

Keep Me Updated: An Empirical Study of Proprietary Vendor Blobs in Android Firmware

Elliott Wen

The University of Auckland, New Zealand
elliott.wen@auckland.ac.nz

Jiaxing Shen

Lingnan University, Hong Kong
jiaxingshen@LN.edu.hk

Burkhard Wuensche

The University of Auckland, New Zealand
burkhard@cs.auckland.ac.nz

Abstract—Despite extensive security research on various Android components, such as kernel or runtime, little attention has been paid to the proprietary vendor blobs within Android firmware. In this paper, we conduct a large-scale empirical study to understand the update patterns and assess the security implications of vendor blobs. We specifically focus on GPU blobs because they are loaded into every process for displaying graphics user interfaces and can affect the entire system’s security. We examine over 13,000 Android firmware releases between January 2018 and April 2024. Our results reveal that device manufacturers often neglect vendor blob updates. About 82% of firmware releases contain outdated GPU blobs (up to 1,281 days). A significant number of blobs also rely on obsolete LLVM core libraries released more than 15 years ago. To analyze their security implications, we develop a performant fuzzer that requires no physical access to mobile devices. We discover 289 security and behavioral bugs within the blobs. We also present a case study demonstrating how these vulnerabilities can be exploited via WebGL. This work underscores the critical security concerns associated with vulnerable vendor blobs and emphasizes the urgent need for timely updates from device manufacturers.

I. INTRODUCTION

Nowadays, Android plays a fundamental role in our digital lives. It holds a 85% of the mobile phone market share and is also extensively used in various smart devices, such as TVs and watches. An Android device tends to have a highly complex software composition. Its base is the Linux Kernel, while the system runtime comes from Google’s Android Open Source Project (AOSP). Additionally, device manufacturers often pre-install specific system applications and incorporate proprietary binary drivers. This diverse composition results in a fragmented Android ecosystem and poses huge security challenges.

Recently, numerous research efforts have been made to investigate Android component security. For instance, Zhang et al. [1] explored the Android Linux kernel code review process and identified bottlenecks in patch application. Tung et al. [2] investigated the device manufacturers’ update practices for their AOSP components. Elsabagh et al. [3] applied static analysis to detect privilege escalation vulnerabilities in pre-installed apps. Despite these advancements, a gap remains in the analysis of security situations regarding proprietary vendor blobs within Android firmware. These blobs are distributed by device vendors and located in a special firmware partition (i.e., /vendor). They provide essential support for the device’s hardware components, such as GPUs, cameras, and fingerprint

readers. Many of them are loaded into apps or system services with high privileges. If these blobs are vulnerable, they can significantly impact the security of the entire system. However, due to their closed-source nature and lack of documentation, they are subject to limited scrutiny.

In this study, we conduct a large-scale empirical evaluation of vendor blobs within Android firmware. Our research aims to 1) understand the update pattern of vendor blobs and 2) identify their vulnerabilities and assess their security implications for the Android ecosystem. We place a specific focus on GPU vendor blobs based on two key reasons. Firstly, GPU blobs are more critical to system security than other blobs because every app loads them into memory for rendering graphical user interfaces (UI). Secondly, they are easily exploitable; many commonly used apps (e.g., Google Chrome and Firefox) or built-in UI components (e.g., WebKit) can accept arbitrary GPU task inputs and forward them to the vulnerable GPU blobs.

To achieve our goals, we construct an automatic analysis pipeline named *GPUBlob-Inspector* as shown in Fig 1. The pipeline begins with a firmware crawler that gathers Android firmware images from various device manufacturers. The firmware is then input into an image unpacker to extract the proprietary vendor blobs. Afterward, the pipeline proceeds to identify the blob’s version number via Executable and Linkable Format (ELF) fingerprints. Subsequently, our pipeline employs a specially designed fuzzer on the GPU vendor blobs to identify potential security vulnerabilities. Our fuzzer utilizes metamorphic testing; it generates and feeds a series of semantically equivalent GPU programs to the GPU blobs to observe any irregular behaviors or inconsistent program outputs. Unlike existing GPU fuzzers that necessitate physical access to mobile devices, our system operates in an offline manner. We achieve this by utilizing the fact that those GPU blobs internally generate Low Level Virtual Machine (LLVM) Intermediate Representation (IR) for the input GPU program. We can capture the LLVM IR and use it to directly generate an executable for performant CPU-based fuzzing. In the final stage, we conduct a manual analysis to validate the vulnerabilities on physical devices.

For our study, we curate a large-scale firmware dataset, comprising 13,901 Android images from 74 distinct phone vendors. The data span a period ranging from Jan 2017 to April 2024. From this dataset, we unveil the following key

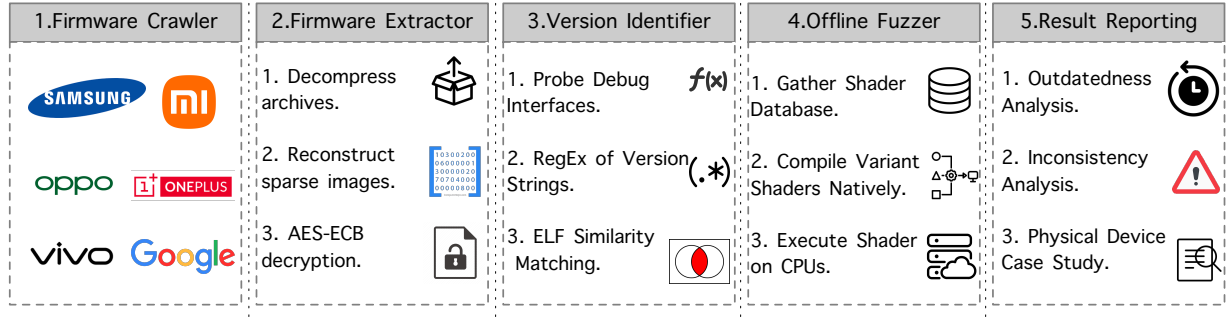


Fig. 1: GPU-Inspector Pipeline.

research findings.

- 1) Most manufacturers neglect timely updates to the GPU vendor blobs; approximately 82% of firmware images in our dataset contain outdated GPU blobs. On average, a GPU blob remains outdated for 273 days, with the longest period stretching up to 1,128 days. In addition, certain manufacturers tend to update these blobs far less frequently than others.
- 2) Most GPU vendor blobs are built upon outdated LLVM core libraries. A significant portion of them rely on *LLVM 2.8*, a version that is released over 15 years ago. Even the most recent LLVM dependency found among these blobs dates back to 5 years ago.
- 3) Our offline fuzzer identifies 289 security vulnerabilities and behavioral anomalies within the most recent Qualcomm GPU blobs. These issues stem from memory access violations, endless iterations of optimization passes, and incorrect program semantics. We present a case study to exploit them via WebGL, which may lead to denial of service or arbitrary code execution on a target device.

To contribute to future research, we also make the source code of our analysis pipeline and the dataset publicly available.

II. BACKGROUND: PROPRIETARY VENDOR BLOBS

Android is an open ecosystem, yet it still allows manufacturers to include closed-source blobs in their distributions. This practice is typically used for hardware-specific code, where manufacturers may opt to keep their implementation details confidential. These vendor blobs must adhere to a standardized set of interface definitions known as the Hardware Abstraction Layer (HAL), as illustrated in Figure 2. For instance, a fingerprint reader blob needs to implement the `biometrics.fingerprint@2.1` interfaces, while a GPU blob needs to implement the core OpenGL ES or Vulkan graphics interfaces. There are two types of HAL interfaces: *Binderized* and *Same-process*. A binderized blob (e.g., fingerprint) is generally loaded into a high-privilege system service, while a same-process blob (e.g., GPU) opens in the same process in which it is used.

Vendor blobs generally operate within the user space and have corresponding Linux modules that run in the kernel space. These blobs and kernel modules communicate through standard input-output subsystem protocols, such as *IOCTL* and

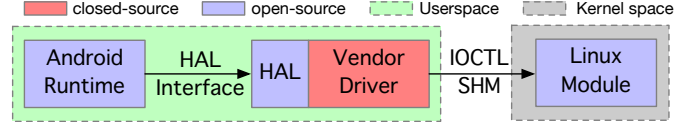


Fig. 2: Proprietary Vendor Blobs in Android.

shared memory. To comply with the GNU General Public License (GPL) of Linux, the kernel modules are open-source. They generally just perform basic low-level tasks such as register manipulation and memory management, while the core logic for major hardware operations remains implemented and concealed within the vendor blobs.

In the context of GPU blobs, there are two main hardware vendors: Qualcomm Adreno and ARM Mali. They both follow the aforementioned HAL architecture. For example, Qualcomm employs a Linux kernel module named *KGSL* to handle basic low-level hardware operations such as power management and register initialization. Its user-space library *QGL* handles the core GPU tasks, such as providing standard graphics APIs (e.g., OpenGL ES and Vulkan) and compiling GPU programs into hardware instructions. The user-space library is closed-source for device end-users, while its source code is provided to device manufacturers such as Google and Xiaomi under a Non-Disclosure Agreement (NDA). Consequently, the responsibility for updating the driver rests entirely with the manufacturers.

III. GPU-INSPECTOR PIPELINE

In this section, we provide implementation details for our *GPUBlob-Inspector* pipeline.

A. Firmware Curation and Vendor Blob Unpacking

We implement a web crawler to curate Android firmware images from various vendors. Some vendors, such as Google and Xiaomi, make all historical firmware versions available on their official websites. However, others like Samsung and OnePlus do not provide publicly accessible URLs. In such cases, we source their firmware images from third-party subscription services like SamMobile¹ and Daxiaamu². Alongside firmware binaries, we also collect metadata information, including firmware version, release date, and supported regions.

¹<https://www.sammobile.com/>

²<https://yun.daxiaamu.com>

To enhance dataset diversity, we also incorporate data from Android Dump, a public repository hosting firmware from lesser-known phone vendors. Our dataset comprises 13,901 firmware images across 24 phone vendors, consuming a total of 38 TBs of disk space. They are released between January 2018 and April 2024. Among them, the top five phone vendors (Samsung, Xiaomi, Oppo, OnePlus, and Google) collectively contribute to 93% of the images.

Generally, a firmware image consists of multiple compressed partitions such as *boot*, *recovery*, *system*, and *vendor*. Our goal is to decompress the vendor partition, where the vendor blobs generally reside. To accomplish this, we adapt an Android firmware analysis tool³ as recommended by previous research. Upon decompression, we proceed to retrieve the GPU vendor blobs. According to the Android framework specifications, they must be situated within the directory `/vendor/lib64/egl` and may be distributed as a singular binary (*libGLES.so*) or as three separate binaries (*libEGL.so*, *libGLESv1_CM.so*, and *libGLESv2.so*). It is important to note that these GPU libraries may exhibit dependencies on other libraries within the vendor partition, which in turn may have further dependencies. To ensure a thorough extraction of all dependent libraries, we utilize a tool named *readelf*⁴ to parse a library’s *DT_NEEDED* header section and identify its dependencies. We initiate this process with the GPU library and we recursively apply *readelf* to each dependent library we encounter.

B. Driver Version Identification

The next task is to identify the version of these GPU vendor blobs. This process is not straightforward due to the absence of version metadata within the binary. Consequently, we devise our own versioning scheme, which comprises the following three components:

- 1) Build ID: One rudimentary yet effective versioning approach is to use the build ID of each blob file. A build ID is essentially a hexadecimal hash string that is dependent on compilation inputs, particularly, the source files and compiler optimization settings. If these elements remain unchanged, the build ID will also remain consistent. This characteristic makes the build ID superior to a general-purpose file hash, which can vary due to non-essential metadata such as timestamps or debugging strings. The build ID can be directly extracted from a specific metadata section *.note.gnu.build-id* of a library file. It should be noted that build IDs alone are not adequate for establishing the relative order of the blobs (i.e., which blob has a more recent version).
- 2) Internal Blob Version: To overcome the Build ID’s limitation, we introduce another component to our versioning scheme. We leverage the fact that certain GPU blobs generate debug logs during their initialization phase. These logs often include an internal version

string with numerical components that can be compared. Specifically, Qualcomm GPU blobs use the format `EV{major}.{minor}.{patch}`, while ARM GPU blobs employ a version string in the format of `r{major}p{patch}`. We can apply these regular expressions to retrieve the internal version numbers. However, it is worth noting that these debug strings may not always be present if the GPU blob developers enable aggressive optimizations during the build process.

- 3) LLVM Compiler Version: In the event of missing internal version numbers, we resort to another observation: both Qualcomm and ARM GPU blobs incorporate an LLVM compiler library. The rationale behind GPU blobs embedding a compiler lies in their need to process shader programs. Specifically, shader programs are executed by the GPU for graphics rendering. They are typically written in a C-like high-level language such as GLSL or HLSL. As such, they must be compiled into corresponding GPU hardware instructions prior to execution. The LLVM library itself contains a version string following a format of `{major}.{minor}.{patch}.{commithash}`. We employ a regular expression to search for this version string if the LLVM binary is not stripped. In case of a stripped library, we can still use fingerprinting strategies to estimate the LLVM library version. One simple fingerprint is the textual strings found in the prevalent logging statements within the LLVM library. As LLVM evolves, for instance, with the introduction of new optimization passes, new strings are frequently added. This characteristic makes it a robust method for identifying the LLVM version. In addition, we can incorporate a more complicated fingerprint known as Binshape [4]. This approach extracts a combination of features from the body of each function, such as the initial byte sequences (i.e., the prologue), call graphs, and statistics of machine instructions within a function. This approach allows us to identify not only the version but also the LLVM function names in a stripped binary.

C. GPU Blob Fuzzing

The upcoming task is to identify potential bugs within the GPU vendor blobs. A commonly-used method is fuzzing, where we supply a system with randomly generated inputs and observe any resulting exceptions. For GPU, a particular fuzzing technique known as metamorphic testing is often favored [5]. This process commences with a reference shader program, typically procured from readily available mobile games. We then apply various transformations that preserve the semantics of the source code (e.g., converting a ‘for’ loop to a ‘while’ loop). This generates a collection of shader variants with heavily modified source code, yet maintaining the same output effect. If a variant shader produces an output significantly different from the reference shader, it indicates a potential bug in the GPU stack. This method is employed by the state-of-the-art GPU fuzzer *GraphicsFuzz* [6]. However,

³<https://github.com/srlabs/extractor>

⁴<https://man7.org/linux/man-pages/man1/readelf.1.html>

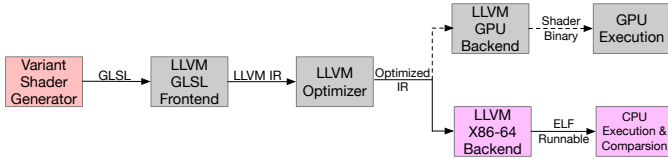


Fig. 3: LLVM Compiler Pipeline.

for our large-scale study, we find GraphicsFuzz unsuitable since it requires physical access to mobile devices. Furthermore, GraphicsFuzz provides a fuzzing speed of less than ten shaders per second. Considering the substantial volume of GPU blobs in our dataset, we require a fuzzer with a higher speed to complete the analysis within a practical timeframe.

In this paper, we develop a performant offline fuzzer for the proprietary GPU blobs. We exploit the fact that both Qualcomm and ARM GPU blobs are constructed on the LLVM compiler infrastructure. LLVM encompasses a three-phase compilation process, as depicted in Figure 3. The frontend is responsible for converting the shader source code into an intermediate representation (IR). This IR is a platform-agnostic representation that preserves the semantic meaning of the source code. Subsequently, the optimizer ingests the frontend IR and applies a series of transformations to augment the code’s efficiency. Exemplary transformations encompass constant folding, dead code elimination, and loop optimizations. Finally, the backend is responsible for translating the optimized LLVM IR into machine code for the target GPU architecture.

Our fuzzer instruments the LLVM library to capture IR after the optimization passes. Instead of sending the IR to a GPU backend, we reroute it to an X86 backend to create an executable for native CPU execution. This approach lets us efficiently compare the output of the reference shader and variant shader without access to mobile phone GPUs. Below, we outline the implementation details of our fuzzer.

Reference Shader Database: Our fuzzer requires a reference shader database to generate shader variants. In this study, we curate a shader database from real-world games. We commence by crawling and analyzing the popular Android app stores such as *Google Play*, *APKPure*, and *APKMirror*. As of Feb 2024, we compile a list of 145,732 gaming apps from these platforms. We also gather relevant metadata, such as game descriptions, categories, and download counts. Through our analysis on the metadata, we identified 10 popular game genres including 1) Action, 2) Adventure, 3) Board, 4) Causal, 5) Puzzle, 6) Racing, 7) Role-Playing, 8) Simulation, 9) Sports, and 10) Strategy. For each genre, we employ a weighted random sampling technique to select 20 games. The weight assigned to each candidate app is proportional to its download count. It ensures our selection leans towards popular games in the app stores. Afterward, we use Patrace⁵ to extract the shader extraction from the selected games. Specifically,

Patrace is designed to capture GLES calls at runtime from a gaming app and record them into a trace file for performance analysis. To extract the shader sources, we can filter out the `glShaderSource` command from the trace files. In our study, we assigned a researcher with extensive experience in mobile gaming to evaluate each selected game for a minimum of 10 minutes. During this process, we activate the Patrace’s capturing functionality, which results in approximately 2000 minutes of trace recording. From these trace files, we recover 18,923 shader programs.

Variant Shader Generation: To generate the shader variants, we adapt a series of semantic-preserving transformations from Graphicfuzz on our shader database. These transformations can be categorized as follows:

- 1) **Statement Mutation:** This transformation process takes a simple statement denoted as *origin* and converts it into a more complex one using constructs from GLSL. One exemplary construct is the function `mix(origin, unused, c)`, which yields a linear combination of $origin * c + unused * (1 - c)$. Consequently, if we set $c = 1$, the output will remain as *origin*.
- 2) **Control Flow Mutation:** In this transformation, we can interchange certain control flow constructs, for example, switching from ‘if’ to ‘switch’, or from ‘for’ to ‘while’. We can also insert additional control flow elements, such as wrapping a code segment in a loop that executes only once. Alternatively, we can make a control flow construct more complex, for instance, unrolling a loop or splitting a loop.
- 3) **Code Donation:** In this transform, we extract a group of statements from a randomly selected reference shader. These statements are then injected into the target shader. To avoid affecting the output of the target shader, we randomly rename each variable in the donated code and store the computational results to some unused built-in output variables.

These transformations can be chained in a random order and in a recursive manner to generate complex variant shaders.

LLVM IR Capturing: After generating shader variants, we feed them to the GPU blobs to perform shader compilation and capture the LLVM IR. One approach is to leverage the standard OpenGL graphics interfaces exposed by the GPU blobs. Specifically, we can first leverage the `eglCreateContext` API to initialize a global EGL context, which serves as an interface between OpenGL APIs and underlying hardware. Afterward, a shader object can then be created using the `glCreateShader` function. Subsequently, the shader source code is attached to the object using `glShaderSource`. Finally, the shader can be compiled using `glCompileShader`. To capture the LLVM IR during the compilation, we can leverage a LLVM command line option, `-print-after-all`. It instructs the compiler to print out the IR after each optimization pass for debugging purposes. We can inject this option by invoking an LLVM function named `cl::ParseCommandLineOptions`. The

⁵<https://github.com/ARM-software/patrice>

```

; Input/Output Variables
@a_color = external global <4 x float>, align 16
@v_fragmentColor = external global <4 x float>, align 16
define void @llvm_main() {
; Shader Input
%reg_a_color_0 = call float @llvm.qgpu.fget.reg.f32.p0v4f32
(<4 x float>* @a_color, i32 0, i32 1)
; Shader Math Operations
%color_rsqr = call float @llvm.qgpu.rsqr(float %reg_a_color_0)
; Shader Sample Functions
%texture_val = call <4 x float> @llvm.qgpu.fsampler.v4f32.
v2i16.v2f32.i32 (i32 0, <2 x i16> zeroinitializer,
<2 x float> %sample_coords, i32 undef,
<4 x i32> <i32 128, i32 0, i32 0, i32 0>,
i16 0, i16 0)
; Shader Output
call void @llvm.qgpu.global.fset.reg.p0f32.v4f32
(float* @getelementptr inbounds (<4 x float>*
@v_fragmentColor, i32 0, i32 0), <4 x float> %4,i32 0,i32 4)
}

```

Fig. 4: Example LLVM IR Code for an OpenGL Shader

memory address of this function can be identified using the string fingerprint method, as discussed in Section III-B.

To execute the aforementioned APIs without access to physical devices, we can set up a QEMU full-system emulator to run a vanilla Android system (i.e., AOSP). By default, the emulated Android system employs a software-based OpenGL implementation. We can upload the proprietary GPU blobs to the emulator and replace the default implementation by adjusting an environment variable named `debug.gles.layers`⁶. This action alone is not sufficient. When creating an EGL context, the proprietary binary needs to issue IOCTL system calls to the GPU kernel module for querying hardware information such as GPU model and VRAM size. Since this kernel module is absent in the emulator, an EGL exception will be raised. We circumvent this issue by implementing a shim Linux kernel module for the emulator. We stub out most IOCTL calls in the original GPU kernel drivers and retain only those necessary for EGL context creation.

For Qualcomm GPU blobs, we can adopt a more efficient approach by exploiting the fact that Qualcomm GPU drivers partition the compiler component into a separate library named `libllvm-glnext.so`. This library relies only on C runtime libraries (bionic C) and exposes accessible compilation interfaces. This allows us to directly invoke the compiler component via a user-space ARM CPU emulator (i.e., `aarch-qemu`), which is more performant than full system emulation. Specifically, we first utilize the dynamic linking API `dlopen()` to load the target library into the memory. Afterward, we retrieve the function pointers of compiler interfaces via the API `dlsym()`. A primary function is `QGLCCompileToIRShader`, which ingests a shader source code string buffer and returns the optimized LLVM IR. An illustrative snippet of LLVM IR is presented in Figure 4.

CPU Binary Generation and Execution The next step is to generate a CPU executable from the captured IR. Specifically, we first use the LLVM static compiler `llc` to translate the IR into x86 assembly language. The resulting assembly output is

then passed through the x86 assembler `llvm-mc` and the x86 linker `lld` to produce a native executable. Special attention is required during the linkage phase, as the captured IR often includes invocations to various intrinsic functions. These functions represent computational operations that can be translated into efficient GPU machine instructions. They are identifiable by their unique naming convention, which begins with `llvm`. Since these intrinsic functions are not available on the X86 platform, we provide our software implementation to prevent linkage errors as follows.

Shader Input/Output: In OpenGL, an input variable and output variable can be defined using the keywords `in` and `out`, respectively. Access to these variables is translated to intrinsic function invocations, `llvm.qgpu.fget` and `llvm.qgpu.fset` respectively. These functions accept a GPU memory address as input. In our backend, we remap the GPU addresses into heap memory regions, which are pre-allocated using the system call `mmap`. At the entry point of each shader program, we populate the input memory regions with randomly generated values using predetermined seeds. Upon the shader program’s exit, we generate hashes for the contents of the output memory regions. This allows us to compare the output of a variant shader with that of a reference shader.

Math Operations: Another common type of intrinsics encompasses mathematical computations. For example, the intrinsic `llvm.qgpu.rsqr` is employed to compute the reciprocal square root of a floating-point number. In a Qualcomm GPU, this intrinsic can be directly mapped into the machine instruction `rsqr`, which consumes only three GPU cycles. In our X86 backend, we need to remap it to corresponding math functions within the `libm.so` library. A technical challenge is that OpenGL allows developers to use half-precision floating point numbers in a shader. However, on the X86 platform, the `libm` library does not support half floating-point computations. One naive workaround is to emulate half-precision operations using software (e.g., Berkeley Softfloat). This inevitably incurs significant performance penalty. Instead, we implement an LLVM transformation pass that iterates over all IR instructions and examines each float operand. If an operand is of half-precision, we then promote it to single precision. Special attention is given to the `fpext` instruction, which extends a value from a smaller to a larger floating-point type. Since all floating-point operands have already been promoted to the highest precision, we convert `fpext` to `bitcast` to make it a no-op operation. This strategy is implemented at compilation time and does not incur any runtime performance overhead.

Shader Sampler: Another common type of intrinsic functions is sampler function, for instance, `llvm.qgpu.fsampler`. A sampler is used to fetch and process texels from texture resources. It controls various aspects of texture representation, such as filtering, wrapping, and mipmapping. Executing these sampler functions on x86 CPUs can be computationally intensive and inefficient. Instead, we simplify the sampler function as a hash function. Given the same sampling parameters and global random seed,

⁶<https://developer.android.com/ndk/guides/rootless-debug-gles>

it consistently generates the same random texture output. It behaves as if we are providing procedurally generated textures to the shaders.

We implement these intrinsic functions in C and compile them into object files using Clang. These object files are subsequently linked with the LLVM IR to produce an ELF executable. In this process, we need to address differences in the Application Binary Interface (ABI), which governs the data exchange protocol between object files originating from different source languages. Specifically, when LLVM attempts to pass a small float vector (e.g., $4 * \text{float}$) to C, the data is, by default, passed through an xmm register. However, a C function expects the float array to be passed on the stack. To resolve this issue, we explicitly define the incoming parameter in the C function using Clang’s `ext_vector_type` attribute. This enforces Clang to retrieve incoming float arrays from the xmm register.

IV. EVALUATION

In this section, we present our research findings for our two primary research questions: 1) What is the update pattern of GPU vendor blobs? and 2) How vulnerable are these GPU vendor blobs? Additionally, we conduct a case study to exploit these vulnerabilities to launch a denial-of-service or even arbitrary code execution attack on the target device.

A. Experiment Setup

We deploy our *GPUBlob-Inspector* pipeline on an Ubuntu 24.04 machine. The machine is equipped with two AMD EPYC 7742 64-Core Processor CPUs, 1024 GB of RAM, 128 TB SSD hard disks, and a 10 Gbps network connection. We provide a detailed breakdown of the time taken at each stage of the pipeline. The initial phase of firmware crawling (approximately 13,000 images) is completed within 27 hours. The subsequent task of firmware extraction is accomplished in 147 minutes. The average processing time for each firmware is approximately 38 seconds, with the maximum processing time recorded at 278 seconds. The version identification stage is completed within 22 minutes, with an average processing time of 12 seconds and a maximum of 19 seconds per GPU blob. In the fuzzing stage, we generate 200 variants for each reference shader. The generation process takes approximately 491 minutes. We then input the variant shaders into a GPU blob to generate LLVM IR, which consumes 189 minutes. The compilation of these IRs into x86-64 executable ELF files requires 127 minutes. Executing these files and comparing their outputs takes about 7 minutes. It is important to note that the duration of the fuzzing stage is contingent on the number of shader variants generated. Increasing the number of shader variants enhances the probability of bug discovery but also prolongs the process.

B. RQ1: Driver Update Frequency

Our pipeline successfully analyzed 13,901 firmware images. Among these, a significant proportion, 78%, contain Qualcomm GPU blobs. Only 21% of the images contain ARM GPU

TABLE I: GPU Composition Across Major Manufacturers

Qualcomm / ARM	Firmware	Distinct Driver	Device
Samsung	73% / 27%	60% / 40%	61% / 39%
Xiaomi	70% / 30%	67% / 33%	73% / 27%
Oppo	72% / 28%	64% / 36%	62% / 38%
Google	75% / 25%	56% / 44%	58% / 42%
OnePlus	82% / 18%	61% / 39%	78% / 22%
Total	78% / 22%	60% / 40%	69% / 31%

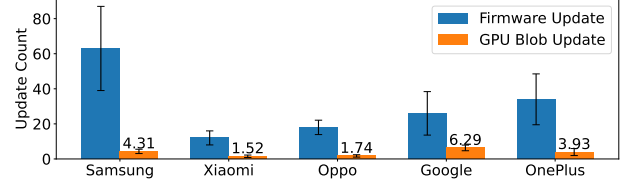


Fig. 5: Update Count for Firmware and GPU Blobs per Device

blobs. The remaining fraction is made up of Nvidia Tegra and MediaTek Dimensity. Due to their minimal market shares, we exclude these two from our results. We further de-duplicate all the GPU blobs in our dataset using their Build IDs. We identify 792 distinct versions. Specifically, 65% are provided by Qualcomm, while 35% are sourced from ARM. We also provide the GPU composition ratio across several top-tier mobile device manufacturers in Table I. The device count also indicates a significantly higher popularity of Qualcomm GPUs compared to ARM GPUs. We observe that Qualcomm GPUs are predominantly found in mid-range to high-end devices potentially due to their higher graphics rendering capabilities. In contrast, ARM GPUs are typically utilized in entry-level models. These findings are consistent with a previous report⁷.

Vendor Blob Update Count per Device: We quantify the number of firmware and GPU driver updates across different manufacturers. The findings are presented in Figure 5. For comparative purposes, the average update number per device is reported. A notable observation is that the number of firmware updates substantially surpasses that of GPU vendor blobs. It suggests that device manufacturers tend to prioritize firmware updates in a device’s life cycle. Another interesting observation is that certain manufacturers, such as Xiaomi and Oppo, exhibit an average GPU driver update count close to one. This indicates that the majority of their devices rarely receive GPU vendor blob updates throughout their lifecycle. In stark contrast, Google devices receive, on average, six GPU vendor blob updates. This discrepancy could be attributed to Google’s role as the first-party developer of the Android operating system. As such, Google likely has more resources for device maintenance and possesses greater influence over its software and hardware ecosystem.

Vendor Blob Update Delay: A blob update delay is defined as a situation where a firmware release contains a GPU blob version that is not as up-to-date as the version already

⁷<https://www.techcenturion.com/mobile-gpu-rankings>

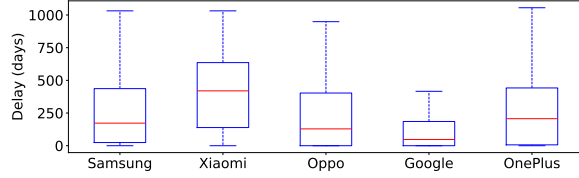


Fig. 6: Blob Update Delay Estimation

in use by another device with the same GPU hardware. This means the firmware release could have potentially utilized the newer blob for enhanced performance or security. The degree of the GPU blob update delay can be estimated as follows:

- 1) Given a target firmware image with a release date r_f and its associated GPU blob version v_o , we search our firmware database for the oldest firmware image containing the same blob and denote its release date as r_o . This process estimates the date when this version of the GPU blob was first introduced.
- 2) Following this, we filter our database for firmware that is equipped with the same GPU model and is released prior to the date r_f . Among these filtered results, we identify the most recent GPU driver version, v_l . In other words, the target firmware could have updated the blob version from v_o to v_l .
- 3) In the final step, we search our firmware database for the earliest firmware that incorporates the driver version v_l and denote its release date as r_l . We then can compute the delay as $D = r_l - r_o$, which estimates the degree of blob outdatedness.

From our dataset, we observed that 82% of firmware images contain outdated GPU blobs. The median value of the delay D is 273 days, with the highest value reaching 1,128 days. This observation suggests that most device manufacturers neglect timely updates on vendor blobs, potentially leaving devices in a more vulnerable state. In Figure 6, we provide a detailed breakdown of the results based on device manufacturers. We can infer that certain manufacturers, such as Google, update their proprietary GPU drivers significantly more promptly than others.

We further investigated whether the update delay is influenced by the GPU vendors. Our research indicates that 85% of firmware images utilizing Qualcomm GPUs contain outdated GPU blobs. This is in contrast to the 64% of firmware images with ARM GPUs that are outdated. When considering the median delay for firmware updates, Qualcomm devices lag behind at 281 days, while ARM devices show a slightly better result at 231 days. It is concerning to note that Qualcomm devices, despite their significantly larger adoption rate, tend to receive blob updates less promptly.

LLVM Compiler Outdatedness: A key element of the GPU blobs is the LLVM library. We investigate their LLVM versions and present their distribution in Figure 7. An important observation is that all Qualcomm GPU blobs rely on the

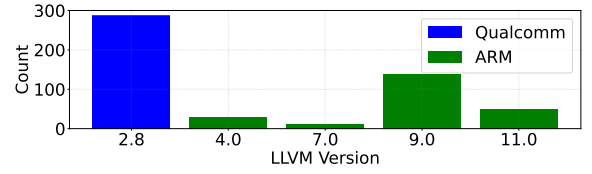


Fig. 7: LLVM Version Distribution

considerably outdated LLVM version 2.8, which is released nearly 15 years ago. In contrast, ARM appears to adopt more recent LLVM versions. Specifically, the majority of ARM blobs are based on LLVM 9.0, followed by LLVM 10.0, and LLVM 11.0. The remaining drivers employ LLVM 5.0 versions. However, these versions are also relatively outdated, with LLVM 11.0 having been released in 2020. This implies that all the GPU blobs could potentially be susceptible to a multitude of LLVM bugs that have been discovered since their respective release dates. One potential reason for Qualcomm and ARM’s reluctance to update their LLVM library could stem from the LLVM project’s emphasis on innovation and performance enhancements. This focus often results in significant API changes across different LLVM versions and managing these breaking changes is known to be a challenging and labor-intensive task.

C. Vulnerability Impacts

In this section, we present the fuzzing outcome for the most recent Qualcomm GPU blob to highlight potential security vulnerabilities. This particular blob is sourced from a Xiaomi Note 13 Pro 5G device firmware, released on February 23, 2024, with the internal blob version number 031.42.23.11.

Vulnerabilities Finding: Over the course of approximately 14 hours of fuzzing, our fuzzer identifies 289 instances of irregular program behaviors or incorrect program outputs. To ensure that the anomalies identified are not a result of a bug within our fuzzer, we validate these instances on physical devices. In brief, we substitute the reference shader in the retrace file with a variant shader. Following this, we employ Patrace to replay the updated retrace file. We utilize an offscreen rendering mode, which facilitates us to transfer the rendered frames into main memory and store them onto a hard disk file. Subsequently, we compare the frames with the reference frames using the chi-square test on their respective histograms. If the test indicates a significant difference, an offending instance is validated. We perform a thorough analysis on the offending variants shaders to pinpoint the root causes of anomalous behaviors. We categorize the causes into several distinct groups as follows.

1. Memory Access Violation: The first type of offending variants triggers memory access violation during the shader compilation stage (i.e., during the invocation of `glCompileShader`). This results in an immediate crash of the application process. We conduct an analysis of the 29 problematic cases using their stack traces. Our observations

reveal that 5 of these cases are triggered by null pointer references, while the rest result from invalid memory access violation. Upon further investigation of these invalid memory addresses, we find that most of them are low memory addresses, such as 0x7 or 0x1b. These could potentially be produced by an offset operation with a zero base address, which once again ties back to a null pointer error. However, we identify four addresses as heap memory. By instrumenting the application process with the memory checker Valgrind, we infer that these are likely triggered by a use-after-free heap error. This is alarming, as it potentially allows an attacker to construct arbitrary code execution exploits. Additionally, we aim to identify the positions of the bugs. One useful technique is to utilize the LLVM debugging command line option `-print-after-all`, which displays the output IR of each LLVM component. If a component fails to generate output IR, we can identify it as the faulty one. Our findings reveal that 7 of them are found in LLVM analysis passes, including type-based alias analysis and dead code analysis. Another 17 originate from LLVM function optimization passes, such as peephole optimization, dead code elimination, and constant expression folding. The remaining 5 are located in backend optimization passes, specifically machine code constant folding and peephole optimization.

2. Compilation Stalls: The second type of problematic shader variants results in the application process stalling during the shader compilation. Our fuzzer identifies three such instances. By attaching a GNU GDB debugger to the compiler thread and performing single-step execution, we observe that they are induced by infinite iteration of optimization passes, such as loop unrolling, control flow graph simplification, and instruction combination. In LLVM, the execution of an optimization pass generates a new set of LLVM IR, which may open up additional opportunities for optimization. To harness them, certain optimization passes are designed to operate iteratively. However, it's not unusual for logic flaws to be present within these passes and exemplary bug reports can be found in [7], [8], [9]. They prevent the IR from achieving a stable state, resulting in the optimization pass being executed repeatedly. As such, a malicious actor could potentially craft a denial-of-service shader payload, which can cause the targeted application to become unresponsive.

3. Incorrect Program Semantics: The most common type of offending shader variants causes the compiler to produce incorrect program semantics. This kind of errors may cause artifacts in the graphics output and does not necessarily lead to security vulnerability. Nevertheless, as suggested by prior research [10], we anticipate a positive correlation between the number of these behavior bugs and the count of security vulnerabilities. To identify the incorrect section of program semantics, we can generate and compare Data Dependency Graphs (DDGs) of the variant IR and the reference IR. A DDG is a directed graph where nodes represent instructions, and edges denote data dependencies between these instructions. For example, if a variable is defined in instruction A and used in instruction B, there will be an edge flowing from A to B.

Given a DDG, we start from instructions that produce shader output variables and backtrack the graph until we reach all instructions that consume the shader input variables. By doing so, we can identify the chain of instructions used to transform those input variables into an output variable. For a correct variant program, its instruction chain should be semantically equivalent to that of the reference program. Using DDGs, we pinpoint several flawed optimization passes, including instruction combination, dead code elimination, and loop simplification. For example, when dealing with a complicated control flow, these passes may incorrectly deduce that certain instructions do not affect the program's observable behavior. If such an instruction is removed, and the values it defined are still used elsewhere in the program, those values are replaced with a keyword *undef* (e.g., `%2 = fmul float %1, undef`). According to the LLVM IR specification, the keyword *undef* has special semantics; it serves as a placeholder for a constant value that can be any arbitrary value. The compiler can replace *undef* with any value (e.g., zero or one) in a completely non-deterministic manner, leading to unpredictable shader output.

Fuzzing Speed: We conduct a performance comparison of our fuzzer against the state-of-the-art GraphicsFuzz. GraphicsFuzz features a client-server architecture. We run its server program *gsl-server* on a desktop PC equipped with an AMD 5990x CPU and 128 GB of RAM for variant shader generation. Its client-side program *gles-worker* is run on a high-end ASUS ROG Phone 8 to fetch, compile, and execute these shaders. GraphicsFuzz achieves a fuzzing speed of approximately 7 shaders per second. In contrast, our fuzzer operates at a significantly faster speed, processing about 75 shaders per second. This enhanced speed enables us to uncover more vulnerabilities within the same timeframe.

Different Blob Versions: We also validate the problematic shaders across 10 versions of GPU blobs, which are evenly sampled from the dataset based on time intervals. Our findings indicate that these problematic shaders consistently trigger the same anomalous behaviors in all selected GPU blobs, suggesting that most vulnerabilities have existed for a considerable period. This aligns with the observation that Qualcomm GPU blobs have been using a very deprecated LLVM compiler for an extended period.

D. Case Study: WebGL

In the previous section, we identified shaders that can induce insecure behaviors during compilation. This section presents a case study to exploit them, which may result in denial of service (via compiler stalling) or arbitrary code execution (via use-after-free memory violations) on a target device.

The first step is to identify a widely accessible interface that can accept arbitrary shader input for compilation. One such interface is WebGL, a JavaScript API for rendering 3D graphics in a web environment. WebGL is derived from the OpenGL ES specification and can be considered a subset of OpenGL ES. In WebGL, shaders can be compiled using the `gl.compileShader` JavaScript interface. Internally, the shader programs are still processed by the vendor GPU blob,

thus triggering its vulnerabilities. WebGL are not only available in system browsers but also in apps that embed browser engines (e.g., WebKit or V8) to support in-app browsing. These apps are not uncommon and they are generally referred to hybrid apps. According to a recent empirical study [11], 149 apps among the top 500 most downloaded apps are hybrid.

To launch an attack via WebGL, an adversary can host a web page with malicious shader code on the internet. Users are then deceived into visiting this page in a system browser or a hybrid app through phishing emails, instant messages, or rogue wireless access point attacks. The exploit shader is then downloaded and executed automatically on the victim’s machine. It should be noted that Chrome introduces an intermediary layer called *Almost Native Graphics Layer Engine* (ANGLE)⁸ to optimize and sanitize input shaders before feeding them to vendor GPU blob. As a result, it necessitates the modification of our fuzzing pipeline to identify new offending shaders. Specifically, we incorporate ANGLE to sanitize generated variant shaders before feeding them into the LLVM compiler. In the course of a 4-hour fuzzing session, we identify 9 sanitized shaders that crash the Chrome browser.

V. DISCUSSION

GPU backend bug detection: Our fuzzer redirects a shader program’s LLVM IRs to an x86 backend to generate a CPU executable. It allows us to compare shader outputs without access to mobile GPUs. However, this method bypasses certain GPU backend processes (e.g., instruction scheduling or register allocation). Therefore, our fuzzer is not capable to detect vulnerabilities in these stages. This implies that the data reported in our evaluation section represents a conservative estimate of the severity. We are currently implementing an interpreter to directly execute GPU instruction output from the GPU blobs using CPUs. This allows us to uncover more bugs in the GPU backend stages. Our interpreter adopts a traditional decode-execute loop design; upon decoding a GPU instruction, the interpreter dispatches it to a corresponding function for execution. To facilitate the instruction decoding, we employ GPU disassemblers from the Mesa 3D Graphics Library⁹. Before executing the program, we randomize all register values before a GPU program begins. Upon the program’s completion, we compare the values in the register sets. Any discrepancy could indicate a potential bug.

Fuzzer Support for Vulkan and ARM: In this study, our primary focus is on the OpenGL ES graphics APIs. This is due to the fact that Android mandates each device manufacturer to provide an implementation of OpenGL ES. Recently, a number of high-end mobile devices have begun supporting Vulkan Graphics APIs. Compared to OpenGL ES, Vulkan offers reduced CPU overhead and more detailed control over GPU resources. To support Vulkan, the GPU vendors integrate an extra LLVM frontend to transform Vulkan code (i.e., SPIR-V) into LLVM IR. As such, our fuzzer can also be adapted to support Vulkan by providing an Vulkan variant shader generator.

In addition, this study places emphasis on Qualcomm GPUs, primarily due to its dominant market share. Nevertheless, our methodology can be easily replicated to ARM GPUs with little modification. Our preliminary experiments identifies 19 vulnerabilities in Mali GPU blobs. We speculate that ARM GPUs, benefiting from a more recent LLVM compiler version, may have fewer vulnerabilities compared to Qualcomm. Further investigation is warranted.

Vulnerabilities in Vendor Blobs Beyond GPUs: We carry out preliminary measurements on other types of vendor blobs in Android firmware, such as accelerometers and fingerprint readers. Unlike GPUs, these blobs are only loaded into a single process, making them likely less critical to system security due to fewer exploitation code paths. They are also sourced from a wide range of hardware manufacturers, which complicates the comparison of update patterns and vulnerability discovery. Nevertheless, further research in this area is warranted.

VI. RELATED WORK

The Android firmware encompasses a variety of software components. Extensive research efforts have been made to scrutinize their security implications.

Linux Kernel: Linux Kernel is often identified as the primary source of vulnerabilities that can compromise Android system security. Consequently, it is subject to extensive scrutiny. For instance, Zhou et al. [12] conducted research investigating the security risks arising from hardware vendors’ unsafe customizations in Linux kernel drivers. Several surveys [13], [14] also indicate that a significant portion of Android vulnerabilities are located in kernel-mode drivers. However, given that the kernel component is open source, it facilitates developers in conducting regular code reviews and swift integration of patches to mitigate vulnerabilities. Zhang et al.[1] examined the Android kernel patch process, uncovering bottlenecks in patch propagation. Similarly, Farhang et al.[15] investigated the latency between a hardware vendor releasing a patch and its integration into the Android kernel repositories. Recently, several studies [16], [17] also explore the kernel-level security module *SEAndroid* to confine system services and reduce the kernel attack surface.

AOSP System Framework: A considerable number of Android vulnerabilities also stem from the system framework [14]. Although Google frequently releases framework security patches to address these vulnerabilities, it is the responsibility of device manufacturers (OEMs) to distribute these patches to their users. Extensive research studies have examined this process. For instance, Zheng et al. [18] developed a tool called *DroidRay* to assess the security patch level of the Android system framework from various firmware images. Similarly, Hou et al. [19] evaluated a large-scale firmware dataset and found that even when a device claims to be updated to the latest system framework, there is no guarantee that all corresponding patches have been integrated by the manufacturers. Additionally, Jones et al. [20] conducted an extensive study to reveal that the Android framework security

⁸<https://github.com/google/angle>

⁹<https://www.mesa3d.org/>

updates rollout process is effectively affected by the carrier-manufacturer relationship. A recent study [21] also observes that the framework patches may not undergo thorough testing before being rolled out by manufacturers, thus resulting in low code coverage.

Pre-installed Android Apps: Pre-installed apps often come with pre-approved, highly sensitive permissions and capabilities. If these apps contain vulnerabilities, they can significantly impact system security. Consequently, extensive research has been conducted to analyze vulnerabilities in these apps. Gamba et al. [22] analyzed pre-installed Android apps to understand their potential impact on device users such as personally identifiable information leakage and pervasive user behavior tracking. Elsabagh et al. [3] applied an automated analysis system named *FirmScope* over two thousand firmware images to uncover privilege escalation vulnerabilities in pre-installed apps. Similarly, Zhang et al. [23] implemented an automated tool called *PITracker* to detect insecure intent vulnerabilities in Android pre-installed apps.

VII. CONCLUSION

This paper presents an empirical study on the security implications of proprietary vendor blobs within Android images. In particular, we focus on GPU vendor blobs, as they are loaded into every app's memory space and expose many exploitation code paths. We design an automatic analysis pipeline and a performant fuzzer to understand their update pattern and uncover security vulnerabilities. We investigate over 13,000 Android firmware images released between January 2018 and April 2024. Our study reveals that device manufacturers often neglect vendor blob updates during a device's life cycle. Approximately 82% of firmware images contain outdated GPU blobs (up to 1,128 days). Additionally, a significant number of these vendor blobs are constructed on an obsolete LLVM library released almost 15 years ago. This exposes numerous security vulnerabilities and poses immediate threats to mobile devices. Our work highlights the urgency for timely updates of these vendor blobs by device manufacturers.

REFERENCES

- [1] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3649–3666, 2021.
- [2] Liam Tung. Android security: Your phone's patch level says you're up to date, but that may be a lie, 2018.
- [3] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. {FIRMScope}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware. In *29th USENIX security symposium (USENIX Security 20)*, pages 2379–2396, 2020.
- [4] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of intrusions and malware, and vulnerability assessment*, pages 301–324. Springer, 2017.
- [5] Alastair F Donaldson. Metamorphic testing of android graphics drivers. In *2019 IEEE/ACM 4th International Workshop on Metamorphic Testing (MET)*, pages 1–1. IEEE, 2019.
- [6] Alastair F Donaldson, Hugues Evrard, and Paul Thomson. Putting randomized compiler testing into production (experience report). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2020.
- [7] Llvm bug report: Infinite loop in splitcriticalsideeffectedges. https://bugs.llvm.org/show_bug.cgi?id=45290.
- [8] Llvm bug report: Loopunroll introduces infinite loop with nested loops. https://bugs.llvm.org/show_bug.cgi?id=37859.
- [9] Llvm bug report: Simplifycfg seems to be stuck in an infinite loop. https://bugs.llvm.org/show_bug.cgi?id=33360.
- [10] Henrique Alves, Balduino Fonseca, and Nuno Antunes. Software metrics and security vulnerabilities: dataset and exploratory study. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 37–44. IEEE, 2016.
- [11] Elliott Wen, Jiaxiang Zhou, Xiapu Luo, Giovanni Russello, and Jens Dietrich. Keep me updated: An empirical study on embedded javascript engines in android apps. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 361–372. IEEE, 2024.
- [12] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*, pages 409–423. IEEE, 2014.
- [13] Shuaibing Lu, Zhechao Lin, and Ming Zhang. Kernel vulnerability analysis: A survey. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 549–554. IEEE, 2019.
- [14] Alejandro Mazuera-Rozo, Jairo Bautista-Mora, Mario Linares-Vásquez, Sandra Rueda, and Gabriele Bavota. The android os stack and its vulnerabilities: an empirical study. *Empirical Software Engineering*, 24:2056–2101, 2019.
- [15] Sadegh Farhang, Mehmet Bahadır Kirdan, Aron Laszka, and Jens Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. *arXiv preprint arXiv:1905.09352*, 2019.
- [16] Dongsong Yu, Guangliang Yang, Guozhu Meng, Xiaorui Gong, Xiu Zhang, Xiaobo Xiang, Xiaoyu Wang, Yue Jiang, Kai Chen, Wei Zou, et al. Sepal: Towards a large-scale analysis of seandroid policy customization. In *Proceedings of the Web Conference 2021*, pages 2733–2744, 2021.
- [17] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. {BigMAC}:{Fine-Grained} policy analysis of android firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 271–287, 2020.
- [18] Min Zheng, Mingshen Sun, and John CS Lui. Droidray: a security evaluation system for customized android firmwares. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 471–482, 2014.
- [19] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Chenglin Mao, Lingyun Ying, Song Liu, Xiaofeng Liu, Yuanzhi Li, Shanqing Guo, Meining Nie, et al. Can we trust the phone vendors? comprehensive security measurements on the android firmware ecosystem. *IEEE Transactions on Software Engineering*, 2023.
- [20] Kailani R Jones, Ting-Fang Yen, Sathya Chandran Sundaramurthy, and Alexandru G Bardas. Deploying android security updates: an extensive study involving manufacturers, carriers, and end users. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 551–567, 2020.
- [21] Christopher D Brant and Tuba Yavuz. A study on the testing of android security patches. In *2022 IEEE Conference on Communications and Network Security (CNS)*, pages 217–225. IEEE, 2022.
- [22] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An analysis of pre-installed android software. In *2020 IEEE symposium on security and privacy (SP)*, pages 1039–1055. IEEE, 2020.
- [23] Chennan Zhang, Shuang Li, Wenrui Diao, and Shanqing Guo. Pittracker: detecting android pendingintent vulnerabilities through intent flow analysis. In *Proceedings of the 15th ACM conference on security and privacy in wireless and mobile networks*, pages 20–25, 2022.