineageOS development closely follows that of Android: when Google releases a new Android version, LineageOS releases a new complying version. Google's lack of updates in AOSP are quite silent, as official Android release announcements do not usually mention them. By analysing LineageOS new version release

[19] Here comes Treble, 2017, Google Developers blog, last accessed in Feb. 2026.
[20] Generic Kernel Image documentation, last accessed in Feb. 2026.

announcements (called changelogs), one can better spot this lack of maintenance and sometimes complete deprecation of essential Android apps. For example, in the Android 14 release in 2023, Google deprecated the Dialer and the Messaging applications in AOSP, with a discrete message revealed by specialized media[21], written in the repository of the source code for the deprecated apps, saying: *"This app is not actively supported and the source is only available as a reference. This project will be removed from the source manifest sometime in the future"*.

In this case, a shift of maintenance towards open-source communities is clearly at play. When the corresponding LineageOS (version 21) was released some months later, it announced that it had taken over the maintenance and further development of the deprecated apps:[22]:

> "*Since AOSP deprecated the Dialer, we have taken over the code base and did heavy cleanups, updating to newer standards (AndroidX) and redesigning[...]. While Messaging was also deprecated by AOSP, at least the Contacts app was not. Nonetheless we gave both of them an overhaul and made them also follow the system colors and look more integrated.*"

Google's strategy of selective updates in AOSP is accompanied by contractual agreements between Google and its vendor partners. These agreements ensure that vendors systematically install an Android version with Google proprietary services and applications. Such contracts frequently stipulate the exclusive use of Google applications, for instance by prohibiting vendors from installing another search engine. In some instances, the contracts may also preclude vendors from marketing devices using an alternative Android-based operating system. As elaborated in 5.3, these practices have been the subject of lawsuits and rulings against Google for abuse of dominant position in several countries worldwide.

As Wissfeld notes, Google Play Services integrate telemetry, gathering user data used for advertisement purpose, advertisement that is injected back in the applications via these same services[23]. The Google Play Services also serve Google's business interests as one of the dominant actors of web and mobile advertisement[24].

4.3.5 How forced dependencies inhibit sustainable maintenance and resilience. When the use of Google applications and services essential to the Android OS is not contractually enforced, it is *de facto* ensured. Wissfeld (participant 3) told us that he has seen these services evolve and become more and more present in the Android ecosystem. While essential system apps get less updates in the AOSP version, they gain Google Play Services dependency in their final official Android build.

Indeed, the most widely used Android development environments and libraries, such as Android Studio or the Firebase platform, both developed by Google, integrate unavoidable dependencies to Google Play Services and Google's Firebase Cloud Messaging. As a result, most Android application developers integrate these services in their apps. Google's Firebase Cloud Messaging (FCM), formerly known as Google Cloud Messaging (GCM), implements the push notification system that enables applications, such as messaging or emailing ones, to use Google servers for sending notification messages to the apps on the user phones (e.g. each time a new message arrives). Given that one of the essential uses of a smartphone is messaging, the push messaging systems are crucial in mobile ecosystems. The technical implementation of FCM is dependant on Google Play Services: in order for FCM to function, Google Play Services have to be present in the user's phone. However, neither FCM nor Google Play Services are implemented in AOSP. As a

[21] Google further guts the AOSP by deprecating the dialer and messaging apps, OSnews, June 13, 2023, last accessed in Dec. 2026.
[22] LineageOS version 21 announcing new Android 14 release in Changelog 28, February 14n 2024 last accessed in Jan. 2026.
[23] See Ads Safety, the user data telemetry and advertisement injection service integrated in Google Play Services, last accessed in January 2026.
[24] See Google Ads program for mobile apps, Google.com, last accessed in January, 2026.

result, most vendors market Android phones with Google apps and Play Services, and almost Android apps integrate the Play Services and use the Google FCM system.

Moreover, the Google Android device certification system, the *Play Protect Certification*, a set of tools for testing Android devices offered by Google to phone vendors and users in order to label their Android phone as *"certified by Google"*, includes mandatory checks of the presence and activation of Google Play Services. Mandatory for device vendors having a contractual agreement with Google, this certification mixes AOSP compatibility tests with Google services and apps tests. Moreover, it is marketed by Google as a safety guarantee for Android phones, while its absence is presented as a severe security issue in Google's communication[25]:

> "*Devices that aren't Play Protect certified may not be secure [...] may not get Android system updates or app updates. Google apps on devices that aren't Play Protect certified aren't licensed and aren't real Google apps. Apps and features on devices without Play Protect certification may not work correctly. Data on devices without Play Protect certification may not have a secure backup.*"

Thus, certification becomes another way of enforcing Google dependencies in practice, and presenting their lack as a severe security issue.

In alternative Android-based OSes such as LineageOS, LineageOS for microG, or e/OS (a LineageOS derivative OS), dependencies from Google are partially removed via use of microG, that reimplements the Google Play Services, removes telemetry and advertisement, but still allows installed apps to use the Google servers via FCM. Open and standardized alternatives that allow to remove all Google dependencies and keep functional systems have been developed in recent years. One of them is UnifiedPush, an open-source system for push notifications in mobile ecosystems implementing the WebPush IETF open standards (RFC 8030, RFC 8291 and RFC 8292)[26], that has been implemented by various open-source systems such as Element, Conversations, Nextcloud, Mastodon, KDE, Mozilla. Simon Gougeon (participant 12), creator and main developer of UnifiedPush, explained to us how using an open standard for push notifications offers resilience in the long term:

> " *The Google push notification system is centralized, fully depends on Google. Huaiwei in China has its own centralized push notification system. In countries or situations where Google, Huaiwei, or other actors are not present, or their presence can change for economical or geopolitical reasons, UnifiedPush can provide standardised push notifications for Android phones. Also, in a context of climate crisis, if a region loses its access to the Internet, Google's push notification system is not available anymore. If a regional network takes is implemented until the global Internet access is restored, UnifiedPush can quickly be deployed in it and help keep phones connected. This is the advantage of open decentralized standards, they have resilience and can take over when centralized services are not available, not desirable or have failed. "*

Following our interview, Gougean published a retrospective article on UnifiedPush, developing on the issues he had discussed with us[27]: *"When a service is controlled by a single entity, nothing can be done when they consider your device too old to be supported"*. For him, services as fundamental as push notifications in mobile systems should be implemented inside the main Android code as open standard APIs. Reflecting on the future of UnifiedPush he says *"The best thing that could happen to UnifiedPush on Android [...] would be for it to no longer exist. If Google gives us in Android*

[25] Google Android Help Center, last accessed in Dec, 2025

[26] IETF: International Engineering Task Force, the main international technical standards organization for the Internet. See IETF WebPush push notifications standard documents, last accessed in Jan. 2026.

[27] Simon Gougean, 5 years of UnifiedPush, F-Droid.org, Jan. 8, 2026, last accessed in Feb. 2026.

a system API to let the user define their push service we would not need UnifiedPush anymore. [...] Hopefully, working on UnifiedPush can push in that direction by increasing the demand, and highlighting the need".

4.4 Success and limits of obsolescence circumvention strategies

The obsolescence circumvention strategies and maintenance efforts do not come without drawbacks for the FOSS communities and some phone vendors that put effort in it. Fairphone and LineageOS achieved to maintain the Fairphone 3 after Qualcomm abandoned its frankenkernel and Google removed support for Android OS 12, but faced difficulties in implementing these update strategies on the long run.

Update problems became particularly important with the release of Android 12 in 2021, which broke the backward compatibility with kernel 4.9 and below. One reason is that *eBPF*, a tool to manage network traffic in the Android kernels, replaced the old *iptables*. Updating devices running on kernel versions 4.9 *"proved challenging due to the sheer number of commits and structure changes*", while for devices using older kernels the upgrade was not possible anymore[28].

The update process of taking parts from a newer version of a software system and porting them to an older version of the same software is called *backporting*. It is a common practice when preferred maintenance solutions like *upstreaming* or *mainlining*, are not taking place. Backport code is often applied as patches in order to incorporate changes into an old code-base (the term *patching* is also used).

To maintain the Fairphone 3, Fairphone and LineageOS backported changes introduced in Android 12 to its old frankenkernel (based on kernel 4.9), which both Qualcomm and Google had stopped supporting. They also included security patches from the Linux kernel team, which Qualcomm had stopped offering. After the release of Android 15, Fairphone announced[29] that it would stop offering Android upgrades to Fairphone 3, due to the eBPF structural changes that became too complicated to backport. Yet LineageOS with Android 15 can still run on the Fairphone 3, thanks to backports provided by the open-source communities, even after Fairphone stopped its upgrades. But, for vendors aiming at the Google certified phones, like the Fairphone does, the stricter rules and controls of Google's *Play Protect Certification*, make things more complicated than for alternative actors, used to release OSes with sometime bugs and missing features, and to fix them in subsequent updates when possible.

Nevertheless, as Google adds new features with each Android release, backports become increasingly difficult to maintain. For developers in the open-source communities we interviewed, backports are seen as a temporary and inadequate solutions to the update problem. Because backports are code that is neither mainlined nor upstreamed, but comes as patches of code applied at a given moment, when the ecosystem evolves, patches need to be updated independently every time. Thus, patches face the same update issues as frankenkernels from vendors do: they are difficult to maintain, especially when they come in big chunks of code.

Some alternative mobile OS projects such as PostmarketOS or Mobian, choose to completely detach themselves from Android and its frankenkernels by directly following the Linux mainline kernel development for mobile phones. Older devices are regularly abandoned by LineageOS maintainers, while mainline projects like PostmarketOS and Mobian succeed in maintaining very few devices, but having a much more reliable maintenance system because mainlining and minimization of patching is here a primary focus. But all of these communities in fine face the same frankenkernel and patching problems. As smartphones grow older and new hardware is released, the burden of kernel maintenance grows on them, while the number of users and maintainers for older devices shrinks, thus decreasing the possibility to sustainably maintain code.

[28] LineageOS 19 (corresponding to Android 12) new release announcement changelog, April 2022, last accessed in Jan. 2026.
[29] A post on the Fairphone forum announcing the first Android 14-based build of LineageOS for Fairphone 3, last accessed in Dec. 2025.

5 Discussion and perspectives

Our results show that the complex Android OS production pipeline generates frictions and lacks of incentives to support maintenance and upgrades in the ecosystem. We reflect now on the values and macro forces shaping this situation.

5.1 On the various types of software maintenance and their role in obsolescence

Towards the end of this study, we realized that the act of updating software is not or little questioned per se by the actors of the Android OS ecosystem. Our informants all assumed that updates are a necessity and that their absence is what causes obsolescence. This is what is defined as Lehman's law of continuing change [28]: "a program that is used undergoes continual change or becomes progressively less useful."

But do updates always respond to a need to mitigate obsolescence and make devices last longer in a useful and safe way ? We notice that each actor had its own way of approaching and envisioning updates and that all had different rhythms. It starts with the continuous development at the Linux kernel or derived OSes that follows, willingly or unwillingly, the endless changes in the hardware market. Then comes the yearly upgrades of Android that sometimes introduce new features, or new development behaviors that break retro-compatibility or make device updates more difficult. From phone vendors we observed few scattered updates before abrupt stops. Some alternative actors had a continuous maintenance process (PostmarketOS, Mobian), while others responded to breakdowns by attempting various forms of repairs or standardisation of practices (Fairphone, LineageOS, microG or UnifiedPush).

Both the diverging nature of updates as well as their different timing among actors of the Android ecosystem, play a central role in the fragmentation. Also, it remains unclear to what extent enforcing forms of backward compatibility could enable smartphones to remain functional without updates, as their broader software environment evolves.

Nevertheless, updates should at least be considered as ambiguous regarding the role that they play in maintenance. Updates are not necessarily only acts of maintenance and care for the devices and the ecosystem. They can and do also trigger incompatibilities and obsolescence. While updates are generally presented as a way to avoid obsolescence, updates from an actor become the obsolescence of another. Maintaining then becomes the obligation to cope with unwanted updates or the act of feeding the "monster" that the software to be maintained has become.

5.2 Values and choices in OS production and maintenance

From these different types of software maintenance we observe that "official" Android providers (Google, phone vendors) and alternative Android or non Android mobile providers define software quality in different ways, according to divergent values and interests. These different values then lead to different temporalities in producing or maintaining code, and different software quality criteria.

5.2.1 Different values lead to different code quality criteria. Vendors and chipset manufacturers value putting new hardware products on the market at a frequent pace (every 6 months or every year) and have limited incentives to update their products. The drivers, frankenkernels and software they produce are mostly oriented towards new devices. Once the device is on the market, they do not offer clear update policies for consumers, perform little updates and the end of support of old devices comes silently and quite often as quickly as two years after release. The recent European regulation (EU) 2023/1670 laying down ecodesign requirements for smartphones among other devices [30], in effect since June 2025, enforces OS security updates and upgrades for at least 5 years after the product has been released. How this

[30] European Regulation (EU) 2023/1670 summary, last accessed in May 2025.

directive will play out in reality, how smartphone vendors will implement it in practice, and how monitoring will be carried out, remains to be seen.

The Linux and alternative OS actors value open-source code, documented development practices, code longevity, and even hacking techniques, considering them legitimate. For them, updates and maintenance are continuous processes, following coding practices that foster collaboration and ease of updating. Alternative Android or mainline-Linux kernel communities place a greater emphasis on running on old, no longer supported architectures and having control on the software stack.

Google values its control over the ecosystem. Its attitude towards the various actors changes during the course of the study. In May 2025, the company announced[31] that Android OS would be subsequently developed internally in a fully private and closed-source way, before the code is pushed to its public branches once development is finished. This unilateral decision of closing development raised many concerns: for alternative OS actors maintenance becomes even more difficult as they can no longer perform maintenance as a continuous process[32].

5.2.2 Variations in the temporalities of code production and maintenance. These diverging values affect the temporality of code production and maintenance practices among actors. Alternative actors are efficient in updating and maintaining, while developing functionalities or community collaboration methods are slow processes requiring much more effort. Vendors have slow maintenance and update rates, for short times, while often producing new frankenkernels for new devices.

5.2.3 Variations in code quality criteria. Code quality is also considered differently. Google and phone vendors will put greater efforts on user experience and standardisation. They will only green-light OS updates that fully pass functional and feature oriented quality tests at release time on every new device, while leaving aside maintenance work.

Alternative OS will favor code openness, privacy, security, sometimes accepting a lack of functionality (e.g. frequent issues with GPS or cameras on LineageOS, or with audio and battery gauge on postMarketOS) while being fully transparent about this. Code quality in these communities will rather relate to the ability to facilitate collaboration and long term maintenance. The users are part of the development and maintenance process: they are expected to test and report bugs as developers fix them and push updates. Infrastructure and tools are provided for this functioning: forums where users and developers can interact and react on bugs and updates, as well as more technical tools for bug reporting or code contributions into the project repositories.

5.3 Power in the Android ecosystem

Google plays a central role in the Android Ecosystem. It owns the trademark and what can be called Android, it controls AOSP, its code and coding process, as well as the proprietary services and the core applications present on most Android phones. It also licenses the Android name and logo to manufacturers through the Android Compatibility Program (ACP)[33]. Android phone manufacturers that want to license Google's apps and services, are required by Google to enter an agreement called the Android Compatibility Commitment (ACC) [40, 41]. Previously called, almost ironically, the Anti-Fragmentation Agreement (AFA), it obliged vendors not to distribute any device based on an alternative Android OS alongside devices running on Google-Android. An antitrust legal case against Google by the European Commission (EC) in 2018[34], deemed it to be anti-competitive and to hinder the development of Android alternative

[31] Exclusive: Google will develop the Android OS fully in private, Android Authority, 26 May 2025, last accessed in Jan. 2026
[32] AOSP isn't dead, but Google just landed a huge blow to custom ROM developers, Android Authority, 12 June 2025, last accessed in Dec. 2025
[33] See Android Brand guidelines and Compatibility Program, last accessed in Feb. 2025.
[34] European Commission case against Google, last accessed in Feb. 2025.

OSes [49]. According to the EC conclusions, these tying practices consolidate Google's dominance and abuse of power in the Android ecosystem and exploit status quo-bias from users who are tied to Google Apps on everyday smartphone activities [36]. Similar antitrust infringement cases against Google have occurred in the US [44], in UK and more recently in Japan[35].

As a consequence, Google replaced AFA with ACC in Europe and vendors could then distribute alternative Android OSes on all of their devices. But they still have to follow the ACC guideline stating that only devices that signed the agreements can use and display the term *"Android"*, a registered trademark of Google[36]. These cases and changes in agreements between vendors and Google did change the situation a little in some regions. The vast majority of vendors continue to offer Google-equipped only Android phones. But some small phone vendors now offer, alongside the traditional Google-equipped Android phones, an alternative version with preinstalled de-Googled OSes: Fairphone since version 3 comes also with e/OS, Shift phones come also with ShiftOS L or e/OS, HIROH comes only with e/OS, to name a few.

According to Google [40], AFA and ACC are responses to the threat of incompatibility or fragmentation to Android. But our study suggests quite the opposite. Alternative Android OSes are concerned with maintaining updated devices, developing strategies of remediation to counter the lack of updates or maintenance largely responsible for the fragmentation problem. As for maintenance and updates, the definition of fragmentation is a matter of perspective, which varies depending on the point of view and values considered.

The same goes for power. SoC manufacturers, phone vendors and network providers have agency. The case of Fairphone demonstrates that vendors with limited resources can maintain Android builds twice as long as average, when they are willing to. As for Google, as repeatedly demonstrated by the various antitrust cases worldwide, it is able to exert power over vendors when it directly serves its interests. But when it comes to enforcing updates, Google's approach is much less coercive. Google ensures that the ecosystem provides a safe environment for vendors, by protecting their intellectual property and traditional production methods, and relies on their goodwill for maintenance. Meanwhile, the incentives to integrate Google services and applications into the OS are much stronger.

5.4 On the many forms of openness

Google has always emphasized the open nature of Android. Google acquired Android Inc. together with its developers and founders, in July 2005, and in November 2007, announced[37] the first Android platform and OS version. The announcement also stated that Android development would be handled by the Open Handset Alliance[38], a consortium of many international phone vendors and network providers, led by Google. Here is how they were presented in 2007: *"the first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications – all of the software to run a mobile phone, but without the proprietary obstacles that have hindered mobile innovation. [...] We hope to enable an open ecosystem for the mobile world by creating a standard, open mobile software platform. We think the result will ultimately be a better and faster pace for innovation that will give mobile customers unforeseen applications and capabilities. [...] Our goals must be independent of device or even platform"*.

Our work shows that this openness is confusing in the Android ecosystem. Android is indeed based on open-source code developed in FLOSS communities, Google also produces some open-source code at the AOSP level, but inside and on top of them, lie layers of proprietary closed-source code, undocumented hardware or software functionalities,

[35] On the Japan Fair Trade Commission's Google Decision: Some Early Reflections, Sangyun Lee, Kyoto University, April 2025, last accessed in Nov. 2025.
[36] Google starts blocking uncertified Android devices from logging in, Ron Amadeo, 2018 Ars Technica, last accessed in Jan. 2025.
[37] Google Blog: Where's my Gphone?, last accessed in Oct. 2024.
[38] Wikipedia - Open Handset Alliance, last accessed in March 2024.

missing policies, and forced behaviors. The resulting OS is a "frankenware" system that the FLOSS communities have trouble understanding and maintaining. The narrative that is also pushed by Google is that of the security as a reason for OS upgrades and big changes in behaviour. Recent scholarship on security by Korn and Wagenknecht propose to consider 'security research' as an ambivalent form of repair and maintenance [27]. They examine how frictions arise in what they call *"social arena of repair"*, and how agentivity is distributed asymmetrically among actors, some of which possess the privilege to repair (industrial actors) while others (hacker and security activists) do not.

FLOSS communities have formed their identity on the notion of code openness, reflecting on it not only as a technical value, but as a form of political activism for building digital commons, considering them through the same lens as important public infrastructure, such as public roads, a metaphor used by Eghbal in her work on the making and maintenance of open source software [11].

But as FLOSS projects grow and become embedded in complex ecosystems like Android, involving Big Tech and other industry actors, they become part of what can be seen as what Ekbia and Nardi have called "heteromation" [13]: a set of practices of industrial actors to extract economic value from under-compensated or free labor in computer-mediated networks. Moreover, this does not only conflict with values of openness towards digital commons in FOSS systems, this also affects deeply their ecosystem. As Geiger, Howard and Irani show, the activities and experiences of maintenance work change and are seriously challenged [18], as FOSS projects are embedded within these broader ecosystems.

Recommendations

Building on our analysis and insights from informants, we identified a set of practices that could support mobile software maintenance:

- updates should not lead to breaking changes and should provide retro-compatibility for long periods of time (e.g. min 10 years);
- if breaking changes were to happen, they should be localized, and not system wide;

Because maintenance practices are enacted only when they align with the values and objectives of the stakeholders involved, it is also necessary to enforce them through regulation. The following recommendations range from the easiest measures to larger-scale transformations that would more radically improve longevity:

- require phone vendors to publish update and upgrade plans for every device, and ensure that they are followed;
- require phone vendors, SoC manufacturers and software companies to liberate all code, schematics and documentation related to hardware behaviour upon expiration of the warranty and support period;
- include mainlining and upstreaming conditionality into FOSS licences;
- require all components of a mobile system (firmware, OS) to be maintained for at least 10 years;
- all OS components should be required to follow (or define) open public standards, to avoid proprietary software lock-ins;
- consider mobile OS as public infrastructure, governed as digital commons.

6 Conclusion

In this paper we studied maintenance practices around the Android operating system: a complex ecosystem that involves open source actors, as well as large and diverse industrial actors. Android OS is the result of their cumulative work, a complex software with billions of lines of code, which can run on highly diverse hardware and which is deployed on

billions of devices. Despite this relative success, Android OS versions are short lived, Android smartphones are rarely updated, and compared to other devices, their lifespan is short.

Building on interviews with developers in the Android and other Linux-derived ecosystems, we show how Android OS differs from a traditional desktop OS: an Android OS build is unique to each device, which dramatically complicates the maintenance process. By mapping the Android ecosystem, the interaction between its actors, and the development flow of Android builds, we show how the actors of this ecosystem have diverging and often conflicting update and software maintenance strategies.

SoC developers and phone vendors have little incentive to maintain the code of the Android kernels that they build for each device, let alone contribute back (upstream) to the Linux kernel on which they are based. This would imply following community conventions and practices that allow for code sharing, documentation, easy updates and better long term maintenance. This lack of contribution also shapes the way these actors code. They focus on their internal specific needs, producing code that increasingly differs from the open-source code base, is little or not documented and contains several anti-patterns. This means that even if available, later opened or reverse-engineered, the code is particularly challenging to understand, reuse or maintain, denoting a lack of care and consideration for the community upon which it builds.

Key player in the ecosystem, we found that Google exercises its power in a selective manner, focusing on creating channels for selective updates, and enforcing its own proprietary services and applications while being reluctant to enforce upstreaming from SoC manufacturers and phone vendors. Long-term maintenance responsibility and work is transferred to free and open-source actors such as the Linux kernel community, alternative mobile OS systems such as LineageOS, Mobian, PostmarketOS, microG, UnifiedPush to cite a few, or to some phone vendors which emphasize longevity such as Fairphone. Because of lack of documentation, code accessibility and lock-ins, these actors put significant effort and develop circumvention strategies in order to maintain some Android builds on specific devices for seven, sometimes up to ten years.

Last but not least, by discussing these maintenance issues in the Android ecosystem, our work provides a better socio-technical understanding of software obsolescence and remediation strategies and offers recommendations for more sustainable software systems.

References

[1] Marwen Abbes, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2011. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In 2011 15th European Conference on Software Maintenance and Reengineering (2011-03). 181–190. doi:10.1109/CSMR.2011.24

[2] Bartels, Ermel, Sandborn, and Pecht. 2012. Software Obsolescence. In Strategies to the Prediction, Mitigation and Management of Product Obsolescence. John Wiley & Sons, Ltd, 143–155. doi:10.1002/9781118275474.ch6

[3] Ruven Brooks. 1983. Towards a Theory of the Comprehension of Computer Programs. 18, 6 (1983), 543–554. doi:10.1016/S0020-7373(83)80031-5

[4] Gerardo Canfora and Aniello Cimitile. 2001. Software Maintenance. In Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Company, 91–120. doi:10.1142/9789812389718_0005

[5] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd. 2002. Surviving the SOC Revolution. Kluwer Academic Publishers. doi:10.1007/b116290

[6] Marisa Leavitt Cohn. 2016. Convivial Decay: Entangled Lifetimes in a Geriatric Infrastructure. In Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (New York, NY, USA, 2016-02-27) (CSCW '16). Association for Computing Machinery, 1511–1523. doi:10.1145/2818048.2820077

[7] Marcel Den Hollander. 2018. Design for Managing Obsolescence: A Design Methodology for Preserving Product Integrity in a Circular Economy. doi:10.4233/uuid:3f2b2c52-7774-4384-a2fd-7201688237af

[8] Marcel Den Hollander, C.A. Bakker, and Erik Hultink. 2017. Product Design in a Circular Economy: Development of a Typology of Key Concepts and Terms: Key Concepts and Terms for Circular Product Design. In Journal of Industrial Ecology (2017-05-15), Vol. 21. doi:10.1111/jiec.12610

[9] Jérôme Denis, Alessandro Mongili, and David Pontille. 2015. Maintenance & Repair in Science and Technology Studies. In Tecnoscienza – Italian Journal of Science & Technology Studies (2015), Vol. 6. 5–15. Issue 2. doi:10.6092/issn.2038-3460/17251

[10] Jérôme Denis and David Pontille. 2015. Material Ordering and the Care of Things. In Science, Technology, & Human Values (2015-05-01), Vol. 40. SAGE Publications Inc, 338–367. doi:10.1177/0162243914553129

[11] Nadia Eghbal. 2020. Working in Public: The Making and Maintenance of Open Source Software. Stripe Press.

[12] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. In IEEE Transactions on Software Engineering (2001-01) (IEEE Transactions on Software Engineering, Vol. 27). 1–12. doi:10.1109/32.895984

[13] Hamid R. Ekbia and Bonnie A. Nardi. 2017. Heteromation, and Other Stories of Computing and Capitalism. The MIT Press. doi:10.7551/mitpress/10767.001.0001

[14] Federico Etro and Cristina Caffarra. 2017. On the Economics of the Android Case. In European Competition Journal (2017-09-02), Vol. 13. Routledge, 282–313. doi:10.1080/17441056.2017.1386957

[15] Daniel Feitosa, Apostolos Ampatzoglou, Antonios Gkortzis, Stamatia Bibi, and Alexander Chatzigeorgiou. 2020. CODE Reuse in Practice: Benefiting or Harming Technical Debt. In Journal of Systems and Software (2020-09-01), Vol. 167. 110618. doi:10.1016/j.jss.2020.110618

[16] Kiri Feldman and Peter Sandborn. 2009. Integrating Technology Obsolescence Considerations Into Product Design Planning. American Society of Mechanical Engineers Digital Collection, 981–988. doi:10.1115/DETC2007-35881

[17] Ben Fiedler, Daniel Schwyn, Constantin Gierczak-Galle, David Cock, and Timothy Roscoe. 2023. Putting out the Hardware Dumpster Fire. In Proceedings of the 19th Workshop on Hot Topics in Operating Systems (New York, NY, USA, 2023-06-22) (HOTOS '23). Association for Computing Machinery, 46–52. doi:10.1145/3593856.3595903

[18] R. Stuart Geiger, Dorothy Howard, and Lilly Irani. 2021. The Labor of Maintaining and Scaling Free and Open-Source Software Projects. In Proc. ACM Hum.-Comput. Interact. (2021-04-22), Vol. 5. 175:1–175:28. Issue CSCW1. doi:10.1145/3449249

[19] Stephen Graham and Nigel Thrift. 2007. Out of Order: Understanding Repair and Maintenance. In Theory, Culture & Society (2007-05-01), Vol. 24. SAGE Publications Ltd, 1–25. doi:10.1177/0263276407075954

[20] Samuel Greengard. 2025. Fighting for the Right to Repair. In Communications of The Acm (2025-09-05).

[21] Jeanne Guien. 2021. Le consumérisme à travers ses objets. Editions Divergences.

[22] Ibrahim Haddad and Cedric Bail. 2020. Technical Debt and Open Source Development. The Linux Foundation.

[23] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding Android Fragmentation with Topic Analysis of Vendor-Specific Bugs. In 2012 19th Working Conference on Reverse Engineering (2012-10). 83–92. doi:10.1109/WCRE.2012.18

[24] Steven J. Jackson and Laewoo Kang. 2014. Breakdown, Obsolescence and Reuse: HCI and the Art of Repair. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (New York, NY, USA, 2014-04-26) (CHI '14). Association for Computing Machinery, 449–458. doi:10.1145/2556288.2557332

[25] Ahmed Amine Jerraya, Sungjoo Yoo, Norbert Wehn, and Diederik Verkest. 2013. Embedded Software for SoC. Springer Publishing Company, Incorporated.

[26] Laura Kocksch and Torben Elgaard Jensen. 2024. The Mundane Art of Cybersecurity: Living with Insecure IT in Danish Small- and Medium-Sized Enterprises. In Proc. ACM Hum.-Comput. Interact. (2024-11-08), Vol. 8. 354:1–354:17. Issue CSCW2. doi:10.1145/3686893

[27] Matthias Korn and Susann Wagenknecht. 2017. Friction in Arenas of Repair: Hacking, Security Research, and Mobile Phone Infrastructure. In Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (New York, NY, USA, 2017-02-25) (CSCW '17). Association for Computing Machinery, 2475–2488. doi:10.1145/2998181.2998308

[28] M.M. Lehman. 1980. Programs, Life Cycles, and Laws of Software Evolution. In Proceedings of the IEEE (1980-09), Vol. 68. 1060–1076. doi:10.1109/PROC.1980.11805

[29] Ernst Leierzopf, René Mayrhofer, Michael Roland, Wolfgang Studier, Lawrence Dean, Martin Seiffert, Florentin Putz, Lucas Becker, and Daniel R. Thomas. 2024. A Data-Driven Evaluation of the Current Security State of Android Devices. In 2024 IEEE Conference on Communications and Network Security (CNS) (2024-09). 1–9. doi:10.1109/CNS62487.2024.10735682

[30] Xuetao Li, Yuxia Zhang, Cailean Osborne, Minghui Zhou, Zhi Jin, and Hui Liu. 2025. Systematic Literature Review of Commercial Participation in Open Source Software. In ACM Trans. Softw. Eng. Methodol. (2025-01-20), Vol. 34. 33:1–33:31. doi:10.1145/3690632

[31] Sherlock A. Licorish, Amjed Tahir, Michael Franklin Bosu, and Stephen G. MacDonell. 2015. On Satisfying the Android OS Community: User Feedback Still Central to Developers' Portfolios. In 2015 24th Australasian Software Engineering Conference (2015). IEEE, 78–87. doi:10.1109/ASWEC.2015.19

[32] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (New York, NY, USA, 2013-08-18) (ESEC/FSE 2013). Association for Computing Machinery, 477–487. doi:10.1145/2491411.2491428

[33] Javier Lloveras, Mario Pansera, and Adrian Smith. 2025. On 'the Politics of Repair Beyond Repair': Radical Democracy and the Right to Repair Movement. In Journal of Business Ethics (2025-01-01), Vol. 196. 325–344. doi:10.1007/s10551-024-05705-z

[34] Lise Magnier and Ruth Mugge. 2022. Replaced Too Soon? An Exploration of Western European Consumers' Replacement of Electronic Products. In Resources, Conservation and Recycling (2022-10-01), Vol. 185. 106448. doi:10.1016/j.resconrec.2022.106448

[35] Irene Maldini, Ingun Grimstad Klepp, and Kirsi Laitala. 2025. The Environmental Impact of Product Lifetime Extension: A Literature Review and Research Agenda. In Sustainable Production and Consumption (2025-06-01), Vol. 56. 561–578. doi:10.1016/j.spc.2025.04.020

[36] Frédéric Marty. 2022. Pré-Installations, Biais de Statu Quo et Consolidation de La Dominance : Les Enseignements de l'arrêt Du Tribunal de l'U.E. Dans l'affaire Google Android. In CIRANO (2022-11), Vol. 2022s-29, CIRANO. doi:10.54932/YOZL1587

[37] L. Merola. 2006. The COTS Software Obsolescence Threat. In Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'05) (2006-02). 7 pp.–. doi:10.1109/ICCBSS.2006.29

[38] Annemarie Mol. 2008. The Logic of Care: Health and the Problem of Patient Choice (1 ed.). Routledge. doi:10.4324/9780203927076

[39] Léa Mosesso, Nolwenn Maudet, Edlira Nano, Thomas Thibault, and Aurélien Tabard. 2023. Obsolescence Paths: Living with Aging Devices. In ICT4S 2023 - International Conference on Information and Communications Technology for Sustainability (Rennes, France, 2023-06). doi:10.1109/ICT4S58814.2023.00011

[40] Competition and Markets Authority of the UK Government. 2022. Google's Agreements with Device Manufacturers and App Developers. https://www.gov.uk/cma-cases/mobile-ecosystems-market-study

[41] Competition and Markets Authority of the UK Government. 2022. Mobile Ecosystems Market Study. https://www.gov.uk/cma-cases/mobile-ecosystems-market-study

[42] Michael Pecht, Rajeev Solomon, Peter Sandborn, Chris Wilkinson, and Diganta Das. 2004. Obsolescence Prediction and Management. In Parts Selection and Management. John Wiley & Sons, Ltd, 231–263. doi:10.1002/0471723886.ch16

[43] Cristiano Politowski, Foutse Khomh, Simone Romano, Giuseppe Scanniello, Fabio Petrillo, Yann-Gaël Guéhéneuc, and Abdou Maiga. 2020. A Large Scale Empirical Study of the Impact of Spaghetti Code and Blob Anti-Patterns on Program Comprehension. In Information and Software Technology (2020-06-01), Vol. 122. 106278. doi:10.1016/j.infsof.2020.106278

[44] C. Paul Rogers. 2021. Competition Law and the E.U. and U.S. Approaches to Dominant Markets: Will the Gap Narrow? Social Science Research Network. https://ssrn.com/abstract=4178276

[45] Daniela K. Rosner and Morgan Ames. 2014. Designing for Repair? Infrastructures and Materialities of Breakdown. In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing (New York, NY, USA, 2014-02-15) (CSCW '14). Association for Computing Machinery, 319–331. doi:10.1145/2531602.2531692

[46] Peter Sandborn. 2013. Design for Obsolescence Risk Management. In Procedia CIRP (2013-01-01) (2nd International Through-life Engineering Services Conference, Vol. 11). 15–22. doi:10.1016/j.procir.2013.07.073

[47] Peter Sandborn. 2008. Software Obsolescence: Complicating the Part and Technology Obsolescence Management Problem. In Components and Packaging Technologies, IEEE Transactions On (2008-01-01), Vol. 30. 886–888. doi:10.1109/TCAPT.2007.910918

[48] Peter A. Sandborn, Frank Mauro, and Ron Knox. 2007. A Data Mining Based Approach to Electronic Part Obsolescence Forecasting. In IEEE Transactions on Components and Packaging Technologies (2007-09), Vol. 30. 397–401. doi:10.1109/TCAPT.2007.900058

[49] Sarah Suzaña. 2021. The European Commission vs. Google: Analysis of the Cases AT. 40099 (Google Android) and AT. 40411 (Google AdSense) for Abuse of Dominant Position. In Università Della Calabria (2021).

[50] Sahra Svensson, Jessika Luth Richter, Eléonore Maitre-Ekern, Taina Pihlajarinne, Aline Maigret, and Carl Dalhammar. 2018. The Emerging 'Right to Repair' Legislation in the EU and the U.S.. In Going Green CARE INNOVATION 2018 (2018).

[51] Andrew S. Tanenbaum and Herbert Bos. 2015. Modern Operating Systems (global ed.). Pearson Education.

[52] The Linux Kernel Media team. 2022. Report of the Kernel CAM Topic. https://linuxtv.org/news.php?entry=2022-11-14-1.hverkuil

[53] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. 2015. Security Metrics for the Android Ecosystem. In Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (New York, NY, USA, 2015-10-12) (SPSM '15). Association for Computing Machinery, 87–98. doi:10.1145/2808117.2808118

[54] Right to Repair coalition and iFixit. 2025. The Current State of Right to Repair in the EU: A Snapshot. https://repair.eu/news/the-current-state-of-right-to-repair-in-the-eu-a-snapshot

[55] Nina Troeger, Harald Wieser, and Renate Hübner. 2017. Smartphones Are Replaced More Frequantly than T-Shirts: Patterns of Consumer Use and Reasons for Replacing Durable Goods. Arbeiterkammer of Austria.

[56] Lachlan D Urquhart, Susan Lechelt, Christopher Boniface, Haili Wu, Anna Marie Rezk, Nidhi Dubey, Melissa Terras, and Ewa Luger. 2024. The Right to Repair (R2R) Cards: Aligning Law and Design For A More Sustainable Consumer Internet of Things.. In Proceedings of the 13th Nordic Conference on Human-Computer Interaction (New York, NY, USA, 2024-10-13) (NordiCHI '24). Association for Computing Machinery, 1–20. doi:10.1145/3679318.3685341

[57] A. Von Mayrhauser and A.M. Vans. 1995. Program Comprehension during Software Maintenance and Evolution. In Computer (1995-08), Vol. 28. 44–55. doi:10.1109/2.402076

[58] Kênia Pereira Batista Webster, Káthia Marçal De Oliveira, and Nicolas Anquetil. 2005. A Risk Taxonomy Proposal for Software Maintenance. In 21st IEEE International Conference on Software Maintenance (ICSM'05) (2005). IEEE, 453–461. doi:10.1109/ICSM.2005.14

[59] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (Singapore Singapore, 2016-08-25). ACM, 226–237. doi:10.1145/2970276.2970312

[60] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and Detecting Fragmentation-Induced Compatibility Issues for Android Apps. In IEEE Transactions on Software Engineering (2018), Vol. 46. IEEE, 1176–1199. doi:10.1109/TSE.2018.2876439

[61] Dunia P. Zongwe. 2023. The Economics of Repair: Fixing Planned Obsolescence by Activating the Right to Repair in India. In International Journal on Consumer Law and Practice (2023-01-01), Vol. 11. https://repository.nls.ac.in/ijclp/vol11/iss1/6