Frequently Asked Questions

This page contains answers to frequently asked questions about GrapheneOS. It's not an overview of the project or a list of interesting topics about GrapheneOS. Many of the answers would be nearly the same or identical for the latest release of the Android Open Source Project. The goal is to provide high quality answers to some of the most common questions about the project, so the developers and other community members can link to these and save lots of time while also providing higher quality answers.

Table of contents

Device support

Which devices are supported?

Devices sold in partnership with specific carriers may be locked by the carrier, which will prevent installing GrapheneOS. This is primarily an issue with US carriers and isn't common elsewhere in the world. To avoid this, either don't buy a carrier device, or make sure it can be unlocked. It's the same hardware/firmware/software either way but carriers dislike having devices able to bypass their paywall for tethering, etc., so they disable it for the devices they sell as part of contracts.

GrapheneOS has official production support for the following devices:

- Pixel 9a (tegu)
- Pixel 9 Pro Fold (comet)
- Pixel 9 Pro XL (komodo)
- Pixel 9 Pro (caiman)
- Pixel 9 (tokay)
- Pixel 8a (akita)
- Pixel 8 Pro (husky)
- Pixel 8 (shiba)
- Pixel Fold (felix)
- Pixel Tablet (tangorpro)
- Pixel 7a (lynx)
- Pixel 7 Pro (cheetah)
- Pixel 7 (panther)
- Pixel 6a (bluejay)
- Pixel 6 Pro (raven)
- Pixel 6 (oriole)

The release tags for these devices have official builds and updates available. These devices meet the stringent privacy and security standards and have substantial upstream and downstream hardening specific to the devices.

We provide extended support releases as a stopgap for users to transition to the far more secure current generation devices.

Many other devices are supported by GrapheneOS at a source level, and it can be built for them without modifications to the existing GrapheneOS source tree. Device support repositories for the Android Open Source Project can simply be dropped into the source tree, with at most minor modifications within them to support GrapheneOS. In most cases, substantial work beyond that will be needed to bring the support up to the same standards. For most devices, the hardware and firmware will prevent providing a reasonably secure device, regardless of the work put into device support.

GrapheneOS does not support being used as a Generic System Image, which only exists for development/testing purposes and isn't usable for GrapheneOS since we require kernel changes and the userspace part of the OS cannot run on top of a kernel without the required functionality. The generic targets simply run on top of the underlying device support code (firmware, kernel, device trees, vendor code) rather than shipping it and keeping it updated. It would be possible to ship generic system images with separate updates for the device support code. However, it would be drastically more complicated to maintain and support due to combinations of different versions and it would cause complications for the hardening done by GrapheneOS. The motivation doesn't exist for GrapheneOS, since full updates with deltas to minimize bandwidth can be shipped for every device and GrapheneOS is the only party involved in providing the updates. For the same reason, it has little use for the ability to provide out-of-band updates to system image components including all the apps and many other components.

Some of the GrapheneOS sub-projects support other operating systems on a broader range of devices. Device support for Auditor and AttestationServer is documented in the overview of those projects. The hardened_malloc project supports nearly any Linux-based environment due to official support for musl, glibc and Bionic along with easily added support for other environments. It can easily run on non-Linux-based operating systems too, and supporting some like HardenedBSD is planned but depends on contributors from those communities.

Which devices are recommended?

Devices sold in partnership with specific carriers may be locked by the carrier, which will prevent installing GrapheneOS. This is primarily an issue with US carriers and isn't common elsewhere in the world. To avoid this, either don't buy a carrier device, or make sure it can be unlocked. It's the same hardware/firmware/software either way but carriers dislike having devices able to bypass their paywall for tethering, etc., so they disable it for the devices they sell as part of contracts.

We strongly recommend only purchasing one of the following devices for GrapheneOS due to better security and a long minimum support guarantee from launch for full security updates and other improvements:

- Pixel 9a
- Pixel 9 Pro Fold
- Pixel 9 Pro XL
- Pixel 9 Pro
- Pixel 9
- Pixel 8a
- Pixel 8 Pro
- Pixel 8

8th/9th generation Pixels provide a minimum guarantee of 7 years of support from launch instead of the previous 5 year minimum guarantee. 8th/9th generation Pixels also bring support for the incredibly powerful hardware memory tagging security feature as part of moving to new ARMv9 CPU cores. GrapheneOS uses hardware memory tagging by default to protect the base OS and known compatible user installed apps against exploitation, with the option to use it for all apps and opt-out on a case-by-case basis for the few incompatible with it.

Which devices will be supported in the future?

Devices are carefully chosen based on their merits rather than the project aiming to have broad device support. Broad device support is counter to the aims of the project, and the project will eventually be engaging in hardware and firmware level improvements rather than only offering suggestions and bug reports upstream for those areas. Much of the work on the project involves changes that are specific to different devices, and officially supported devices are the ones targeted by most of this ongoing work.

Hardware, firmware and software specific to devices like drivers play a huge role in the overall security of a device. The goal of the project is not to slightly improve some aspects of insecure devices and supporting a broad set of devices would be directly counter to the values of the project. A lot of the low-level work also ends up being fairly tied to the hardware.

Non-exhaustive list of requirements for future devices, which are standards met or exceeded by current Pixel devices:

- Support for using alternate operating systems including full hardware security functionality
- Complete monthly Android Security Bulletin patches without any regular delays longer than a week for device support code (firmware, drivers and HALs)
- At least 5 years of updates from launch for device support code with phones (Pixels now have 7) and 7 years with tablets
- Device support code updated to new monthly, quarterly and yearly releases of AOSP within several months to provide new security improvements (Pixels receive these in the month they're released)
- Linux 6.1, 6.6 or 6.12 Generic Kernel Image (GKI) support
- Hardware accelerated virtualization usable by GrapheneOS (ideally pKVM to match Pixels but another usable implementation may be acceptable)
- Hardware memory tagging (ARM MTE or equivalent)
- Hardware-based coarse grained Control Flow Integrity (CFI) for baseline coverage where type-based CFI isn't used or can't be deployed (BTI/PAC, CET IBT or equivalent)
- PXN, SMEP or equivalent
- PAN, SMAP or equivalent
- Isolated radios (cellular, Wi-Fi, Bluetooth, NFC, etc.), GPU, SSD, media encode / decode, image processor and other components
- Support for A/B updates of both the firmware and OS images with automatic rollback if the initial boot fails one or more times
- Verified boot with rollback protection for firmware
- Verified boot with rollback protection for the OS (Android Verified Boot)
- Verified boot key fingerprint for yellow boot state displayed with a secure hash (non-truncated SHA-256 or better)
- StrongBox keystore provided by secure element
- Hardware key attestation support for the StrongBox keystore
- Attest key support for hardware key attestation to provide pinning support
- Weaver disk encryption key derivation throttling provided by secure element
- Insider attack resistance for updates to the secure element (Owner user authentication required before updates are accepted)

- Inline disk encryption acceleration with wrapped key support
- 64-bit-only device support code
- Wi-Fi anonymity support including MAC address randomization, probe sequence number randomization and no other leaked identifiers
- Support for disabling USB data and also USB as a whole at a hardware level in the USB controller
- Reset attack mitigation for firmware-based boot modes such as fastboot mode zeroing memory left over from the OS and delaying opening up attack surface such as USB functionality until that's completed
- Debugging features such as JTAG or serial debugging must be inaccessible while the device is locked

In order to support a device, the appropriate resources also need to be available and dedicated towards it. Releases for each supported device need to be robust and stable, with all standard functionality working properly and testing for each of the releases.

The expectation is for people to buy a secure device meeting our requirements to run GrapheneOS. Broad device support would imply mainly supporting very badly secured devices unable to support our features. It would also take a substantial amount of resources away from our work on privacy and security, especially since a lot of it is closely tied to the hardware such as the USB-C port control and fixing or working around memory corruption bugs uncovered by our features. We plan to partner with OEMs to have devices produced meeting all our requirements, providing additional privacy/security features beyond them and ideally shipping with GrapheneOS rather than massively lowering our standards.

Why are older devices no longer supported?

GrapheneOS aims to provide reasonably private and secure devices. It cannot do that once device support code like firmware, kernel and vendor code is no longer actively maintained. Even if the community was prepared to take over maintenance of the open source code and to replace the rest, firmware would present a major issue, and the community has never been active or interested enough in device support to consider attempting this. Unlike many other platforms, GrapheneOS has a much higher minimum standard than simply having devices fully functional, as they also need to provide the expected level of security. It would start to become realistic to provide substantially longer device support once GrapheneOS controls the hardware and firmware via custom hardware manufactured for it. Until then, the lifetime of devices will remain based on manufacturer support. It's also important to keep in mind that phone vendors claiming to provide longer support often aren't actually doing it and some never even ship firmware updates when the hardware is still supported by the vendors...

GrapheneOS also has high standards for the privacy and security properties of the hardware and firmware, and these standards are regularly advancing. The rapid pace of improvement has been slowing down, but each hardware generation still brings major improvements. Over time, the older hardware starts to become a substantial liability and holds back the project. It becomes complex to simply make statements about the security of the project when exceptions for old devices need to be listed out. Our current standards for security based on current generation devices are only applied to new devices rather than ones which used to meet previous standards. Devices remain supported until end-of-life despite no longer meeting our current standards.

Which devices did GrapheneOS support in the past?

The following end-of-life devices are no longer supported:

- Pixel 5a (barbet)
- Pixel 5 (redfin)
- Pixel 4a (5G) (bramble)
- Pixel 4a (sunfish)
- Pixel 4 XL (coral)
- Pixel 4 (flame)
- Pixel 3a XL (bonito)
- Pixel 3a (sargo)
- Pixel 3 XL (crosshatch)
- Pixel 3 (blueline)
- Pixel 2 XL (taimen)
- Pixel 2 (walleye)
- Pixel XL (marlin)
- Pixel (sailfish)
- Nexus 6P (angler)
- Nexus 5X (bullhead)
- Nexus 9 (flounder)
- Nexus 5 (hammerhead)
- Samsung Galaxy S4 (jflte)

GrapheneOS also used to provide official support for the following development boards (without publishing official builds) but dropped support due to lack of community interest and lack of hardware availability:

- HiKey 960 (hikey960)
- HiKey (hikey)

How long can GrapheneOS support my device for?

GrapheneOS can only fully provide security updates to a device provided that the OEM is releasing them. When an OEM is no longer providing security updates, GrapheneOS aims to provide harm reduction releases for devices which only have a minimum of 3 years support. Extended support updates at minimum will be done until the next Android version. It is likely that we will make a decision around harm reduction releases for other devices with longer lifetimes in Q4 2024. Harm reduction releases do not have complete security patches because it's not possible to provide full security updates for the device without OEM support and they are intended to buy users some limited time to migrate to a supported device.

Device	OEM minimum support end	OEM minimum support length
Google Pixel 9a	April 2032	7 years
Google Pixel 9 Pro Fold	August 2031	7 years
Google Pixel 9 Pro XL	August 2031	7 years
Google Pixel 9 Pro	August 2031	7 years
Google Pixel 9	August 2031	7 years
Google Pixel 8a	May 2031	7 years
Google Pixel 8 Pro	October 2030	7 years
Google Pixel 8	October 2030	7 years
Google Pixel Fold	June 2028	5 years
Google Pixel Tablet	June 2028	5 years
Google Pixel 7a	May 2028	5 years
Google Pixel 7 Pro	October 2027	5 years
Google Pixel 7	October 2027	5 years
Google Pixel 6a	July 2027	5 years
Google Pixel 6 Pro	October 2026	5 years
Google Pixel 6	October 2026	5 years

Security and privacy

How is disk encryption implemented?

GrapheneOS uses an enhanced version of the modern filesystem-based disk encryption implementation in the Android Open Source Project. The officially supported devices have substantial hardware-based support for enhancing the security of the encryption implementation. GrapheneOS has full support for the hardware-based encryption features just as it does with other hardware-based security features.

Firmware and OS partitions are identical copies of the images published in the official releases. The authenticity and integrity of these partitions is verified from a root of trust on every boot. No data is read from any of these images without being cryptographically verified. Encryption is out of scope due to the images being publicly available. Verified boot offers much stronger security properties than disk encryption. Further details will be provided in another section on verified boot in the future.

The data partition stores all of the persistent state for the operating system. Full disk encryption is implemented via filesystem-based encryption with metadata encryption. All data, file names and other metadata is always stored encrypted. This is often referred to as file-based encryption but it makes more sense to call it filesystem-based encryption. It's implemented by the Linux kernel as part of the ext4 / f2fs implementation rather than running a block-based encryption layer. The advantage of filesystem-based encryption is the ability to use fine-grained keys rather than a single global key that's always in memory once the device is booted.

Disk encryption keys are randomly generated with a high quality CSPRNG and stored encrypted with a key encryption key. Key encryption keys are derived at runtime and are never stored anywhere.

Sensitive data is stored in user profiles. User profiles each have their own unique, randomly generated disk encryption key and their own unique key encryption key is used to encrypt it. The owner profile is special and is used to store sensitive system-wide operating system data. This is why the owner profile needs to be logged in after a reboot before other user profiles can be used. The owner profile does not have access to the data in other profiles. Filesystem-based encryption is designed so that files can be deleted without having the keys for their data and file names, which enables the owner profile to delete other profiles without them being active.

GrapheneOS enables support for ending secondary user profile sessions after logging into them. It adds an end session button to the lockscreen and in the global action menu accessed by holding the power button. This fully purges the encryption keys and puts the profiles back at rest. This can't be done for the owner profile without rebooting due to it encrypting the sensitive system-wide operating system data.

Using a secondary profile for regular usage allows you to make use of the device without decrypting the data in your regular usage profile. It also allows putting it at rest without rebooting the device. Even if you use the same passphrase for multiple profiles, each of those profiles still ends up with a unique key encryption key and a compromise of the OS while one of them is active won't leak the passphrase. The advantage to using separate passphrases is in case an attacker records you entering it.

File data is encrypted with AES-256-XTS and file names with AES-256-CTS. A unique key is derived using HKDF-SHA512 for each regular file, directory and symbolic link from the per-profile encryption keys, or the global encryption key for non-sensitive data stored outside of profiles. The directory key is used to encrypt the file names. GrapheneOS increases the file name padding from 16 bytes to 32 bytes. AES-256-XTS with the global encryption key is also used to encrypt filesystem metadata as a whole beyond the finer-grained file name encryption.

The OS derives a password token from the profile's lock method credential using scrypt. This is used as the main input for key derivation.

The OS stores a high entropy random value as the Weaver token on the secure element (Titan M on Pixels) and uses it as another input for key derivation. The Weaver token is stored alongside a Weaver key derived by the OS from the password token. In order to retrieve the Weaver token, the secure element requires the correct Weaver key. A secure internal timer is used to implement hardware-based delays for each attempt at key derivation. It quickly ramps up to 1 day delays before the next attempt. Weaver also provides reliable wiping of data since the secure element can reliably wipe a Weaver slot. Deleting a profile will wipe the corresponding Weaver slot and a factory reset of the device wipes all of the Weaver slots. The secure element also provides insider attack resistance preventing firmware updates before authenticating with the owner profile.

Standard delays for encryption key derivation enforced by the secure element:

- 0 to 4 failed attempts: no delay
- 5 failed attempts: 30 second delay
- 6 to 9 failed attempts: no delay
- 10 to 29 failed attempts: 30 second delay
- 30 to 139 failed attempts: $30 \times 2^{\lfloor (n - 30) \div 10 \rfloor}$ where n is the number of failed attempts. This means the delay doubles after every 10 attempts. There's a 30 second delay after 30 failed attempts, 60s after 40, 120s after 50, 240s after 60, 480s after 70, 960s after 80, 1920s after 90, 3840s after 100, 7680s after 110, 15360s after 120 and 30720s after 130
- 140 or more failed attempts: 86400 second delay (1 day)

Invalid input outside the minimum or maximum length limits of the UI won't trigger an attempt at authentication or key derivation.

GrapheneOS only officially supports devices with Weaver. The fallback implementation for devices without it is out-of-scope for this FAQ.

The password token, Weaver token and other values like the OS verified boot key are used by the TEE as inputs to a hardware-bound key derivation algorithm provided by the SoC. The general concept is having the SoC perform hardware accelerated key derivation using an algorithm like AES or HMAC keyed with a hard-wired hardware key inaccessible to software or firmware. This is meant to prevent offloading a brute force attack onto more powerful hardware without an expensive process of extracting the hardware key from the SoC.

Many apps use the hardware keystore, their own encryption implementation or a combination of those to provide an additional layer of encryption. As an example, an app can use the hardware keystore to encrypt their data with a key only available when the device is unlocked to keep their data at rest when the profile is locked but not logged out. This is beyond the scope of this FAQ section.

Can apps spy on the clipboard in the background or inject content into it?

As of Android 10, only the configured default input method editor (your keyboard of choice) and the currently focused app can read the clipboard content. Apps without focus cannot read the clipboard. This is a stricter restriction than preventing apps in the background from reading it, since an app in the foreground or a foreground service cannot read it, only the foreground app that's currently focused. Clipboard managers need to be implemented by the keyboard chosen as the default by the user.

Both background and foreground apps can write data to the clipboard.

GrapheneOS previously restricted background clipboard access as a much earlier and slightly less strict implementation of this feature. It provided a toggle for users to whitelist clipboard managers, which is no longer needed now that keyboards are expected to provide it.

As of Android 12, the user is notified when an app reads clipboard content which was set by a different app. This notice is enabled by default and can be toggled under Settings > Security & privacy > Privacy controls > Show clipboard access.

Can apps access hardware identifiers?

As of Android 10, apps cannot obtain permission to access non-resettable hardware identifiers such as the serial number, MAC addresses, IMEIs/MEIDs, SIM card serial numbers and subscriber IDs. Only privileged apps included in the base system with `READ_PRIVILEGED_PHONE_STATE` whitelisted can access these hardware identifiers. Apps targeting Android 10 will receive a `SecurityException` and older apps will receive an empty value for compatibility. The default SMS app is a special case that's given access to the IMEI, which is normally the GrapheneOS fork of the AOSP Messaging app unless users explicitly change it to another app.

Since these restrictions became standard, GrapheneOS only makes a small change to remove a legacy form of access to the serial number by legacy apps, which was still around for compatibility. It used to need more extensive changes such as disallowing access to the serial number but those restrictions are now standard.

Apps can determine the model of the device (such as it being a Pixel 6) either directly or indirectly through the properties of the hardware and software. There isn't a way to avoid this short of the OS supporting running apps in a virtual machine with limited functionality and hardware acceleration. Hiding the CPU/SoC model would require not even using basic hardware virtualization support and these things could probably still be detected via performance measurements.

What about non-hardware identifiers?

In addition to not having a way to identify the hardware, apps cannot directly identify the installation of the OS on the hardware. Apps only have a small portion of the OS configuration exposed to them and there is not much for device owners to change which could identify their installation. Apps can detect that they're being run on GrapheneOS via the privacy and security features placing further restrictions on them and hardening them against further exploitation. Apps can identify their own app installation via their app data and can directly (until that's removed) or indirectly identify a profile. Profiles should be used when separate identities are desired. Profiles can be used as temporary ephemeral identities by creating them for a specific need and then deleting them. The rest of this answer only provides more technical details, so you can stop reading here if you only want an overview and actionable advice (i.e. use profiles as identities not inherently tied to each other).

Examples of the global OS configuration available to apps are time zone, network country code and other similar global settings. Per-profile examples are dark mode and language. Similar to extension and browser configuration / state being fingerprinted by web sites, an app could use a combination of these things in an attempt to identify the installation. All of these things vary at runtime and can be changed, but some are fairly unlikely to change in practice after the initial setup of the device such as the ones listed above. GrapheneOS will likely add further restrictions in this area and a couple toggles for certain cases like time zones to use a standard value instead.

Apps can generate their own 128-bit or larger random value and use that as an identifier for the app installation. Apps can create data in their app-specific external storage directory by default without needing permission, and in the legacy storage model before API 29 that data persists after the app is uninstalled, so it can be used to store an ID that persists through the app being uninstalled and reinstalled. However, external storage is under control of the user and the user can delete this data at any time, including after uninstalling the app. In the modern storage model, this data is automatically removed when the app is uninstalled. GrapheneOS includes Seedvault as an OS backup service which must be explicitly enabled, and it has the option to automatically restore app data when an app is reinstalled, so it wouldn't lose track of it being the same profile.

The *ANDROID_ID* string is a 64-bit random number, unique to each combination of profile and app signing key. The 64-bit limitation means it isn't particularly useful due to the possibility of collisions. It's tied to the lifetime of profiles and does not persist through profile deletion or a factory reset. This is comparable to an app targeting the legacy storage model storing a 64-bit random value in the app-specific external storage directory. In the future, GrapheneOS will likely change this to be tied to the lifetime of app installations rather than profiles. An app could still track the identity of the profile through data you give it access to or via data another app chooses to share with them.

The advertising ID is a Google Play services feature not included in the baseline Android API, so it isn't an API included in GrapheneOS. The advertising ID is unique to each profile. It isn't unique to each app signing key like *ANDROID_ID*, but that makes little difference since apps within the same profile can communicate with each other with mutual consent. It's comparable to *ANDROID_ID* but provides an 128-bit value so it provides a strong cryptographic guarantee against collisions, although a device messing with apps could set it to the same value used in another profile. The advertising ID is exposed via the Settings app and can be reset to a new random value, unlike *ANDROID_ID* which remains the same for the lifetime of the profile, but apps can tie it to the previous ID since they can detect that it changed via their own ID in their app data. The advertising ID can also now be disabled (zeroed).

Apps do not have access to user data by default and cannot ever access the data of other apps without those apps going out of the way to share it with them. If apps are granted read access to user data like media or contacts, they could use it to identify the profile. If apps are granted write access to user data, they could tag it to keep track of the profile. Apps previously had little reason to do things like this because they were able to persist data through an uninstall and reinstallation by default. The modern storage model means they need to request access to user data to do this. The existence of *ANDROID_ID* means they don't yet need to bother with that but that will change on GrapheneOS and will likely change for baseline Android too. However, profiles are the only way to provide a strong assurance of separate identities since the application model of the OS is designed to support communication between apps within the same profile, but never between them.

What does GrapheneOS do about cellular tracking, interception and silent SMS?

GrapheneOS always considers networks to be hostile and avoids placing trust in them. It leaves out various carrier apps included in the stock OS granting carriers varying levels of administrative access beyond standard carrier configuration. GrapheneOS also avoids trust in the cellular network in other ways including providing a secure network time update implementation rather than trusting the cellular network for this. Time is sensitive and can be used to bypass security checks depending on certificate / key expiry.

Cellular networks use inherently insecure protocols and have many trusted parties. Even if interception of the connection or some other man-in-the-middle attack along the network is not currently occurring, the network is still untrustworthy and information should not be sent unencrypted.

Authenticated transport encryption such as HTTPS for web sites avoids trusting the cellular network. End-to-end encrypted protocols such as the Signal messaging protocol also avoid trusting the servers. GrapheneOS uses authenticated encryption with modern protocols, forward secrecy and strong cipher configurations for our services. We only recommend apps taking a decent approach in this area.

Legacy calls and texts should be avoided as they're not secure and trust the carrier / network along with having weak security against other parties. Trying to detect some forms of interception rather than dealing with the root of the problem (unencrypted communications / data transfer) would be foolish and doomed to failure.

GrapheneOS does not add gimmicks without a proper threat model and rationale. We won't include flawed heuristics to guess when the cellular network should be trusted. These kinds of features provide a false sense of security and encourage unwarranted trust in cellular protocols and carrier networks as the default. These also trigger false positives causing unnecessary concern and panic. The correct approach is avoiding trusting the network as explained above.

Connecting to your carrier's network inherently depends on you identifying yourself to it and anyone able to obtain administrative access. Activating airplane mode will fully disable the cellular radio transmit and receive capabilities, which will prevent your phone from being reached from the cellular network and stop your carrier (and anyone impersonating them to you) from tracking the device via the cellular radio. The baseband implements other functionality such as Wi-Fi and GPS functionality, but each of these components is separately sandboxed on the baseband and independent of each other. Enabling airplane mode disables the cellular radio, but Wi-Fi can be re-enabled and used without activating the cellular radio again. This allows using the device as a Wi-Fi only device.

The LTE-only mode added by GrapheneOS is solely intended for attack surface reduction. It should not be mistaken as a way to make the cellular network into something that can be trusted.

Receiving a silent SMS is not a good indicator of being targeted by your cell carrier, police or government because *anyone on the cell network can send them* including yourself. Cellular triangulation will happen regardless of whether or not SMS texts are being sent or received by the phone. Even if an SMS did serve a useful purpose for tracking, a silent SMS would be little different than receiving unsolicited spam. In fact, sending spam would be stealthier since it wouldn't trigger alerts for silent SMS but rather would be ignored with the rest of the spam. Regardless, sending texts or other data is not required or particularly useful to track devices connected to a network for an adversary with the appropriate access.

Airplane mode is the only way to avoid the cellular network tracking your device and works correctly on the devices we support.

How private is Wi-Fi?

See the usage guide section on Wi-Fi privacy.

What kind of connections do the OS and bundled apps make by default?

By default, GrapheneOS only makes remote connections to GrapheneOS services and the network provided DNS resolvers. There aren't any analytics/telemetry in GrapheneOS. The only information revealed to the GrapheneOS servers are the generic device model (such as Pixel 7 Pro) and OS version which are necessary for obtaining updates. The default connections provide the OS and apps with updates, set the system clock, check each network connection for internet connectivity, download a global database (does not vary based on location) with predicted satellite locations when using Location and obtain attestation chain signing keys for the hardware keystore needed for the hardware-based attestation feature.

Make sure to read the other connections section below this one too which covers non-default connections triggered by having a certain carrier, having apps installed, etc.

The expected default connections by GrapheneOS (including all base system apps) are the following:

- The GrapheneOS System Updater app fetches update metadata from https://releases.grapheneos.org/*DEVICE*-*CHANNEL* approximately once every six hours when connected to a permitted network for updates.

  Once an update is available, it tries to download https://releases.grapheneos.org/*DEVICE*-incremental-*OLD_VERSION*-*NEW_VERSION*.zip for a delta update, and then falls back to https://releases.grapheneos.org/*DEVICE*-ota_update-*NEW_VERSION*.zip.

  No query / data is sent to the server, so the only information given to it are the variables in these 3 URLs (device, channel, current version) which is necessary to obtain the update.

  Users are in control of which types of networks the Updater app will use and can disable the Updater app in extreme cases. It's strongly recommended to leave it enabled to quickly receive security updates including updates outside the regular monthly schedule.

  The update client avoids trusting the data obtained from the update server via signature verification with downgrade protection. Verified boot provides another layer of signature verification with downgrade protection. GrapheneOS servers do not have access to GrapheneOS signing keys.

  See the usage guide's section on updates for more information.

- The GrapheneOS app repository client (App Store) fetches generic signed update metadata and signed package updates (APKs) from https://apps.grapheneos.org/ (a separate name for the same servers as https://releases.grapheneos.org). It provides out-of-band updates to certain apps bundled with the OS and other apps available in our repository.

- Vanadium, our browser and WebView implementation, uses `update.vanadium.app` to check for updates to components providing revoked certificates and other data. It downloads the components from `dl.vanadium.app`.

- Opening the Info app will fetch the latest GrapheneOS release notes from https://grapheneos.org/releases.atom.

- An HTTPS connection is made to https://time.grapheneos.org/generate_204 to update the time with a millisecond precision X-Time header. As part of future support for using other services, it falls back to the standard Date header with second precision.

  This is a full replacement for Android's standard network time update implementation, which uses unauthenticated SNTP (Simple Network Time Protocol) with fallback to the cellular network when it's not available (GNSS can also be used as a time source but is disabled by default, and OEMs can choose the priority order). Network time updates are security sensitive since certificate validation depends on having an accurate time, but the standard NTP / SNTP protocols used across most OSes have no authentication. Our servers obtain the time from 6 independent NTP servers with NTS for authentication where at least 3 servers need to agree on the time for it to be updated.

  We plan to offer a toggle to use the standard functionality instead of HTTPS-based time updates in order to blend in with other devices.

Network time can be disabled with the toggle at Settings > System > Date & time > Set time automatically. Unlike AOSP or the stock OS on the supported devices, GrapheneOS stops making network time connections when using network time is disabled rather than just not setting the clock based on it. The time zone is still obtained directly via the time zone provided by the mobile network (NITZ) when available which you can also disable by the Set time zone automatically toggle.

- Connectivity checks designed to mimic a web browser user agent are performed by using HTTP and HTTPS to fetch standard URLs generating an HTTP 204 status code. This is used to detect when internet connectivity is lost on a network, which triggers fallback to other available networks if possible. These checks are designed to detect and handle captive portals which substitute the expected empty 204 response with their own web page.

  The connectivity checks are done by performing an empty GET request to a server returning an empty response with a 204 No Content response code. The request uses a standard, frozen value for the user agent matching the same value used by billions of other Android devices:

```
Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/60.0.3112.32 Safari/537.36
```

  No query / data is sent to the servers and the response is unused beyond checking the response code.

  Connectivity checks are performed for each network connection and for VPN connections on top of those. This allows the OS to choose the right underlying network for a VPN and to handle many types of captive portals without the user turning off their VPN.

  You can change the connectivity check URLs via the Settings > Network & internet > Internet connectivity check setting. At the moment, it can be toggled between the GrapheneOS server (default), the Standard (Google) server used by billions of other Android devices or Off.

  By default, the GrapheneOS server is used via the following URLs:

  - HTTPS: https://connectivitycheck.grapheneos.network/generate_204
  - HTTP: http://connectivitycheck.grapheneos.network/generate_204
  - HTTP fallback: http://grapheneos.online/gen_204
  - HTTP other fallback: http://grapheneos.online/generate_204

  Changing this to Standard (Google) server will use the same URLs used by AOSP and the stock OS along with the vast majority of other devices, blending in with billions of other Android devices both with and without Play services:

  - HTTPS: https://www.google.com/generate_204
  - HTTP: http://connectivitycheck.gstatic.com/generate_204
  - HTTP fallback: http://www.google.com/gen_204
  - HTTP other fallback: http://play.googleapis.com/generate_204

GrapheneOS also adds the ability to fully turn Off the connectivity checks. This results in the OS no longer handling captive portals itself, not falling back to other networks when some don't have internet access and not being able to delay scheduled jobs depending on internet access until it becomes available.

- HTTPS connections are made to fetch PSDS information to assist with satellite based location. These are static files and are downloaded automatically to improve location resolution speed and accuracy. No query or data is sent to these servers. These contain orbits and statuses of satellites, Earth environmental data and time adjustment information.

  Pixel 8a, Pixel 9, Pixel 9 Pro, Pixel 9 Pro XL and Pixel 9 Pro Fold use a Samsung GNSS chip. Almanacs are downloaded from https://samsung.psds.grapheneos.org/p4/42F3 which is a cache of Samsung's data. Alternatively, the standard servers can be enabled in the Settings app which will use https://1.ssiloc.com/p4/42F3.

  Pixel 6, Pixel 6 Pro, Pixel 6a, Pixel 7, Pixel 7 Pro, Pixel 7a, Pixel Fold, Pixel 8 and Pixel 8 Pro use a Broadcom GNSS chip. Almanacs are downloaded from https://broadcom.psds.grapheneos.org/lto2.dat, https://broadcom.psds.grapheneos.org/rto.dat and https://broadcom.psds.grapheneos.org/rtistatus.dat which are a cache for Broadcom's data available at https://gllto.glpals.com/7day/v5/latest/lto2.dat, https://gllto.glpals.com/rto/v1/latest/rto.dat and https://gllto.glpals.com/rtistatus4.dat. Alternatively, the standard servers can be enabled in the Settings app which are https://agnss.goog/lto2.dat, https://agnss.goog/rto.dat and https://agnss.goog/rtistatus.dat providing a similar cache of Broadcom's data currently (as of October 2022) hosted on GCP (Google Cloud Platform).

  Pixel 4, Pixel 4 XL, Pixel 4a, Pixel 4a (5G), Pixel 5 and Pixel 5a use a Qualcomm baseband providing cellular, Wi-Fi, Bluetooth and GNSS in separate sandboxes), almanacs are downloaded from https://qualcomm.psds.grapheneos.org/xtra3Mgrbeji.bin which is a cache of Qualcomm's data. Alternatively, the standard servers can be enabled in the Settings app which will use https://path1.xtracloud.net/xtra3Mgrbeji.bin, https://path2.xtracloud.net/xtra3Mgrbeji.bin and https://path3.xtracloud.net/xtra3Mgrbeji.bin. GrapheneOS improves the privacy of Qualcomm PSDS (XTRA) by removing the User-Agent header normally containing an SoC serial number (unique hardware identifier), random ID and information on the phone including manufacturer, brand and model. We also always fetch the most complete XTRA database variant (xtra3Mgrbeji.bin) instead of model/carrier/region dependent variants to avoid leaking a small amount of information based on the database variant.

  Qualcomm Snapdragon SoC devices also fetch time via NTP for xtra-daemon instead of using potentially incorrect OS time. We use time.grapheneos.org when using the default GrapheneOS PSDS servers or the standard time.xtracloud.net when using Qualcomm's servers. Stock Pixel OS uses time.google.com but we follow Qualcomm's standard settings to match other devices and to avoid the incompatible leap second handling. These connections all go through the Owner VPN so it isn't a real world fingerprinting issue.

- Android devices launched with Android 8 or later provide support for hardware-based attestation as part of the hardware keystore API. Secure devices like Pixels provide both the traditional Trusted Execution Environment (TrustZone) keystore and StrongBox keystore based on a secure element, each providing attestation support. The hardware-based attestation feature is a standard part of the Android Open Source Project and are used to implement our Auditor app among other things.

  Initially, attestation signing keys were required to be batch keys provisioned to at least 100k devices to avoid them being used as unique identifiers. Unique attestation signing keys are an optional feature only available to privileged system components. Recent devices have replaced the batch and unique key system with remotely provisioned signing keys. The device obtains encrypted keys from a service to be decrypted by batch or unique keys inside the TEE and optional secure element. The new system improves privacy and security by using separate attestation signing keys for each app instead of needing to balance privacy and security by sharing the same attestation signing keys across a large batch of devices.

  GrapheneOS uses https://remoteprovisioning.grapheneos.org/ by default which is a private reverse proxy to the https://remoteprovisioning.googleapis.com/ service. The service splits up the implementation of provisioning to preserve privacy, and our reverse proxy adds to that since it's unable to decrypt the provisioned keys.

  A setting is added at Settings > Network & internet > Attestation key provisioning for switching to directly using the Google service if you prefer.

  A future device built to run GrapheneOS as the stock OS would be able to have a GrapheneOS attestation root and GrapheneOS attestation key provisioning service rather than a GrapheneOS proxy. A device built to run another OS without Google certification would need their own service and we'd need to support proxying to that service too.

- A test query is done via DNS-over-TLS in the automatic and manually enabled modes to detect if DNS-over-TLS is available. It won't happen when DNS-over-TLS is disabled. For the automatic mode, it uses this to determine if it should be using it and for the manual mode it uses it to report an error. This DNS query is not used to make a connection to the resulting resolved IP.

  GrapheneOS queries the DNS resolver for `randomstring-dnsotls-ds.dnscheck.grapheneos.org` by default but switches to using the standard `randomstring-dnsotls-ds.metric.gstatic.com` when the HTTP(S) connectivity check mode is set to Standard (Google) instead of the default GrapheneOS mode or Disabled mode to avoid identifying itself as GrapheneOS to the DNS resolver. The DNS-over-TLS test query will still happen with HTTP(S) connectivity checks disabled but DNS-over-TLS can be disabled by disabling Private DNS.

  The random string is used to bypass DNS caching to make sure the DNS resolver works. It's generated with a cryptographically secure random number generator (CSPRNG) for each request and therefore can't leak any identifying info.

- DNS resolution for other connections involving connections to the network / user provided DNS resolvers

Other Vanadium browser connections are initiated by the user such as the search engine (defaults to DuckDuckGo), websites and retrieving favicons for your bookmarks and the frequent sites shown on the home page. We enable connection and cache partitioning to keep connections separate between different sites. There are a few parts of state partitioning still being implemented including cookie partitioning which is currently in the testing phase but causes too many compatibility issues to fully deploy it by default without any exceptions like Brave's heuristics to disable it for cross-site login flows, etc.

Unlike Chrome/Chromium, Vanadium WebView does not make connections beyond those initiated by the app using it so there are no default connections from apps using the WebView.

Which additional connections can the OS make with a non-default configuration?

The previous section is an exhaustive list of all the default connections made by a fresh GrapheneOS installation. Using a carrier, installing apps and changing configuration can enable additional connections. This section aims to list the cases which are not completely obvious to users. For example, if you explicitly configure a Private DNS server, we don't need to explain here that the OS will be connecting to that server.

Apps can list domains where they want to handle URLs instead of them being handled by the browser. Domains officially associated with an app can add the required metadata authorizing the app to automatically handle URLs which the OS will fetch via HTTPS after installing the app to confirm if the app claims to be authorized. See our usage guide section on app link verification for more details such as how to block these connections. The apps bundled with GrapheneOS don't require this and we could hard-wire domains as verified if they did and we wanted to avoid more default connections.

GrapheneOS uses https://widevineprovisioning.grapheneos.org/certificateprovisioning/v1/devicecertificates/create by default which is a private reverse proxy to https://www.googleapis.com/certificateprovisioning/v1/devicecertificates/create as part of Widevine provisioning. This is another form of key provisioning for per-app keys that are used when playing DRM protected media. DRM support is enabled in the OS by default but we don't include any apps using it by default, since it's disabled in Vanadium. A setting is added at Settings > Network & Internet > Widevine provisioning for switching to directly using the Google service if you prefer.

Most other connections made by the OS itself are made based on your chosen carrier. The OS has a database of APN and other carrier configuration settings which determines how this works by default. Normally, carriers can force their configuration choices on users by making APNs read-only and disabling various configuration options. GrapheneOS ignores this and always allows configuring APNs, APN types, changing preferred network mode, toggling off 2G and using tethering regardless of what the carrier wants. We leave the defaults chosen by the carriers alone. For example, if you want tethering traffic treated normally, you can remove the dun APN type from your APN configuration.

When you have both a cellular connection and Location enabled, control plane and/or user plane (SUPL) A-GNSS is used in addition to PSDS to greatly reduce the time needed for GNSS to obtain an initial location lock. These obtain coarse location info from a server based on nearby cell towers. Control plane A-GNSS is provided by the cellular connection itself and therefore doesn't have any real privacy implications while SUPL connects to a server often not provided by the carrier. Most A-GNSS services only accelerate obtaining a satellite-based location and won't provide an estimate on their own. The carrier can choose a SUPL server as part of their carrier configuration but most leave it at the default of supl.google.com. By default, GrapheneOS overrides the carrier/fallback SUPL server and uses the supl.grapheneos.org proxy. GrapheneOS adds a toggle for configuring SUPL in Settings > Location where you can choose between the default GrapheneOS proxy supl.grapheneos.org, the Standard server (carrier/fallback) or turning it Off completely. GrapheneOS also disables sending IMSI and phone number as part of SUPL. Pixels with a Qualcomm baseband use it to provide both cellular and GNSS including both control plane and user plane A-GNSS being implemented inside the baseband. For Qualcomm baseband devices, SUPL is only enabled if the APN configuration for the carrier includes `supl` as an APN type. Pixels with a Samsung baseband have a separate Broadcom or Samsung GNSS chip without integration between them so SUPL is done by the OS with regular networking (can use Wi-Fi and VPN) and SUPL is used regardless of the carrier's APN type configuration.

MMS, RCS, SMS over LTE, VVM (Visual Voicemail), VoLTE (carrier-based calls on 4G and higher), VoNR (5G) and VoWi-Fi are largely implemented by the OS via TCP/IP rather than by the cellular layer itself. This means there will be connections by the OS to carrier servers instead of being handled by cellular. GrapheneOS modifies carrier configuration so that toggles for disabling VoLTE, VoNR and VoWi-Fi are always available. We plan to provide more toggles to control these things in the future beyond making all the standard toggles available.

What is the privacy policy for GrapheneOS services?

GrapheneOS services follow the EFF's privacy-friendly Do Not Track (DNT) policy for all users of our publicly available services, not just those opting out of tracking via Do Not Track. Our policy is the same with "DNT User" redefined as "user" to cover any user. This serves as a standard privacy policy across all of our public services:

- attestation.app
- grapheneos.network
- grapheneos.online
- grapheneos.org
- grapheneos.social
- apps.grapheneos.org
- broadcom.psds.grapheneos.org
- samsung.psds.grapheneos.org
- qualcomm.psds.grapheneos.org
- discuss.grapheneos.org
- element.grapheneos.org
- mail.grapheneos.org
- matrix.grapheneos.org

- releases.grapheneos.org
- remoteprovisioning.grapheneos.org
- widevineprovisioning.grapheneos.org
- supl.grapheneos.org
- time.grapheneos.org
- gs-loc.apple.grapheneos.org
- update.vanadium.app
- dl.vanadium.app

Our implementation of the policy primarily consists of making sure our servers only retain logs for at most 10 days with a lower limit or no persistent logs for certain services. In practice, we follow much stricter privacy guidelines than the rules laid out in the EFF policy. However, we don't want to define our own complex, ad-hoc privacy policy rather than reusing a sensible one with serious thought put into it by experts.

Our mail server (mail.grapheneos.org), Matrix server (matrix.grapheneos.org), Element instance (element.grapheneos.org) and Mastodon server (grapheneos.social) only provide accounts for GrapheneOS project members so most functionality is outside the scope of what's relevant to a public privacy policy.

Which DNS servers are used by default?

The OS uses the network-provided DNS servers by default. Typically, dynamic IP configuration is used to auto-configure the client on the network. IPv4 DNS servers are obtained via DHCP and IPv6 DNS servers are obtained via RDNSS. For a static IP configuration, the DNS servers are manually configured as part of the static configuration.

A VPN provides a network layered on top of the underlying networks and the OS uses the VPN-provided DNS servers for everything beyond resolving the IP address of the VPN and performing network connectivity checks on each of the underlying networks in addition to the VPN itself.

Using the network-provided DNS servers is the best way to blend in with other users. Network and web sites can fingerprint and track users based on a non-default DNS configuration. Our recommendation for general purpose usage is to use the network-provided DNS servers.

In some broken or unusual network environments, the network could fail to provide DNS servers as part of dynamic IP configuration. The OS has high availability fallback DNS servers to handle this case. A network can fail to provide DNS servers in order to fingerprint clients based on what they use as the fallback so it's important for it to be consistent across each install. GrapheneOS replaces Google Public DNS with Cloudflare DNS for the fallback DNS servers due to the superior privacy policy and widespread usage including as the fallback DNS servers in other Android-based operating systems. We're considering hosting our own servers and offering a toggle for using the standard (Google) servers to blend in with other devices similarly to how we handle the internet connectivity checks.

How do I use a custom DNS server?

It isn't possible to directly override the DNS servers provided by the network via DHCP. Instead, use the Private DNS feature in Settings > Network & internet > Private DNS to set the hostname of a DNS-over-TLS server. It needs to have a valid certificate such as a free certificate from Let's Encrypt. The OS will look up the Private DNS hostname via the network provided DNS servers and will then force all other DNS requests through the Private DNS server. Unlike an option to override the network-provided DNS servers, this prevents the network from monitoring or tampering with DNS requests/responses.

As an example, set the hostname to `one.one.one.one` for Cloudflare DNS. There are various other mainstream DNS-over-TLS options available including Quad9, Google and AdGuard.

Configuring a static IP address for a network requires entering DNS servers manually, but you should still use the Private DNS feature with it, and you shouldn't misuse the static IP address option just to override the DNS servers.

VPN service apps can also provide their own DNS implementation and/or servers, including an alternate implementation of encrypted DNS. Private DNS takes precedence over VPN-provided DNS, since it's just the network-provided DNS.

Apps and web sites can detect the configured DNS servers by generating random subdomains resolved by querying their authoritative DNS server. This can be used as part of fingerprinting users. If you're using a VPN, you should consider using the standard DNS service provided by the VPN service to avoid standing out from other users.

Why does Private DNS not accept IP addresses?

By default, in the automatic mode, the Private DNS feature provides opportunistic encryption by using DNS-over-TLS when supported by the DNS server IP addresses provided by the network (DHCP) or the static IP configuration. Opportunistic encryption provides protection against a passive listener, not an active attacker, since they can force falling back to unencrypted DNS by blocking DNS-over-TLS. In the automatic mode, certificate validation is not enforced, as it would provide no additional security and would reduce the availability of opportunistic encryption.

When Private DNS is explicitly enabled, it uses authenticated encryption without a fallback. The authentication is performed based on the hostname of the server, so it isn't possible to provide an IP address. The OS will look up the hostname of the Private DNS server via unencrypted DNS and then force all other DNS lookups via DNS-over-TLS with the identity of the server authenticated as part of providing authenticated encryption.

Does DNS-over-TLS (Private DNS) protect other connections?

No, it only provides privacy for DNS resolution. Even authenticating DNS results with DNSSEC does not protect other connections, unless the DNS records are part of the system used to provide authenticated encryption, and DNS-over-TLS is not a substitute for DNSSEC. If connections have authenticated encryption, they're secure even if DNS resolution is hijacked by an attacker. If connections do not have authenticated encryption, an attacker can listen in and tamper with them without hijacking DNS. There are other ways to perform a MITM attack than DNS hijacking and internet routing is fundamentally insecure. DNS-over-TLS may make a MITM harder for some attackers, but don't count on it at all.

Does DNS-over-TLS (Private DNS) hide which sites are visited, etc.?

Private DNS only encrypts DNS, and an adversary monitoring connections can still see the IP address at the other end of those connections. Many domains resolve to ambiguous IP addresses, so encrypted DNS is part of what's required to take away a lot of the information leaked to adversaries. However, TLS currently leaks domains via SNI, so encrypted DNS is not yet accomplishing much. It's a forward looking feature that will become more useful in the future. Using it is recommended, but it's not an alternative to using Tor or a VPN.

What kind of VPN and Tor support is available?

VPNs can be configured under Settings > Network & internet > VPN. Support for the following protocols is included: IKEv2/IPSec MSCHAPv2, IKEv2/IPSec PSK and IKEv2/IPSec RSA. Apps can also provide userspace VPN implementations. The only apps we can recommend are the official WireGuard app and the official Mullvad app. Mullvad's app is tested on GrapheneOS with hardware memory tagging, unlike the official WireGuard app which has invalid memory accesses detected by memory tagging if it's enabled. We recommend using one of these VPN apps instead of the built-in IPSec VPN support.

VPN configurations created with the built-in support can be set as the always-on VPN in the configuration panel. This will keep the VPN running, reconnecting as necessary and will force all connections through them. An app providing a VPN service can also be set as the always-on VPN via the entry in the Settings page. For app-based VPN implementations, there's also an additional "Block connections without VPN" toggle which is needed to prevent leaks when the app's VPN service isn't running.

If you're using a VPN, we recommended against having a Private DNS server configured. If you want to filter traffic while using a VPN, use a VPN service app able to do both such as RethinkDNS. Private DNS also interacts strangely with multiple profiles since each profile has their own VPN configuration but Private DNS is global. We strongly recommend fully disabling Private DNS when using a VPN on any secondary profile until it's overhauled.

Can apps monitor network connections or statistics?

Apps cannot monitor network connections unless they're made into the active VPN service by the user. Apps cannot normally access network stats and cannot directly request access to them. However, app-based stats can be explicitly granted by users as part of access to app usage stats in Settings > Apps > Special app access > Usage access.

This was previously part of the GrapheneOS privacy improvements, but became a standard Android feature with Android 10.

Does GrapheneOS provide a firewall?

Yes, GrapheneOS inherits the deeply integrated firewall from the Android Open Source Project, which is used to implement portions of the security model and various other features. The GrapheneOS project historically made various improvements to the firewall but over time most of these changes have been integrated upstream or became irrelevant.

GrapheneOS adds a user-facing Network permission toggle providing a robust way to deny both direct and indirect network access to applications. It builds upon the standard non-user-facing INTERNET permission, so it's already fully adopted by the app ecosystem. Revoking the permission denies indirect access via OS components and apps enforcing the INTERNET permission, such as DownloadManager. Direct access is denied by blocking low-level network socket access. A packet-based firewall would only block direct access so our approach is much more complete. Additionally, GrapheneOS pretends that the Network is down for most APIs when the Network permission is disabled. For example, it won't run scheduled jobs depending internet availability and most APIs for checking the state of the network will report it as down and internet access as unavailable. This means apps won't try to keep trying to access the internet and draining battery because they'll treat it the way they do when internet access is genuinely unavailable.

How can I set up system-wide ad-blocking?

The recommended approach to system-wide ad-blocking is setting up domain-based ad-blocking as part of DNS resolution. You can do this by choosing a Private DNS (DNS-over-TLS) server with support for blocking ad domains. As an example, AdGuard DNS can be used by setting `dns.adguard-dns.com` as the Private DNS domain. This feature used to be included by the project many years ago, but it needs to be reimplemented, and it's a low priority feature depending on contributors stepping up to work on it.

Apps and web sites can detect that ad-blocking is being used and can determine what's being blocked. This can be used as part of fingerprinting users. Using a widely used service like AdGuard with a standard block list is much less of an issue than a custom set of subscriptions / rules, but it still stands out compared to the default of not doing it.

Are ad-blocking apps supported?

Content filtering apps are fully compatible with GrapheneOS, but they have serious drawbacks and using apps doing more than DNS-based filtering are not recommended. These apps use the VPN service feature to route traffic through themselves to perform filtering.

The approach of intercepting traffic is inherently incompatible with encryption from the client to the server. The AdGuard app works around encryption by supporting optional HTTPS interception by having the user trust a local certificate authority, which is a security risk and weakens HTTPS security even if their implementation is flawless (which they openly acknowledge in their documentation, although it understates the risks). It also can't intercept connections using certificate pinning, with the exception of browsers which go out of the way to allow overriding pinning with locally added certificate authorities. Many of these apps only provide domain-based filtering, unlike the deeper filtering by AdGuard, but they're still impacted by encryption due to Private DNS (DNS-over-TLS) and require disabling the feature. They could provide their own DNS-over-TLS resolver to avoid losing the feature, but few of the developers care enough to do that.

Using the VPN service to provide something other than a VPN also means that these apps need to provide an actual VPN implementation or a way to forward to apps providing one, and very few have bothered to implement this.

RethinkDNS combines local filtering via DNS with the ability to directly use a WireGuard VPN without another app. It also has other features such as connection monitoring. This is a much better approach than most of the apps in this space which force choosing between them and a VPN, recommend problematic TLS interception (AdGuard), etc.

Is the baseband isolated?

Yes, the baseband is isolated on all of the officially supported devices. Memory access is partitioned by the IOMMU and limited to internal memory and memory shared by the driver implementations. The baseband on the officially supported devices with a Qualcomm SoC implements Wi-Fi and Bluetooth as internal sandboxed processes rather than having a separate baseband for those like earlier devices.

Earlier generation devices we used to support prior to Pixels had Wi-Fi + Bluetooth implemented on a separate SoC. This was not properly contained by the stock OS and we put substantial work into addressing that problem. However, that work has been obsoleted now that Wi-Fi and Bluetooth are provided by the SoC on the officially supported devices.

A component being on a separate chip is orthogonal to whether it's isolated. In order to be isolated, the drivers need to treat it as untrusted. If it has DMA access, that needs to be contained via IOMMU and the driver needs to treat the shared memory as untrusted, as it would do with data received another way. There's a lot of attack surface between the baseband and the kernel/userspace software stack connected to it. OS security is very relevant to containing hardware components including the radios and the vast majority of the attack surface is in software. The OS relies upon the hardware and firmware to be able to contain components but ends up being primarily responsible for it due to control over the configuration of shared memory and the complexity of the interface and the OS side implementation.

The mobile Atheros Wi-Fi driver/firmware is primarily a SoftMAC implementation with the vast majority of the complexity in the driver rather than the firmware. The fully functional driver is massive and the firmware is quite small. Unfortunately, since the Linux kernel is monolithic and has no internal security boundaries, the attack surface is problematic and a HardMAC implementation with most complexity in the isolated firmware could be better than the status quo. An isolated driver would be ideal.

Why am I seeing a message about my bootloader not being locked when setting up the device?

If you are seeing this warning when setting up your device after GrapheneOS has been installed, it means that not all of the installation steps have been completed. This can be remediated by finishing the installation process and locking the device's bootloader.

To do this, follow these steps:

1. Navigate to the bootloader locking section of the web install (recommended) or CLI install guide on the device you'll be using to lock the bootloader.
2. If you are at the welcome screen on your GrapheneOS device, tap "Next".
3. On the resulting screen, tap "Reboot to bootloader".
4. Connect your GrapheneOS device to the device where you have the install page open.
5. Follow the steps on the install page to lock the bootloader.
6. Once locked, you can start setting up your GrapheneOS device, and the warning should no longer be present.

It is important to note that this is something that should be done before placing any data onto the device, as until this step is completed, the device is considered to be insecure. Please note that locking the bootloader wipes all user data on the device. GrapheneOS doesn't provide any support to users running GrapheneOS with an unlocked bootloader, as this is considered to be an incomplete installation.

Day to day use

How do I keep the OS updated?

GrapheneOS has entirely automatic background updates. More details are available in the the usage guide's updates section.

How do I update without connecting the device to the internet?

Updates can be sideloaded via recovery.

Do notifications work properly on GrapheneOS?

Yes, notifications work properly on GrapheneOS. Portable apps avoiding a hard dependency on Google Play services for their functionality have fully working notifications on GrapheneOS. Apps that are not fully portable across Android implementations often lack support for background notifications due to only bothering to implement support for it via Google Play services.

Most apps that are able to run without Google Play services will have working notifications when they're in the foreground. Unfortunately, many apps don't implement a service to continue receiving events from their server in the background. On the stock OS, they rely on receiving events through Google servers via Firebase Cloud Messaging (FCM) in the background and sometimes even in the foreground, although it doesn't have good reliability/latency.

Polling is the traditional pull-based approach of checking for new events at an interval. This is badly suited to mobile devices for anything more than very infrequent checks. Apps using infrequent polling are supposed to use the JobScheduler service. A minority of apps may only know how to use Firebase WorkManager or the legacy Firebase JobDispatcher. Most apps know how to use JobScheduler rather than depending on Google Play services for the Firebase services. Typical examples of apps using this approach are a feed reader for RSS/Atom feeds or an email client providing notifications of new emails for a server without IMAP IDLE push support.

Push messaging is the modern push-based model of receiving events from the server as they occur by keeping open a connection to it. Push messaging still uses occasional polling to keep the connection from being killed by a network using a stateful firewall or some form of NAT. IPv4 mobile networks use large scale NAT (CGNAT) to work around IPv4 addresses running out. The occasional polling will also detect a silently dropped connection. An efficient push implementation will figure out that it's on a reliable network and throttle the polling to be very infrequent.

In order to properly implement either push messaging or frequent polling themselves, an app needs to run a foreground service. This is displayed as a persistent notification. It will normally be marked as a silent notification with the lowest possible default importance for a foreground service (IMPORTANCE_LOW). It can be reduced to the lowest importance (IMPORTANCE_MIN) by the user if they set the notification channel to be collapsed which will collapse it in the notification tray and it won't show up as an icon in the status bar or on the lockscreen anymore. Users can also disable the notification and the foreground service will continue working. A battery optimization exception is also needed for the app to bypass device idle states and run while the device is idle. If you can tolerate delays while the device is idle, then the battery optimization exception isn't mandatory.

FairEmail and Signal are examples of apps using the proper approach of a foreground service combined with an optional battery optimization exception. Signal doesn't have an optimized implementation throttling the polling used to keep the connection alive, but it does work well. Signal always uses their own push implementation in the foreground, but switches to FCM in the background when it's available. FairEmail uses the IMAP IDLE push feature provided by email servers. Most email servers don't provide FCM-based push in the first place, and the only way for an email app to provide push via FCM would be to give the user's credentials to their own server to act as a middleman.

How do I transfer files to another device?

Files can be transferred to another device using an external drive, USB file transfer via MTP / PTP or an app-based mechanism.

The only difference on GrapheneOS is that by default the USB-C port and pogo pins block new connections as soon as the device is locked by default and disable USB data when any existing connections end. The default behavior won't interfere with this use case at all. If you've changed it to a stricter value such as completely disallowing USB-C data, it can be changed back to the default "Charging-only when locked" value in Settings > Security > Exploit protection > USB-C port.

To use an external drive, plug it into the phone and use the system file manager to copy files to and from it.

Transferring files to an attached computer is done with MTP / PTP. Users on a Mac computer will need to install OpenMTP or another MTP app to be able to transfer files between macOS and Android. After plugging in the phone to the computer, there will be a notification showing the current USB mode with charging as the default. Pressing the notification acts as a shortcut to Settings > Connected devices > USB. You can enable File Transfer (MTP) or PTP with this menu. It will provide read/write access to the entire profile home directory, i.e. the top-level directory named after the device in the system file manager which does not include internal app data. Due to needing to trust the computer with coarse-grained access, we recommend transferring files with a flash drive or by sending the files to yourself via an end-to-end encrypted messaging app like Element (Matrix).

Will GrapheneOS include support for Google services?

Like the Android Open Source Project, GrapheneOS doesn't include Google apps and services. They won't ever be bundled with the OS. GrapheneOS includes a compatibility layer for sandboxed Play services to make user installed Play services apps able to run as fully sandboxed, unprivileged apps. This is documented as part of the usage guide. Many apps work perfectly without Play services and many others only depend on it for a subset of their functionality. Users can choose to install Play services in specific profile(s) to control which apps can use it.

AOSP APIs not tied to Google but that are typically provided via Play services will continue to be implemented using open source providers like the Seedvault backup app. Text-to-speech, speech-to-text, geocoding, accessibility services, etc. are examples of other open Android APIs where we need to develop/bundle an implementation based on existing open source projects. GrapheneOS is not going to be implementing these via a Google service compatibility layer because these APIs are in no way inherently tied to Google services.

Ideally, Google themselves would support installing the official Play services as regular apps rather than taking the monopolistic approach of forcing it to be bundled into the OS in a deeply integrated way with special privileged permissions and capabilities unavailable to other service providers competing with them. Until that happens, if ever, GrapheneOS can continue teaching it to function that way to the extent possible without any special privileges.

What features does GrapheneOS implement?

See the features page.

Does GrapheneOS provide Factory Reset Protection?

No, since this is strictly a theft deterrence feature, not a security feature, and the standard implementation depends on having the device tied to an account on an online service. The only advantage would be encouraging thieves to return a stolen device for a potential reward after realizing that it has no value beyond scrapping it for parts.

Google's Factory Reset Protection ties devices to a Google account using a tiny, special region of persistent state not wiped by a factory reset. It prevents a thief from wiping the device to a fresh state for resale without being stuck at a screen for authenticating with the Google account persisted on the device after wiping. Google's approach works well because if users forget their Google password, there are account recovery methods available to avoid a bricked phone.

It would be possible to make an implementation not reliant upon an online service where the user has the option to enable Factory Reset Protection and is given a seed phrase required to use the device after wiping data from recovery. However, since this has no security value and the ability to deter theft is questionable, implementing this is an extremely low priority. Users will end up with a bricked phone if they lose the seed phrase and need to wipe the phone after forgetting their passphrase or something else causing them to need to wipe such as breaking the OS via the Android Debug Bridge shell. Bricked phones would be a far bigger problem than any theft deterrence this could provide. This approach may be implemented by GrapheneOS in some form in the future but it's a low priority and we don't want to cause people to brick their phones. We won't be able to offer any help if people brick their phones with this.

Providing the option to disable wiping from recovery would be simpler, but would be incompatible with features designed to wipe data automatically in certain cases. It would also result in far more bricked phones than the seed phrase approach describe above since setting a new lock method and forgetting it which is a relatively common occurrence would mean a bricked phone. This will not be implemented by GrapheneOS since it isn't a good approach and it conflicts with other planned features.

Why aren't my favorite apps bundled with GrapheneOS?

There are drawbacks to bundling apps into the OS and few advantages in most cases. Rather than GrapheneOS bundling a bunch of apps, it makes far more sense for users to install their preferred apps from the developers and various app stores. GrapheneOS also has a first party app update system for a first party repository with higher robustness and security than the existing options. Rather than bundling apps, it could just offer recommendations as part of an initial setup wizard. Users have unique needs and preferences and there has to be a very compelling reason to bundle additional apps with the OS. For example, it's useful to have the Auditor app available before connecting to the internet (see the installation guide documentation on this).

Bundling additional apps with the OS can increase attack surface, unless users go out of the way to disable apps they aren't using. Bundling an app into the base OS is also painful to reverse, since removing the app without implementing a migration mechanism will lose user data stored in the app. Some users are also going to take issue with the choices made by the project or will want to make suggestions for bundling more apps, and having this as a regular topic of discussion and debate is unproductive and distracts from the real work of the project. Each bundled app also increases the size of the base OS, and shipping the app updates as part of the OS updates results in more overall bandwidth usage. It would be possible to ship only out-of-band app updates to avoid wasted bandwidth for apps users have disabled, but then the apps would be temporarily out-of-date and vulnerable to patched security issues after a factory reset or the user re-enabling them. If the updates aren't going to be shipped with the OS, it really makes no sense to bundle them.

GrapheneOS is focused on making meaningful improvements to privacy and security, and bundling assorted apps into the OS is not only usually outside of that focus but often counter to it.

In some cases, licensing is also an issue. GrapheneOS is permissively licensed and is usable for building devices with an immutable root of trust. GPLv3 is deliberately incompatible with these kinds of locked down devices, unlike GPLv2 code such as the Linux kernel. This means GrapheneOS can't include GPLv3 code without forbidding use cases we want to support. GPLv3 is no problem for our own usage, but we don't want to forbid using GrapheneOS as a replacement for the Android Open Source Project in locked down devices.

What is the roadmap for GrapheneOS?

To get an idea of the near term roadmap, check out the issue trackers. The vast majority of the issues filed in the trackers are planned enhancements, with care taken to make sure all of the issues open in the tracker are concrete and actionable.

In the long term, GrapheneOS aims to move beyond a hardened fork of the Android Open Source Project. Achieving the goals requires moving away from relying on the Linux kernel as the core of the OS and foundation of the security model. It needs to move towards a microkernel-based model with a Linux compatibility layer, with many stepping stones leading towards that goal including adopting virtualization-based isolation.

The initial phase for the long-term roadmap of moving away from the current foundation will be to deploy and integrate pKVM and CrosVM, reinforcing existing security boundaries. The Android Open Source Project has been making significant progress on this front, so our next milestone is to securely deploy Android apps using this virtualization setup. In the longer term, Linux inside the sandboxes can be replaced with a compatibility layer like gVisor, which would need to be given a new backend alongside the existing KVM backend. Over the longer term, i.e. many years from now, Linux and the usage of virtualization can fade away completely. The anticipation is that many other projects are going to be interested in this kind of migration, so it's not going to be solely a GrapheneOS project; this is demonstrated by the current existence and development of gVisor and pKVM, as well as various other projects working on virtualization deployments for mobile. Having a hypervisor with verified boot still intact will also provide a way to achieve some of the goals based on extensions to Trusted Execution Environment (TEE) functionality even without having GrapheneOS hardware.

Hardware and firmware security are core parts of the project, but it's currently limited to research and submitting suggestions and bug reports upstream. In the long term, the project will need to move into the hardware space.

How do I install GrapheneOS?

GrapheneOS has two officially supported installation methods. You can either use the WebUSB-based installer recommended for most users or the command-line installation guide aimed at more technical users.

We strongly recommend using one of the official installation methods. Third party installation guides tend to be out-of-date and often contain misguided advice and errors. If you have trouble with the installation process, ask for help from the GrapheneOS chat room.

The command-line approach offers a way to install GrapheneOS without trusting our server infrastructure. This requires being on an OS with proper fastboot and OpenSSH packages along with understanding the process enough to avoid blindly trusting the instructions from our site. For most users, the web-based installation approach is no less secure and avoids needing any software beyond a browser with WebUSB support.

Can I buy a device with GrapheneOS preinstalled?

There are various companies selling devices with GrapheneOS, something which is permitted by the project's licensing provided that they make it clear that they're not GrapheneOS or officially associated with the project.

We are currently not affiliated with or endorse any company selling devices with GrapheneOS. Our official recommendation is to buy a supported device and install GrapheneOS yourself.

Our official web installer is easy to use, and our community is happy to help with the process should that be needed. On top of that, doing it yourself will also likely be a lot cheaper, as you're avoiding having to pay the premium of having someone else install GrapheneOS for you.

If you decide to buy a device with GrapheneOS installed regardless, here are the things you should pay attention to before doing so, as well as the things you should do after receiving the device:

1. Make sure that you're not buying an end-of-life or near end-of-life device. You should consult our recommended devices section and choose one of the devices listed there.
2. Verify that the device you have purchased is running genuine GrapheneOS. You can do this by checking the verified boot key hash. We provide the steps to do that in our post-install steps.
3. After verifying that you're running genuine GrapheneOS, you should factory reset the device from recovery to make sure that it has not been tampered with. This will wipe all your data, so make sure to do it before you start using your device.
4. After factory resetting the device, we recommend setting up local or remote attestation via our Auditor app.

How do I build GrapheneOS?

Follow the official GrapheneOS building guide. Third party build guides tend to be out-of-date and often contain misguided advice and errors. If you have trouble with the build process, ask for help from the GrapheneOS chat room.

How can I donate to GrapheneOS?

The donate page provides multiple options for donating to support the GrapheneOS project. GrapheneOS is entirely funded by donations.

Does GrapheneOS make upstream contributions?

GrapheneOS has made substantial contributions to the privacy and security of the Android Open Source Project, along with contributions to the Linux kernel, LLVM, OpenBSD and other projects. Much of our past work is no longer part of the downstream GrapheneOS project because we've successfully landed many patches upstream. We've had even more success with making suggestions and participating in design discussions to steer things in the direction we want. Many upstream changes in AOSP such as removing app access to low-level process, network, timing and profiling information originated in the GrapheneOS project. The needs of the upstream projects are often different from ours, so they'll often reimplement the features in a more flexible way. We've almost always been able to move to using the upstream features and even when we still need our own implementation it helps to have the concepts/restrictions considered by the upstream project and apps needing to be compatible with it. Getting features upstream often leads to an improved user experience and app compatibility.

Is GrapheneOS audited?

Yes, the GrapheneOS code is reviewed by external security researchers, companies and organizations on a continuous basis. This is the main benefit of GrapheneOS being an open source project actively used by other organizations, but it is certainly not something to take for granted based on a project being open source. We put a lot of work into making our code well documented and easy to review. Auditing and code review cannot be done properly as a one time thing but rather need to be done continuously as the code changes. Most of the code review and auditing results for GrapheneOS can be seen from the public pull requests and issue trackers. For example, the French ANSSI organization uses a bunch of our work and has given us suggestions along with reporting issues including a couple issues in hardened_malloc where it could have a false positive detection of memory corruption and wrongly abort the process.[1][2] We've built relationships with security researchers and organizations interested in GrapheneOS or using it which results in a lot of this kind of collaboration. This is not a one-time event but rather something that happens regularly as the code evolves, features are added and we ported to new release. The benefits of a group unfamiliar with the code spending a short time doing a shallow review are greatly overstated in marketing. We instead focus on having people very familiar with areas of the code regularly auditing all our changes. The large number of upstream Android security vulnerabilities discovered by GrapheneOS despite us not actively seeking them out speaks to the results of our review and testing.

Other AOSP-based OS projects including DivestOS and ProtonAOSP using lots of our code results in it getting additional review from outside our project. There have been multiple app compatibility issues fixed as a result of this collaboration. In another case, DivestOS using the GrapheneOS Camera app led to the discovery that we were using incorrect units for the auto-focus region used for QR scanning in our Camera and Auditor apps. This was only a very minor issue reducing the quality of the focus in our apps, but it uncovered a serious bug on certain older devices where memory corruption in the camera service can be triggered by setting an overly large focus region in an app. The devices are unmaintained by the vendors, so this was only reported to the CameraX developers.

GrapheneOS code is not just open source but well documented and organized in order to make it much easier to review. Every change we make to Android Open Source Project repositories is maintained as a clean patch set on top of the latest stable release of AOSP. We regularly clean up our changes by splitting up the commits in a more sensible way, providing better commit messages, stripping away everything but the essential changes, reordering the commits to group the related changes and improving the implementation. This makes sense because these are downstream changes ported between each stable release. Our approach also makes it easy to port our patches elsewhere including upstreaming them. Ease of porting helps us when we port to new quarterly and yearly major releases of AOSP. The best way to review our changes over time is the `git range-diff` command by passing the old range of commits and the new range of commits. For example, you would pass `git range-diff OLD_AOSP_TAG..OLD_GRAPHENEOS_TAG NEW_AOSP_TAG..NEW_GRAPHENEOS_TAG` to see how our changes on top of AOSP have changed between releases without looking at the upstream AOSP changes. This is a major part of our own regular review of our changes and porting work.

Our own standalone projects such as Auditor and AttestationServer also aim to keep the code very easy to audit/review. In those projects, we're writing all the code and choosing the dependencies ourselves, so we can take a minimalist and easy to understand approach to the overall codebase instead of only our changes to it.

We also get broad code review out of attempting to land changes in upstream projects. For example, our CONFIG_FORTIFY_SOURCE feature for the Linux kernel providing a modern take on that feature with support for detecting both read and write overflows was accepted in the Linux kernel in 2017. It has taken several years for all the extra bits of CONFIG_FORTIFY_SOURCE to be accepted upstream. Substantial improvements have been made as part of upstreaming it including upstream deciding to support detecting overflows within objects for the memory family of functions, going far beyond the traditional approach. Based on our issue reports and recommendations, multiple bugs were fixed in both GCC and Clang for FORTIFY_SOURCE support along with a new feature for more dynamic object size checks being added. Android also ended up shipping automatic array bounds checks in addition to FORTIFY_SOURCE for the kernel. We would not have been able to do the same level of testing to uncover the same number of upstream bugs, and we also couldn't have made the more aggressive changes on our own due to lack of resources to review/test the upstream code for compatibility issues.

When was the GrapheneOS project founded?

GrapheneOS was founded as an open source project in 2014. It has been through multiple renames and wasn't initially known as GrapheneOS. See the history page for more details.

How is CopperheadOS related to GrapheneOS?

GrapheneOS was previously known as CopperheadOS. There's a new closed source product using our legacy CopperheadOS branding and code that's not associated with the original project. See our page on the legacy CopperheadOS branding for more details.

Will GrapheneOS create a company?

No, GrapheneOS will remain a non-profit open source project / organization. It will remain an independent organization not strongly associated with any specific company. We partner with a variety of companies and other organizations, and we're interested in more partnerships in the future. Keeping it as an non-profit avoids the conflicts of interest created by a profit-based model. It allows us to focus on improving privacy/security without struggling to build a viable business model that's not in conflict with the success of the open source project.

Who owns the GrapheneOS code and how is it licensed?

The copyright for GrapheneOS code is entirely owned by the GrapheneOS developers and is made available under OSI-approved Open Source licenses. The upstream licensing is inherited for the modifications to those projects and MIT licensing is used for our own standalone projects. GrapheneOS has never had any copyright assignment and the developers have always owned their own contributions, including for code written from 2014 up to the rebranding to GrapheneOS in 2019.

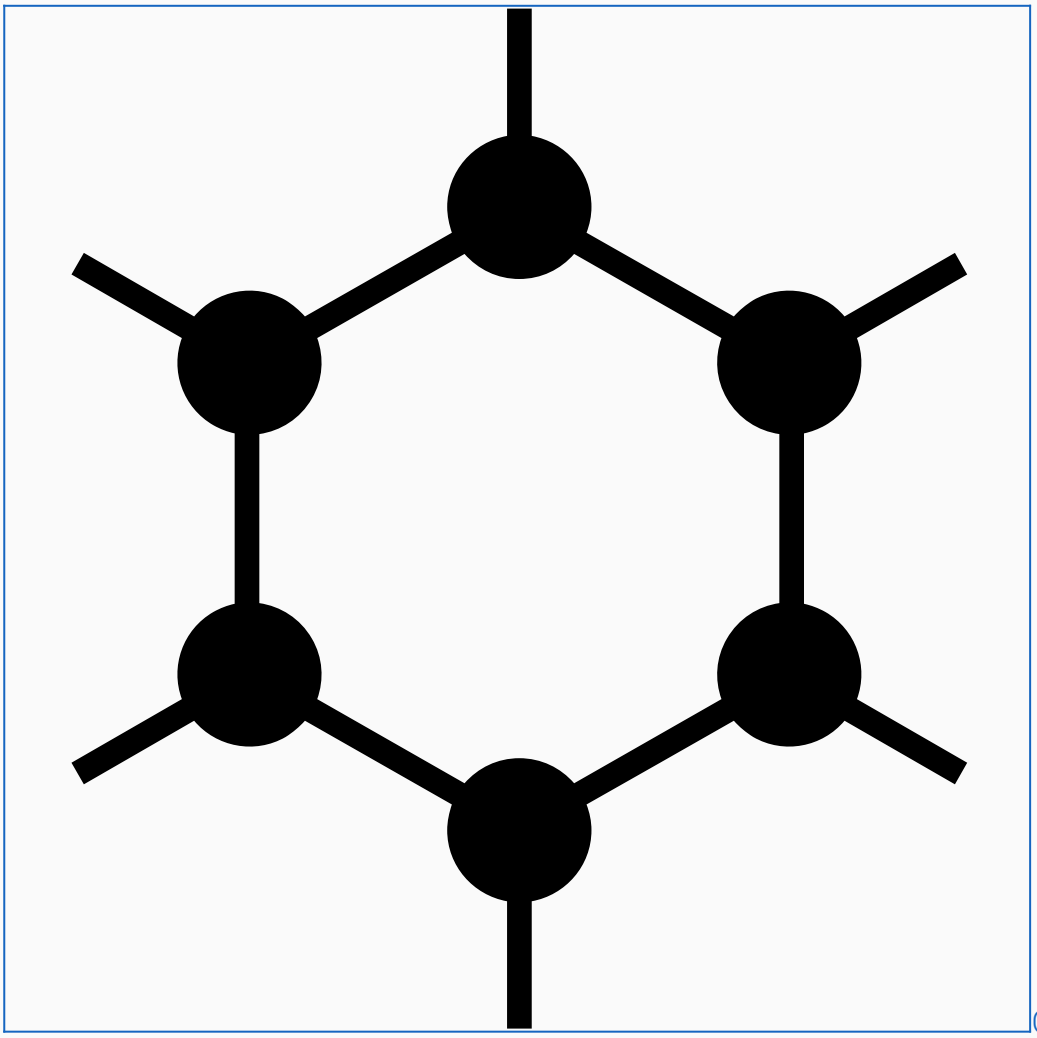
There was an era from September 2016 until the project split from the former sponsor in 2018 where non-commercial usage licensing was used for the official builds and from January 2017 onwards for revisions to the existing permissively licensed code. This was an attempt to prop up the sponsor that was supposed to be supporting the open source project. This did not impact ownership of the code and the copyright owners have relicensed the portions of the code that are used by GrapheneOS under open source licenses. GrapheneOS does not contain any code based on code under non-commercial usage licensing.

What about the GrapheneOS name and logo?

The GrapheneOS name and logo are trademarks of the GrapheneOS project. These marks are in the process of being formally registered in Canada and the US. In the meantime, they're protected as common law trademarks.

Derivatives of GrapheneOS that are being published/redistributed should replace the GrapheneOS branding with their own. It needs to be clear to users that it's a distinct OS based on GrapheneOS. Forks of GrapheneOS are not GrapheneOS itself and should not be presented that way.

Only unofficial builds of the official GrapheneOS sources should be referred to as unofficial builds. An unofficial build is a build of the official GrapheneOS sources with the update server URL changed to another server. A project making modifications beyond that isn't simply an unofficial build and should be presented as a distinct OS based on GrapheneOS.



GrapheneOS

- Forum
- Discord
- Matrix
- Hiring
- X
- Mastodon
- Bluesky
- GitHub
- Reddit
- LinkedIn