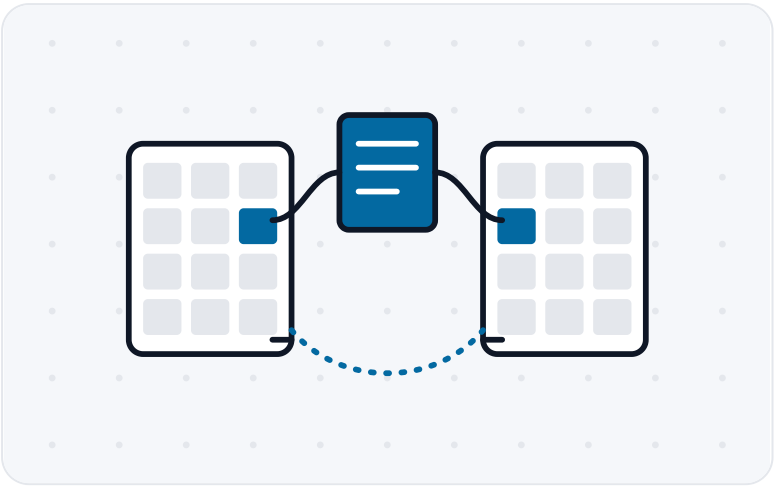
A-SIT Technology

E-Government Innovation Center

17 May 2024 Overview · Mobile Security · 17 min read

Potential Covert Channel Through Kernel Samepage Merging on Android

Florian Draschbacher



Authors	Florian Draschbacher
Version	2024-05-14
Source	https://technology2.a-sit.at/en/publications/potential-covert-channel-through-kernel-samepage-merging-on-android/
Note	Machine translation.

Key findings

- KSM is not enabled in the standard Android kernel configuration, but some vendor devices (e.g. Samsung Galaxy A51, OnePlus) have enabled it.
- On OnePlus devices, the KSM daemon can be started without root rights via a switch in the system settings.
- KSM can form a covert channel through timing effects on write access, transmitting 1 bit per memory page.
- Such a covert channel could bypass the isolation of the Android Virtualization Framework or the SDK Runtime.

- Per Possemato et al., 20% of almost 3000 Android ROMs deviated from Google's CDD; 190 of 2396 kernel images in their kernel configuration.

Kernel Samepage Merging (KSM) is a mechanism in the Linux kernel that allows identical pages in memory to be shared between multiple processes, thereby improving memory usage efficiency. To utilize KSM, a program must explicitly mark a memory area as mergeable. If a page with the same content is marked as mergeable by multiple processes, one of the copies is deleted, and a shared physical memory page is referenced by the page tables of both processes. Since Android is based on the Linux kernel, KSM can also be used here. While there is already research exploring the security implications of KSM on virtual machines, projects on KSM in mobile devices are scarcely found in the literature.

In the scope of this project, we first explain the Android-Linux kernel, the possibility of modification by device manufacturers, and the security issues that have arisen from such modifications in the past. Subsequently, we discuss Kernel Same Page Merging on Android, explain existing covert and side channels, and discuss how KSM on Android could enable new such channels.

1. Introduction

With a market share of over 70%, Android is the most widely used operating system for tablets and smartphones. The licensing model of the mobile operating system was also decisive for this success. It is developed as an open-source project under the leadership of Google. Any manufacturer of mobile devices can modify the source code as it sees fit, for example to adapt it to the installed hardware. As a result, Android devices are offered by a large number of different manufacturers.

In the past, however, this abundance of devices has proven to be not only an advantage of the Android platform. The possibility certainly ensures a never-ending stream of new Android devices, but these devices repeatedly stand out for their lack of security. Many manufacturers also use the option to customise Android in order to implement novel features that are meant to set them apart from the competition. In this development, the visible user experience is usually the main focus, while security is often only an afterthought. Manufacturers also frequently present so many new devices in such a short time that they lack the capacity to supply these devices with security updates in the long term. Once a security vulnerability exists, the device may never again be operated securely.

A topic that has only recently moved into the focus of research is side channels and covert channels. Side channels generally refer to the possibility of drawing conclusions about the internal state of a program from side effects of its execution (such as power consumption or execution time). In a covert channel, these phenomena are used to break process isolation, that is, to let processes communicate with each other that actually have no way of doing so (if these side effects are disregarded). Attackers can use this method, for example, to extract data despite existing security precautions.

This project investigated which manufacturer customisations are suitable for implementing a covert channel. In particular, it addresses Kernel Samepage Merging, which has already been used in other Linux environments in the past to implement covert channels that were able to bypass virtualisation boundaries.

In addition, an overview of the development process of the Linux kernel trees in the Android ecosystem was compiled, together with a survey of existing literature on security vulnerabilities in the Android Linux kernel caused by customisations, and on existing side and covert channels on Android. The potential of covert channels with regard to current developments on the Android platform was also studied and is discussed in this report.

2. Background

This section explains the technologies that are necessary for understanding the later explanations.

2.1. Android

The Android platform is one of the most widely used mobile operating systems in the world and enjoys enormous popularity among millions of users worldwide. From smartphones and tablets to embedded systems and IoT devices, Android is present across a wide variety of device types and applications.

Android is based on the Linux kernel. All components above it in the software stack (“userspace”) were developed for the Android operating system specifically for the requirements of mobile devices (low power consumption, touch display, …). These userspace components are available under the Apache 2.0 licence. The licence allows arbitrary modifications without requiring the source code of the derivative to be published under the same licence.

Android allows the execution of applications installed by the user. Each of these is executed in its own isolated environment (“sandbox”), for example to prevent a malicious app from stealing data from another app or influencing its execution. To access resources that are shared with other apps (such as the public file system or sensors), apps must hold a permission. For particularly dangerous (sensitive) permissions, the user must explicitly grant access by means of a dialogue.

Even unprivileged standard applications (without permissions) can communicate with each other using inter-process communication (IPC) functionality. However, because of the clearly defined interfaces, it is easy to intercept communication implemented in this way. If the communication is to be concealed, covert channels can be used for this purpose. In addition, certain applications have no way of using IPC. These can fall back on covert channels to circumvent this limitation and still communicate with other processes.

2.2. Linux

Linux is an operating system kernel that is licensed under the GNU GPL. It is therefore not only open source, but all derived works must also be published as open source under the same licence. For this reason, manufacturers of Android devices are also obliged to make the source code freely accessible for every Android kernel adapted to a device. This also allows, for example, insight into the configuration used for compilation.

The Linux kernel configures and manages the device hardware and makes it available in a uniform form to higher software layers. The uniform interfaces of the Linux kernel offer, for example, basic functionality such as memory management, process isolation, threading, file systems, locking, inter-process communication (IPC), networking, etc. It should be emphasised here that some of these mechanisms are also used to implement central security mechanisms in the Android operating system. For example, Android uses the process isolation and the Discretionary Access Control (DAC) of Linux to implement the app sandbox.

3. Android Common Kernel and Generic Kernel Image

In the past, manufacturers had to maintain a separate Android version for each mobile device model. This version then contained, for example, the appropriate drivers for the installed hardware, such as the main processor (System-on-Chip, SoC), communication modules (Wi-Fi, Bluetooth, LTE, NFC, ...), cameras and other sensors, and so on. More precisely, the process worked as shown in Figure 1 and described below.

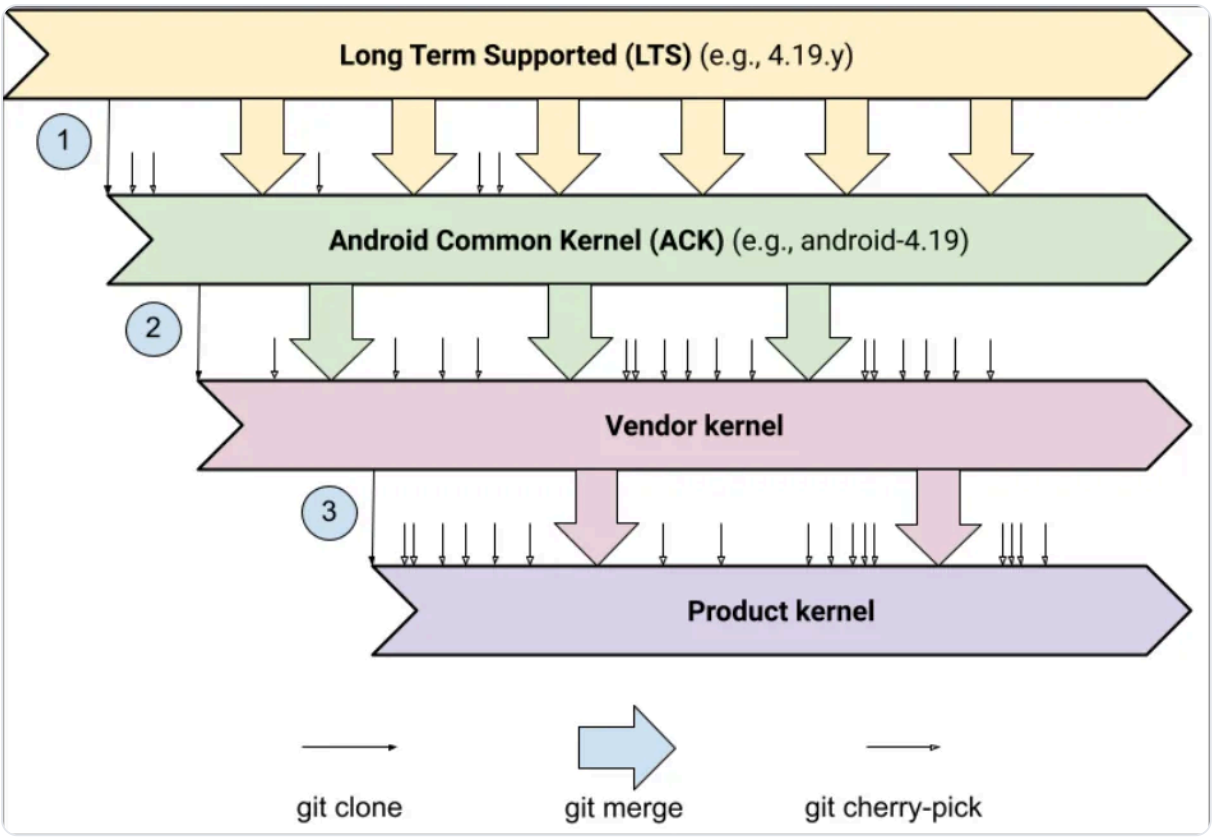


Figure 1. The original development process of Android Linux kernels. Source: Android Open Source Project [1].

The Linux project regularly publishes a Long Term Stable (LTS) kernel. This is a Linux kernel that is officially supported for a defined period (usually at least 2 years). When a security vulnerability is found, a patch (a fix for the vulnerability) is published for all currently officially supported LTS kernels.

For each LTS kernel, the AOSP project (Google) maintains an Android Common Kernel (ACK). This is a modified variant of the LTS kernel that is extended with Android-specific functionality (such as special drivers that are not hardware-specific). Google maintains these Android Common Kernels over a longer period (as long as they are officially supported by a reasonably current Android version), that is, it merges security patches from the corresponding LTS kernel.

Each manufacturer in turn derived its own variant from these ACK kernels, extended with manufacturer-specific functionality (such as self-written functionality that is used on all of the manufacturer's devices).

Finally, each manufacturer derived a device kernel from its device-agnostic ACK variant, which was adapted specifically to a particular device model. This is where the hardware drivers mentioned at the beginning were incorporated.

Because a device kernel is derived through several levels in this complex way, it is understandable that the regular provision of security updates was a challenge for device manufacturers. At a point when no new revenue could be generated from the device that had already been sold, it seemed natural to some manufacturers to stop investing resources in software updates, even if these contained fixes for critical security vulnerabilities.

To counter this problem, Google started the Generic Kernel Image (GKI) project. It provides that all Android devices are operated with a GKI kernel image (compiled kernel) supplied by Google. Hardware-specific and manufacturer-specific components in the kernel can be integrated via a Kernel Module Interface (KMI).

Officially, Android devices that ship with Android 11 or later must at least be compatible with the GKI kernel. According to official Google requirements, devices with Android 13 or later must be shipped with the GKI kernel. These kernels can be identified by means of the uname string. For GKI kernels, the version designation always follows the scheme `Version.PatchLevel.SubLevel-AndroidRelease-KmiGeneration-suffix`. The kernel with the version designation 5.4.42-android12-0-00544-ged21d463f856 is therefore, for example, a GKI kernel. Using this information, the kernel package (and the corresponding source code) can also be downloaded from Google's website.

An analysis of the kernels of current devices shows, however, that Google allows exemptions for some manufacturers. For example, the kernel of a current Samsung device (Galaxy S23) has the version designation 5.15.94-android13-8-27763874-abS911BXXU3BWJM. Closer examination shows that it is in fact not a GKI kernel. The image contains symbols that are not contained in the GKI kernel. Devices are also still in circulation that use Android versions (and kernel versions) for which GKI cannot yet, or need not yet, be used.

Since the uniform GKI kernel is therefore not used uniformly despite corresponding requirements, manufacturer-specific changes to the kernel configuration remain possible.

4. Manufacturer-specific kernel customisations

Several studies have already concluded in the past that manufacturer-specific customisations to the kernel have serious consequences for device security.

Possemato et al. [2] analysed almost 3000 Android ROMs in 2021 with regard to their compliance with Google's compatibility guidelines. These guidelines are intended to ensure that Android devices from different manufacturers meet certain minimum standards with regard to security and user experience. They are published as the Compatibility Definition Document (CDD). Some aspects of the document can also be covered by automated tests. For a device to be sold as an Android device, it (or the operating system adapted by the manufacturer) must meet this CDD.

The researchers found that 20% of the examined ROMs nevertheless deviated from the CDD. In almost 8% (190 of 2396 kernel images), guidelines relating to the Linux kernel configuration that were mandatory in the CDD were not complied with. These are configuration flags that enable hardening functionality against memory exploits. On affected kernels, it is therefore easier for attackers to exploit a memory management vulnerability. In addition, in 10% of the examined kernels, strong recommendations of the CDD were not implemented. In many cases, for example, Kernel Address Space Layout Randomization was not enabled. This function also makes it harder for attackers to exploit a memory vulnerability.

These deviations from the CDD are in each case kernel customisations. The manufacturers decided on their own authority not to meet strong recommendations or unconditional requirements. One possible reason is the performance overhead of the disabled kernel functionality.

5. Side channels and covert channels on Android

There are also already some contributions in the literature on side and covert channels on Android.

5.1. Side channels

Side channels are alternative communication channels that reveal information through unconventional means, often through the unintentional exploitation of physical or logical properties of a system. These channels can reveal vulnerabilities that attackers can use to read out sensitive data without breaking the actual security mechanisms. Common examples of side channels are:

- Timing attacks: Exploiting the time a system needs for certain operations in order to obtain secret information.
- Electromagnetic leaks: Analysing the electromagnetic radiation emitted by electronic devices in order to extract information.
- Power analysis: Monitoring the power consumption of a device in order to obtain encryption keys or other sensitive data.

On Android, work on all three categories of side channels has been published in the past. Palfinger et al. [3] presented AndroTIME, a solution that automatically examines the timing of Android APIs in order to identify side channels. They were able to find several side channels that could be used, among other things, to determine the installed apps, active user accounts, the existence of files or browser logins without the corresponding permission.

As early as 2016, Genkin et al. [4] showed that the electromagnetic emissions of a smartphone can be used to extract keys that are currently being used in cryptographic operations. To do so, they placed a magnetic probe near the mobile device. As an example of a practical attack, the researchers mention the possibility of attaching this probe, for example, underneath a table on which the victim places their smartphone. Among other things, the researchers used the approach to read out parts of an ECDSA key from OpenSSL on Android.

Approaches to power analysis use, among other things, manipulated chargers. Cronin et al. [5] presented a power line side channel that can capture touchscreen interactions from the power consumption of a charging mobile device. This exploits the fact that touch events often lead to visual feedback that causes fluctuations in the power consumption of the display. If the power consumption is captured precisely enough, the exact point in time within the screen refresh can be used to determine where on the screen the visual feedback was shown, that is, where the touch event took place. As the researchers show, this approach can be used, for example, to reconstruct the unlock PIN of a smartphone solely from its power consumption.

5.2. Covert channels

Covert channels are communication paths that are used to transmit information while bypassing the normal security policies. They are often created deliberately in order to hide or obscure data. Covert channels are a method of transmitting information between processes that actually should not have communication rights with each other.

Here too, similar physical or logical effects can be used as for side channels. Gasior et al. [6] implemented a covert channel that communicates hidden content through variations in the timing of TCP messages. Spolaor et al. [7] used an app installed on the smartphone to control power consumption, by deliberately executing CPU-intensive tasks, in such a way that information can be encoded that is read by the charging station.

6. Kernel Same Page Merging on Android

Linux Kernel Samepage Merging (KSM) is a mechanism in the Linux kernel that optimises memory usage by identifying and merging identical memory contents (page by page, that is, usually in blocks of 4096 bytes) between different processes or virtual machines. Several conditions must be met in order to use KSM:

- KSM must be compiled into the kernel. To do so, the CONFIG_KSM flag must be set in the kernel configuration at the kernel's compile time.
- The memory areas must each be marked as mergeable. To do so, the process must mark the memory area using a madvise call with the argument MADV_MERGEABLE.
- The KSM daemon must be running. It scans the areas marked as mergeable at regular intervals in order to detect and merge duplicates. Both the interval and the number of memory pages checked per pass are configurable.

6.1. KSM on Android

Although KSM is not enabled in the standard Android kernel configuration, there are current Android devices on which it is. This is the result of customisations by the device manufacturer.

An example of such a device is the Samsung Galaxy A51, the best-selling Android smartphone in the first quarter of 2020. The configuration file (extracted either from the device or from the kernel's source code package) shows that CONFIG_KSM is enabled. Indeed, the corresponding sysfs entries for configuration can also be found. It turns out, however, that the daemon is not running. Why Samsung enables the flag only to then not start the daemon is not understandable. Perhaps the daemon is only started under certain conditions. For research purposes, the daemon can nevertheless be started by first rooting the device. Only the root user has the ability to change the configuration of the daemon, for example in order to start its execution.

In addition, some devices from the manufacturer OnePlus have been equipped with KSM. There, the daemon can also be started without root rights. Specifically, this is done via a switch in the system settings.

Finally, a development must be mentioned here that could lead to further manufacturers (possibly even Google in the GKI) enabling KSM soon. With Android 13, the Android Virtualization Framework (AVF) was introduced [8]. It allows simple Android instances to be started in virtual environments within an Android device. The system is intended to isolate individual components even further from the rest of the system than the app sandbox does. Since KSM was originally created to minimise the memory requirements of virtualised environments, it cannot be ruled out that KSM will find its way into the GKI kernel as part of the AVF.

6.2. Application for covert channels

Like other memory deduplication mechanisms, KSM can be used to realise side channels and covert channels. Since it is relatively easy for a process to prevent a KSM-based side channel (by not marking sensitive memory areas as mergeable), only covert channels are addressed here.

Covert channels can be implemented with KSM via timing effects. As a naive approach, process A can mark one of two possible memory pages of defined (but different) content as mergeable. Process B marks both memory pages as mergeable. Process B then measures how long the write access to the two memory pages takes. Longer write times are to be expected for the page that was also marked as mergeable by process A. This is because the kernel has to resolve the merge, that is, it first has to copy the merged page and map it anew into the process. The information about which of the two pages was marked as mergeable by process A therefore transmits 1 bit of information to process B. As for other cache-based covert channels, the transmission rate can be optimised [9].

6.3. Use cases on Android

As mentioned at the beginning, Android apps can communicate with each other via IPC on Android even without special permissions. Nevertheless, in some scenarios it is desirable not to use clearly defined programming interfaces for communication. In addition, in some scenarios the option of using IPC is not available.

Worth mentioning in this context is the Android Virtualization Framework (AVF), which was already referred to above. A covert channel (for example based on KSM) is suitable here for bypassing the isolation of the individual virtualised environments.

In addition, a new possible application arises in connection with the SDK Runtime as part of the larger Privacy Sandbox project, which Google is gradually integrating into the Android operating system. Here, advertising libraries are to no longer be executed within the app process, but in a separate process. This isolation, too, could be broken by means of covert channels.

7. Summary

In this report, we examined the development process of the Android Linux kernel, as well as various ways in which manufacturer customisations cause security problems. We then focused our descriptions on side channels and covert channels on Android, and described how Linux Kernel Samepage Merging can be used to implement covert channels that bypass the isolation of virtual environments of the Android Virtualization Framework (AVF) or of the SDK sandbox.

References

[1] Google, „The Generic Kernel Image (GKI) project,“ 03 2024. [Online]. Available: https://source.android.com/docs/core/architecture/kernel/generic-kernel-image.

[2] A. Possemato, S. Aonzo, D. Balzarotti und Y. Fratantonio, „Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization,“ in IEEE Symposium on Security and Privacy (SP), 2021.

[3] G. Palfinger, B. Prünster und D. J. Ziegler, „AndroTIME: Identifying Timing Side Channels in the Android API,“ in IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2020.

[4] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer und Y. Yarom, „ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels,“ in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2016.

[5] P. Cronin, X. Gao, C. Yang und H. Wang, „Charger-Surfing: Exploiting a Power Line Side-Channel for Smartphone Information Leakage,“ in 30th USENIX Security Symposium, 2021.

[6] W. Gasior und L. Yang, „Exploring Covert Channel in Android Platform,“ in International Conference on Cyber Security, 2012.

[7] R. Spolaor, L. Abudahi, V. Moonsamy, M. Conti und R. Poovendran, „No Free Charge Theorem: A Covert Channel via USB Charging Cable on Mobile Devices,“ in Applied Cryptography and Network Security, 2017.

[8] S. Patil, „Virtual Machine as a core Android Primitive,“ 05 12 2023. [Online]. Available: https://android-developers.googleblog.com/2023/12/virtual-machines-as-core-androidprimitive.html.

[9] Y. Yarom und K. Falkner, „FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack,“ in 23rd USENIX Security Symposium, 2014.