IN THE UNITED STATES DISTRICT COURT
FOR THE WESTERN DISTRICT OF TEXAS
WACO DIVISION

ANCORA TECHNOLOGIES, INC. Plaintiff, v. GOOGLE, LLC, Defendant.	Civil Action No. 6:21-cv-00735 Jury Trial Requested

COMPLAINT FOR PATENT INFRINGEMENT

This is an action for patent infringement in which Ancora Technologies, Inc. makes the following allegations against Google, LLC ("Google"):

RELATED CASE

1. This case is related to the actions *Ancora Technologies, Inc. v. Roku, Inc.* (W.D. Tex. Jul. 16, 2021); *Ancora Technologies Inc. v. Nintendo Co. Ltd. et al.* (W.D. Tex. Jul. 16, 2021); and *Ancora Technologies Inc. v. Vizio, Inc.* (W.D. Tex. Jul. 16, 2021)—each of which was filed on July 16, 2021, in the United States District Court for the Western District of Texas, Waco Division, asserting infringement of United States Patent No. 6,411,941.

PARTIES

2. Plaintiff Ancora Technologies, Inc. is a corporation organized and existing under the laws of the State of Delaware with a place of business at 23977 S.E. 10th Street, Sammamish, Washington 98075.

3. Defendant Google, LLC is a limited liability corporation organized under the laws of Delaware. Google maintains a regular and established place of business in this district at 500 West

2nd Street, Austin, Texas, 78701. Google may be served with process through its registered agent, the Corporation Service Company, at 211 East 7th Street, Suite 620, Austin Texas 78701. Google is registered to do business in the State of Texas and has been since at least November 17, 2006.

JURISDICTION AND VENUE

4. This action arises under the patent laws of the United States, Title 35 of the United States Code, such that this Court has subject matter jurisdiction pursuant to 28 U.S.C. §§ 1331 and 1338(a).

5. This Court also has personal jurisdiction over Google including because Google maintains a regular and established place of business in the Western District of Texas, including at 500 West 2nd Street, Austin, Texas, 78701.

6. In addition, directly or through intermediaries, Google has committed acts within the Western District of Texas giving rise to this action and/or has established minimum contacts with the Western District of Texas such that the exercise of jurisdiction would not offend traditional notions of fair play and substantial justice.

7. For example, Google has placed or contributed to placing infringing products like the Nexus 6P and Pixelbook into the stream of commerce via an established distribution channel knowing or understanding that such products would be sold and used in the United States, including in the Western District of Texas.

8. Further, on information and belief, Google also has derived substantial revenues from infringing acts in the Western District of Texas, including from the sale and use of infringing products like the Nexus 6P and Pixelbook.

9. In addition, venue is proper under 28 U.S.C. § 1391(b)-(c) and 28 U.S.C. § 1400 as Google maintains a regular and established place of business in the Western District of Texas,

including at least at 500 West 2nd Street, Austin, Texas, 78701. *In re HTC Corp.*, 889 F.3d 1349, 1354 (Fed. Cir. 2018); *In re Cray Inc.*, 871 F.3d 1355, 1362-63 (Fed. Cir. 2017).

10. Furthermore, Google employs at its regular and established place of business at 500 West 2nd Street, Austin, Texas, individuals with responsibility for Google Cloud servers which, as set forth below, are utilized by Google to transmit infringing over-the-air ("OTA") software updates to the Accused Devices.

THE ASSERTED PATENT

11. This lawsuit asserts causes of action for infringement of United States Patent No. 6,411,941 ("the '941 Patent"), which is entitled "Method of Restricting Software Operation Within a License Limitation."

12. The U.S. Patent and Trademark Office duly and legally issued the '941 Patent on June 25, 2002.

13. Subsequent to issue, and at least by December 21, 2004, all right, title, and interest in the '941 Patent, including the sole right to sue for any infringement, were assigned to Ancora Technologies, Inc., which has held, and continues to hold, all right, title, and interest in the '941 Patent.

14. The president of Ancora Technologies, Inc.—Mr. Miki Mullor—is one of the inventors of the '941 Patent.

15. A reexamination certificate to the '941 Patent subsequently was issued on June 1, 2010.

16. Since being assigned to Ancora Technologies, Inc., the '941 Patent has been asserted in patent infringement actions filed against Microsoft Corporation, Dell Incorporated, Hewlett Packard Incorporated, Toshiba America Information Systems, Apple Inc., HTC America, Inc., HTC

Corporation, Samsung Electronics Co., Ltd., Samsung Electronics America, Inc., LG Electronics, Inc., LG Electronics U.S.A., Inc., Sony Mobile Communications AB, Sony Mobile Communications, Inc., Sony Mobile Communications (USA) Inc., Lenovo Group Ltd., Lenovo (United States) Inc., Motorola Mobility, LLC, TCT Mobile (US) Inc., and Huizhou TCL Mobile Communication Co., Ltd.

17. In the course of these litigations, a number of the '941 Patent's claim terms have been construed, and the validity of the '941 Patent has been affirmed repeatedly.

18. For example, in December 2012, the United States District Court for the Northern District of California issued a claim construction order construing the terms (1) "volatile memory"; (2) "non-volatile memory"; (3) "BIOS"; (4) "program"; (5) "license record"; and (6) "verifying the program using at least the verification structure." *Ancora Techs., Inc. v. Apple Inc.*, No. 11–CV–06357 YGR, 2012 WL 6738761, at *1 (N.D. Cal. Dec. 31, 2012).

19. Further, the court rejected Apple's indefiniteness arguments and further held that, at least with respect to Claims 1-3 and 5-17, "[t]he steps of the Claim do not need to be performed in the order recited." *Ancora Techs., Inc. v. Apple Inc.*, No. 11–CV–06357 YGR, 2012 WL 6738761, at *5, *13 (N.D. Cal. Dec. 31, 2012).

20. Subsequently, the United States Court of Appeals for the Federal Circuit affirmed the district court's rejection of Apple's indefiniteness argument. *Ancora Techs., Inc. v. Apple, Inc.*, 744 F.3d 732, 739 (Fed. Cir. 2014).

21. The Federal Circuit also agreed with Ancora Technologies, Inc. that "the district court erred in construing 'program' to mean 'a set of instructions for software applications that can be executed by a computer'"—holding that, as Ancora had argued, the term should be accorded its

normal meaning of "'a set of instructions' for a computer." *Ancora Techs., Inc. v. Apple, Inc.*, 744 F.3d 732, 734-35, 737 (Fed. Cir. 2014).

22. Subsequently, in a more recent decision, the Federal Circuit held that the '941 Patent satisfied § 101 as a matter of law—stating: "[W]e conclude that claim 1 of the '941 patent is not directed to an abstract idea." *Ancora Techs., Inc. v. HTC Am., Inc.*, 908 F.3d 1343 (Fed. Cir. 2018), *as amended* (Nov. 20, 2018).

23. In addition, the Patent Trial and Appeal Board rejected HTC's request to institute covered business method review proceedings on the '941 Patent—explaining that "the '941 [P]atent's solution to the addressed problem is rooted in technology, and thus, is a 'technical solution'" and also rejecting HTC's argument that "the '941 [P]atent recites a technological solution that is not novel and nonobvious."

24. This Court likewise issued a claim construction order construing or adopting the plain and ordinary meaning of various claims of the '941 Patent, including (1) "non-volatile memory"; (2) "license"; (3) "license record"; (4) "volatile memory"; (5) "BIOS"; (6) "memory of the BIOS"; (7) "program"; (8) "selecting a program residing in the volatile memory"; (9) "using an agent to set up a verification structure in the erasable, non-volatile memory of the BIOS"; (10) "set up a verification structure"; (11) "verifying the program using at least the verification structure"; (12) "acting on the program according to the verification"; (13) "first non-volatile memory area of the computer"; (14) the Claim 1 preamble; and (15) the order of Claim 1 steps. *Ancora Technologies, Inc. v. LG Electronics, Inc.*, 1:20-cv-00034-ADA, at Dkt. 69 (W.D Tex. June 2, 2020).

25. Finally, and most recently, the United States District Court for the Central District of California issued a claim construction order construing the terms (1) "volatile memory"; (2) "selecting a program residing in the volatile memory"; (3) "set up a verification structure"; (4)

"license record"; (5) "memory of the BIOS"; and (6) the whole of Claim 8. *Ancora Techs., Inc v. TCT Mobile (US), Inc., et al.*, No. 8:19-cv-02192-GW-AS, ECF No. 66 & 69 (C.D. Cal. Nov. 18-19, 2020).

COUNT 1 – INFRINGEMENT

26. Plaintiff repeats and incorporates by reference each preceding paragraph as if fully set forth herein and further state:

27. Google has infringed the '941 Patent in violation of 35 U.S.C. § 271(a) by, prior to the expiration of the '941 Patent, selling, and/or offering for sale in the United States, and/or importing into the United States, without authorization, products and/or operating system software for products that are capable of performing at least Claim 1 of the '941 Patent literally or under the doctrine of equivalents and, without authorization, then causing such products to perform each step of at least Claim 1 of the '941 Patent.

28. At a minimum, such Accused Products include those servers/software utilized by Google to transmit an over-the-air ("OTA") software update, as well as those smartphones, laptops, smart home devices and other devices and technology that included Google's operating system software and to which Google sent or had sent an OTA update that caused such device to perform the method recited in Claim 1 prior to the expiration of the '941 Patent.

29. Such Accused Products include products like the Nexus 6P and Pixelbook, which—as detailed below—Google configured such that it would be capable of performing each step of Claim 1 of the '941 Patent and subsequently provided one or more OTA updates that caused the device to perform each step of Claim 1.[1]

---

[1] This description of infringement is illustrative and not intended to be an exhaustive or limiting explanation of every manner in which each Accused Product infringes the '941 patent. Further, on information and belief, the identified functionality of the Nexus 6P and Pixelbook are representative of components and functionality present in all Accused Products.

30. Such Accused Products also include products like the Nexus 6, Nexus 9, Nexus Player, Nexus 6P, Google Pixel C, Chromebook Pixel, Pixelbook, Pixel Slate, Chromecast 1st Generation, Chromecast 2nd Generation, Chromecast Audio, Chromecast Ultra, Chromecast 3rd Generation, Google Home, Home Mini, Home Max, Home Hub / Nest Hub, Google Wifi AC1200, Nest Thermostat, Nest Thermostat v1.12, Nest Thermostat Generation 2, Nest Thermostat Generation 2 V2.8, Nest Protect, Nest Cam Indoor, Nest Thermostat Gen 3, Nest T3019US, Nest T3021US, Nest T3032US, Nest Cam Outdoor, Nest Cam IQ, Nest Secure, Nest Guard, Nest Detect, Nest Tag, Nest Hello, Nest Thermostat E, Nest T4000ES, and Nest T5000SF, as well as any predecessor models to such devices, to which Google sent, or had sent, an OTA update prior to the expiration of the '941 Patent.

31. For example, Claim 1 of the '941 Patent claims "a method of restricting software operation within a license for use with a computer including an erasable, non-volatile memory area of a BIOS of the computer, and a volatile memory area; the method comprising the steps of: [1] selecting a program residing in the volatile memory, [2] using an agent to set up a verification structure in the erasable, non-volatile memory of the BIOS, the verification structure accommodating data that includes at least one license record, [3] verifying the program using at least the verification structure from the erasable non-volatile memory of the BIOS, and [4] acting on the program according to the verification."

32. When Google transmitted an OTA update like its Nexus 6P Version 7.0, 7.1, 8.0, and 8.1 updates, Google performed and/or caused devices like the Nexus 6P to perform each element of Claim 1 as part of its Google-specified, pre-configured software update process:

What is it?

Verified boot is the process of assuring the end user of the integrity of the software running on a device. It typically starts with a read-only portion of the device firmware which loads code and executes it only after cryptographically verifying that the code is authentic and doesn't have any known security flaws. AVB is one implementation of verified boot.

https://android.googlesource.com/platform/external/avb/+/master/README.md#Build-System-Integration.

OTA Updates

Android devices in the field can receive and install over-the-air (OTA) updates to the system, application software, and time zone rules. This section describes the structure of update packages and the tools provided to build them. It is intended for developers who want to make OTA updates work on new Android devices and those who want to build update packages for released devices.

OTA updates are designed to upgrade the underlying operating system, the read-only apps installed on the system partition, and/or time zone rules; these updates do *not* affect applications installed by the user from Google Play.

https://source.android.com/devices/tech/ota.

Verified Boot

Verified Boot strives to ensure all executed code comes from a trusted source (usually device OEMs), rather than from an attacker or corruption. It establishes a full chain of trust, starting from a hardware-protected root of trust to the bootloader, to the boot partition and other verified partitions including `system`, `vendor`, and optionally `oem` partitions. During device boot up, each stage verifies the integrity and authenticity of the next stage before handing over execution.

In addition to ensuring that devices are running a safe version of Android, Verified Boot check for the correct version of Android with rollback protection. Rollback protection helps to prevent a possible exploit from becoming persistent by ensuring devices only update to newer versions of Android.

In addition to verifying the OS, Verified Boot also allows Android devices to communicate their state of integrity to the user.

https://source.android.com/security/verifiedboot.

Life of an OTA update

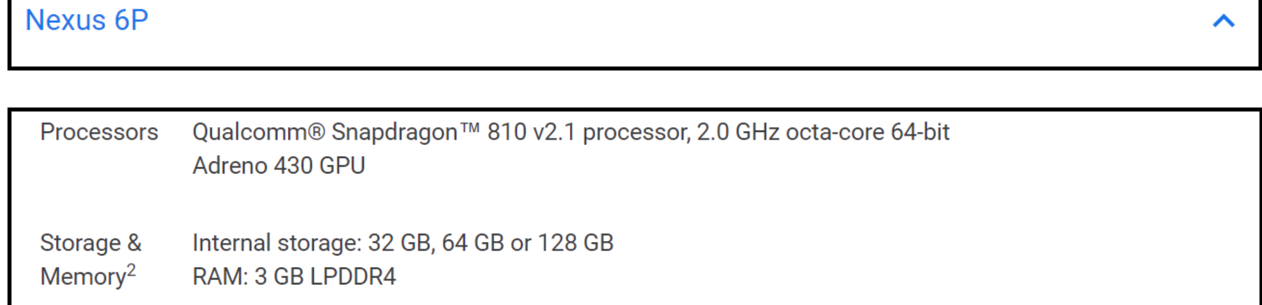
A typical OTA update contains the following steps:

1. Device performs regular check in with OTA servers and is notified of the availability of an update, including the URL of the update package and a description string to show the user.
2. Update downloads to a cache or data partition, and its cryptographic signature is verified against the certificates in `/system/etc/security/otacerts.zip`. User is prompted to install the update.
3. Device reboots into recovery mode, in which the kernel and system in the recovery partition are booted instead of the kernel in the boot partition.
4. Recovery binary is started by init. It finds command-line arguments in `/cache/recovery/command` that point it to the downloaded package.
5. Recovery verifies the cryptographic signature of the package against the public keys in `/res/keys` (part of the RAM disk contained in the recovery partition).
6. Data is pulled from the package and used to update the boot, system, and/or vendor partitions as necessary. One of the new files left on the system partition contains the contents of the new recovery partition.
7. Device reboots normally.
   a. The newly updated boot partition is loaded, and it mounts and starts executing binaries in the newly updated system partition.
   b. As part of normal startup, the system checks the contents of the recovery partition against the desired contents (which were previously stored as a file in `/system`). They are different, so the recovery partition is reflashed with the desired contents. (On subsequent boots, the recovery partition already contains the new contents, so no reflash is necessary.)

The system update is complete! The update logs can be found in `/cache/recovery/last_log.#`.

https://source.android.com/devices/tech/ota/nonab.

33. In particular, each Nexus 6P contains both erasable, non-volatile memory in the form of flash memory and volatile memory in the form of RAM memory. Such non-volatile memory includes a cache or data partition which—on information and belief—is an example of BIOS memory:



Nexus 6P

Processors	Qualcomm® Snapdragon™ 810 v2.1 processor, 2.0 GHz octa-core 64-bit Adreno 430 GPU
Storage & Memory[2]	Internal storage: 32 GB, 64 GB or 128 GB RAM: 3 GB LPDDR4

https://support.google.com/nexus/answer/6102470?hl=en-GB&ref_topic=3415518#zippy=%2Cnexus-p.

> 2. Update downloads to a cache or data partition, and its cryptographic signature is verified against the certificates in `/system/etc/security/otacerts.zip`. User is prompted to install the update.

https://source.android.com/devices/tech/ota/nonab.

34. Further, as detailed above, each Nexus 6P was configured by Google to repeatedly check to see if a new software update was available, including through the following method:

> ## Get the latest Android updates available for you
>
> When you get a notification, open it and tap the update action.
>
> If you cleared your notification or your device has been offline:
>
> 1. Open your phone's Settings app.
> 2. Near the bottom, tap **System** > **Advanced** > **System update**.
> 3. You'll see your update status. Follow any steps on the screen.
>
> ## Get security updates & Google Play system updates
>
> Most system updates and security patches happen automatically. To check if an update is available:
>
> 1. Open your device's Settings app.
> 2. Tap **Security**.
> 3. Check for an update:
>   - To check if a security update is available, tap **Security update**.
>   - To check if a Google Play system update is available, tap **Google Play system update**.
> 4. Follow any steps on the screen.

https://support.google.com/nexus/answer/7680439?hl=en-GB&ref_topic=3415518.

35. During this process, one or more OTA servers owned or controlled by Google set up a verification structure in the erasable, non-volatile memory of the BIOS of the Nexus 6P by transmitting to the device an OTA update, which the Nexus 6P is configured by Google to save to

the erasable, non-volatile memory of its BIOS. As noted previously, on information and belief, such BIOS areas include what Google refers to as the cache or data memory area partition.

36. This OTA update contains a verification structure that includes data accommodating at least one license record.

37. Examples of such a license record includes include a cryptographic signature or key:

Signing Builds for Release

Android OS images use cryptographic signatures in two places:

1. Each .apk file inside the image must be signed. Android's Package Manager uses an .apk signature in two ways:
   - When an application is replaced, it must be signed by the same key as the old application in order to get access to the old application's data. This holds true both for updating user apps by overwriting the .apk, and for overriding a system app with a newer version installed under `/data`.
   - If two or more applications want to share a user ID (so they can share data, etc.), they must be signed with the same key.
2. OTA update packages must be signed with one of the keys expected by the system or the installation process will reject them.

https://source.android.com/devices/tech/ota/sign_builds.

38. Other examples include a cryptographic hash or hash tree:

Verifying Boot

Verified boot requires cryptographically verifying all executable code and data that is part of the Android version being booted before it is used. This includes the kernel (loaded from the `boot` partition), the device tree (loaded from the `dtbo` partition), `system` partition, `vendor` partition, and so on.

Small partitions, such as `boot` and `dtbo`, that are read only once are typically verified by loading the entire contents into memory and then calculating its hash. This calculated hash value is then compared to the *expected hash value*. If the value doesn't match, Android won't load. For more details, see Boot Flow.

Larger partitions that won't fit into memory (such as, file systems) may use a hash tree where verification is a continuous process happening as data is loaded into memory. In this case, the root hash of the hash tree is calculated during run time and is checked against the *expected root hash value*. Android includes the dm-verity driver to verify larger partitions. If at some point the calculated root hash doesn't match the *expected root hash value*, the data is not used and Android enters an error state. For more details, see dm-verity corruption.

The *expected hashes* are typically stored at either the end or beginning of each verified partition, in a dedicated partition, or both. Crucially, these hashes are signed (either directly or indirectly) by the root of trust. As an example, the AVB implementation supports both approaches, see Android Verified Boot for details.

https://source.android.com/security/verifiedboot/verified-boot.

What is it?

Verified boot is the process of assuring the end user of the integrity of the software running on a device. It typically starts with a read-only portion of the device firmware which loads code and executes it only after cryptographically verifying that the code is authentic and doesn't have any known security flaws. AVB is one implementation of verified boot.

The VBMeta struct

The central data structure used in AVB is the VBMeta struct. This data structure contains a number of descriptors (and other metadata) and all of this data is cryptographically signed. Descriptors are used for image hashes, image hashtree metadata, and so-called *chained partitions*. A simple example is the following:

vbmeta
Hash for `boot`
Hashtree metadata for `system`
Hashtree metadata for `vendor`
(signed by key0)

boot payload

system payload hashtree

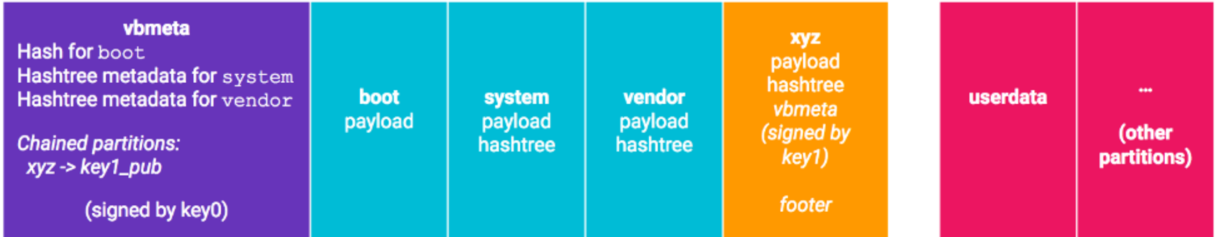
vendor payload hashtree

userdata

... (other partitions)

where the `vbmeta` partition holds the hash for the `boot` partition in a hash descriptor. For the `system` and `vendor` partitions a hashtree follows the filesystem data and the `vbmeta` partition holds the root hash, salt, and offset of the hashtree in hashtree descriptors. Because the VBMeta struct in the `vbmeta` partition is cryptographically signed, the boot loader can check the signature and verify it was made by the owner of `key0` (by e.g. embedding the public part of `key0`) and thereby trust the hashes used for `boot` , `system` , and `vendor` .

A chained partition descriptor is used to delegate authority - it contains the name of the partition where authority is delegated as well as the public key that is trusted for signatures on this particular partition. As an example, consider the following setup:



In this setup the `xyz` partition has a hashtree for integrity-checking. Following the hashtree is a VBMeta struct which contains the hashtree descriptor with hashtree metadata (root hash, salt, offset, etc.) and this struct is signed with `key1` . Finally, at the end of the partition is a footer which has the offset of the VBMeta struct.

This setup allows the bootloader to use the chain partition descriptor to find the footer at the end of the partition (using the name in the chain partition descriptor) which in turns helps locate the VBMeta struct and verify that it was signed by `key1` (using `key1_pub` stored in the chain partition descriptor). Crucially, because there's a footer with the offset, the `xyz` partition can be updated without the `vbmeta` partition needing any changes.

> ### The VBMeta Digest
>
> The VBMeta digest is a digest over all VBMeta structs including the root struct (e.g. in the `vbmeta` partition) and all VBMeta structs in chained partitions. This digest can be calculated at build time using `avbtool calculate_vbmeta_digest` and also at runtime using the `avb_slot_verify_data_calculate_vbmeta_digest()` function. It is also set on the kernel command-line as `androidboot.vbmeta.digest`, see the `avb_slot_verify()` documentation for exact details.
>
> This digest can be used together with `libavb` in userspace inside the loaded operating system to verify authenticity of the loaded vbmeta structs. This is useful if the root-of-trust and/or stored rollback indexes are only available while running in the boot loader.
>
> Additionally, if the VBMeta digest is included in hardware-backed attestation data a relying party can extract the digest and compare it with list of digests for known good operating systems which, if found, provides additional assurance about the device the application is running on.

https://android.googlesource.com/platform/external/avb/+/master/README.md#the-vbmeta-digest.

39. Other examples include x509 and/or root certificate authority:

> ### Certificates and private keys
>
> Each key comes in two files: the *certificate*, which has the extension .x509.pem, and the *private key*, which has the extension .pk8. The private key should be kept secret and is needed to sign a package. The key may itself be protected by a password. The certificate, in contrast, contains only the public half of the key, so it can be distributed widely. It is used to verify a package has been signed by the corresponding private key.

https://source.android.com/devices/tech/ota/sign_builds.

40. Once the verification structure has been set up in the BIOS, the Nexus 6P is configured by Google to reboot, load the OTA update into its volatile memory (e.g., RAM), and then use the at least one license record from the BIOS to verify the OTA update as part of its secure or verified boot process:

> ## Bootloader
>
> A bootloader is a vendor-proprietary image responsible for bringing up the kernel on a device. It guards the device state and is responsible for initializing the Trusted Execution Environment (TEE) and binding its root of trust.
>
> The bootloader is comprised of many things including splash screen. To start boot, the bootloader may directly flash a new image into an appropriate partition or optionally use `recovery` to start the reflashing process that will match how it is done for OTA. Some device manufacturers create multi-part bootloaders and then combine them into a single bootloader.img file. At flash time, the bootloader extracts the individual bootloaders and flashes them all.
>
> Most importantly, the bootloader verifies the integrity of the boot and recovery partitions before moving execution to the kernel and displays the warnings specified in the section Boot state.

https://source.android.com/devices/bootloader.

Life of an OTA update

A typical OTA update contains the following steps:

1. Device performs regular check in with OTA servers and is notified of the availability of an update, including the URL of the update package and a description string to show the user.
2. Update downloads to a cache or data partition, and its cryptographic signature is verified against the certificates in `/system/etc/security/otacerts.zip`. User is prompted to install the update.
3. Device reboots into recovery mode, in which the kernel and system in the recovery partition are booted instead of the kernel in the boot partition.
4. Recovery binary is started by init. It finds command-line arguments in `/cache/recovery/command` that point it to the downloaded package.
5. Recovery verifies the cryptographic signature of the package against the public keys in `/res/keys` (part of the RAM disk contained in the recovery partition).
6. Data is pulled from the package and used to update the boot, system, and/or vendor partitions as necessary. One of the new files left on the system partition contains the contents of the new recovery partition.
7. Device reboots normally.
   a. The newly updated boot partition is loaded, and it mounts and starts executing binaries in the newly updated system partition.
   b. As part of normal startup, the system checks the contents of the recovery partition against the desired contents (which were previously stored as a file in `/system`). They are different, so the recovery partition is reflashed with the desired contents. (On subsequent boots, the recovery partition already contains the new contents, so no reflash is necessary.)

The system update is complete! The update logs can be found in `/cache/recovery/last_log.#`.

https://source.android.com/devices/tech/ota/nonab.

41. If the OTA update is verified, the Nexus 6P is further configured to load and execute the update.

7. Device reboots normally.
   a. The newly updated boot partition is loaded, and it mounts and starts executing binaries in the newly updated system partition.
   b. As part of normal startup, the system checks the contents of the recovery partition against the desired contents (which were previously stored as a file in `/system`). They are different, so the recovery partition is reflashed with the desired contents. (On subsequent boots, the recovery partition already contains the new contents, so no reflash is necessary.)

https://source.android.com/devices/tech/ota/nonab.

42. Further, during the infringing time period, Google performed or caused to be performed each of the Claim 1 steps identified above by providing an OTA update to each Accused Product: https://developers.google.com/android/ota.

43. In addition, during the infringing time period, when Google transmitted an OTA update to Chrome OS for Pixelbook, Google performed and/or caused devices like the Pixelbook to perform each element of Claim 1 as part of its Google-specified, pre-configured software update process.

Chromium OS > Design Documents >

Verified Boot

Abstract

- The Chromium OS team is implementing a verified boot solution that strives to ensure that users feel secure when logging into a Chromium OS device. Verified boot starts with a read-only portion of firmware, which only executes the next chunk of boot code after verification.
- Verified boot strives to ensure that all executed code comes from the Chromium OS source tree, rather than from an attacker or corruption.
- Verified boot is focused on stopping the opportunistic attacker. While verified boot is not expected to detect every attack, the goal is to be a significant deterrent which will be improved upon iteratively.
- Verification during boot is performed on-the-fly to avoid delaying system start up. It uses stored cryptographic hashes and may be compatible with any trusted kernel.

Contents

1 Abstract
2 Goals of verified boot
3 Getting to the kernel safely
4 Extending verification from the kernel on upward
5 Known weaknesses of verified boot
6 Mitigating potential bottlenecks
7 Handling updates
8 The implementation
9 Other issues, ideas, and notes
10 Attack cases

This document extends and expands on the Firmware Boot and Recovery, Verified Boot Crypto, and Verified Boot Data Structures documents.

Verified Boot should provide a mechanism that aids the user in detecting when their system is in need of recovery due to boot path changes. In particular, it should meet these requirements:

- Detect non-volatile memory changes from expected state (rw firmware)
- Detect file system changes relevant to system boot (kernel, init, modules, fs metadata, policies)
- Support functionality upgrades in the field

It is important to note that restraining the boot path to only Chromium-project-supplied code is not a goal. The focus is to ensure that when code is run that is not provided for or maintained by upstream, that the user will have the option to immediately reset the device to a known-good state. Along these lines, there is no dependence on remote attestation or other external authorization. Users will always own their computers.

https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot.

Verified Boot

Even if malware manages to escape the sandbox, your device is still protected. Every time it starts up, it does a self-check called "Verified Boot." If it detects that the system has been tampered with or corrupted in any way, typically it will repair itself without any effort, taking the device back to an operating system that's as good as new.

https://support.google.com/pixelbook/answer/9133875?hl=en.

Life of an A/B Update

In A/B update capable systems, each partition, such as the kernel or root (or other artifacts like DLC), has two copies. We call these two copies active (A) and inactive (B). The system is booted into the active partition (depending on which copy has the higher priority at boot time) and when a new update is available, it is written into the inactive partition. After a successful reboot, the previously inactive partition becomes active and the old active partition becomes inactive.

But everything starts with generating update payloads in (Google) servers for each new system image. Once the update payloads are generated, they are signed with specific keys and stored in a location known to an update server (Omaha).

When the updater client initiates an update (either periodically or user initiated), it first consults different device policies to see if the update check is allowed. For example, device policies can prevent an update check during certain times of a day or they require the update check time to be scattered throughout the day randomly, etc.

Once policies allow for the update check, the updater client sends a request to the update server (all this communication happens over HTTPS) and identifies its parameters like its Application ID, hardware ID, version, board, etc. Then if the update server decides to serve an update payload, it will respond with all the parameters needed to perform an update like the URLs to download the payloads, the metadata signatures, the payload size and hash, etc. The updater client continues communicating with the update server after different state changes, like reporting that it started to download the payload or it finished the update, or reports that the update failed with specific error codes, etc.

Each payload consists of two main sections: metadata and extra data. The metadata is basically a list of operations that should be performed for an update. The extra data contains the data blobs needed by some or all of these operations. The updater client first downloads the metadata and cryptographically verifies it using the provided signatures from the update server's response. Once the metadata is verified as valid, the rest of the payload can easily be verified cryptographically (mostly through SHA256 hashes).

Next, the updater client marks the inactive partition as unbootable (because it needs to write the new updates into it). At this point the system cannot rollback to the inactive partition anymore.

Then, the updater client performs the operations defined in the metadata (in the order they appear in the metadata) and the rest of the payload is gradually downloaded when these operations require their data. Once an operation is finished its data is discarded. This eliminates the need for caching the entire payload before applying it. During this process the updater client periodically checkpoints the last operation performed so in the event of failure or system shutdown, etc. it can continue from the point it missed without redoing all operations from the beginning.

During the download, the updater client hashes the downloaded bytes and when the download finishes, it checks the payload signature (located at the end of the payload). If the signature cannot be verified, the update is rejected.

After the inactive partition is updated, the entire partition is re-read, hashed and compared to a hash value passed in the metadata to make sure the update was successfully written into the partition.

In the next step, the Postinstall process (if any) is called. The postinstall reconstructs the dm-verity tree hash of the ROOT partition and writes it at the end of the partition (after the last block of the file system). The postinstall can also perform any board specific or firmware update tasks necessary. If postinstall fails, the entire update is considered failed.

Then the updater client goes into a state that identifies the update has completed and the user needs to reboot the system. At this point, until the user reboots (or signs out), the updater client will not do any more system updates even if newer updates are available. However, it does continue to perform periodic update checks so we can have statistics on the number of active devices in the field.

After the update proved successful, the inactive partition is marked to have a higher priority (on a boot, a partition with higher priority is booted first). Once the user reboots the system, it will boot into the updated partition and it is marked as active. At this point, after the reboot, The updater client calls into the `chromeos-setgoodkernel` program. The program verifies the integrity of the system partitions using the dm-verity and marks the active partition as healthy. At this point the system is basically updated successfully.

https://chromium.googlesource.com/aosp/platform/system/update_engine/#Life-of-an-A_B-Update.

44. In particular, each Pixelbook contains both erasable, non-volatile memory in the form of flash memory and volatile memory in the form of RAM memory. Such non-volatile memory includes an alternate partition or "slot" which—on information and belief—is an example of BIOS memory:

https://support.google.com/pixelbook/answer/7503982?hl=en&ref_topic=7504137

Processor
Powered by a 7th Gen Intel® Core™ i5-7Y57 or i7-7Y75 processor for faster browsing, gaming, and seamless 4K output to an external monitor. Compare i5 and i7 processors

Memory (RAM)
8GB or 16GB for seamless multitasking

Storage
128GB, 256GB, or 512GB NVMe solid state internal storage

https://support.google.com/pixelbook/answer/7504948?hl=en.

Life of an A/B Update

In A/B update capable systems, each partition, such as the kernel or root (or other artifacts like DLC), has two copies. We call these two copies active (A) and inactive (B). The system is booted into the active partition (depending on which copy has the higher priority at boot time) and when a new update is available, it is written into the inactive partition. After a successful reboot, the previously inactive partition becomes active and the old active partition becomes inactive.

https://chromium.googlesource.com/aosp/platform/system/update_engine/#Life-of-an-A_B-Update.

45. Further, as detailed above, each Pixelbook was configured by Google to repeatedly check to see if a new software update was available, including through the following methods:

Update your Pixelbook's operating system

Your Pixelbook automatically checks for and downloads updates when connected to Wi-Fi or Ethernet.

Finish an update

1. When your device downloads a software update, at the bottom right, look for the "Update available" notification.
2. Select **Restart to Update**.
3. Your device will restart and update.

Note: To learn more about the newest Chrome OS features, in the "Update available" notification, select **Learn more about the latest Chrome OS update**.

If you're using your device at work or school:

1. When your device downloads a software update, at the bottom right, the notification will be colored:
   - **Blue:** An update is recommended.
   - **Orange:** An update is required.
2. Select **Restart to update**.
3. Your device will restart and update.

https://support.google.com/pixelbook/answer/9134767?hl=en.

Pixelbook security

Pixelbooks use the principle of "defense in depth" to provide multiple layers of protection, so if any one layer is bypassed, others are still in effect. So while it's still important to take precautions to protect your data, Pixelbooks let you breathe just a little bit easier. Your Pixelbook has the following security features built-in:

Automatic updates

The most effective way to protect against malware is to ensure all software is up-to-date and has the latest security fixes. This can be difficult to manage on traditional operating systems with many software components from many vendors all with different update mechanisms and user interfaces. Chromebooks manage updates automatically so Chromebooks are always running the latest and most secure version.

https://support.google.com/pixelbook/answer/9133875?hl=en.

46. During this process, one or more OTA servers owned or controlled by Google set up a verification structure in the erasable, non-volatile memory of the BIOS of the Pixelbook by transmitting to the device an OTA update, which the Pixelbook is configured by Google to save to the erasable, non-volatile memory of its BIOS. As noted previously, on information and belief, such BIOS areas include what Google refers to as the target slot or inactive partition.

47. This OTA update contains a verification structure that includes data accommodating at least one license record.

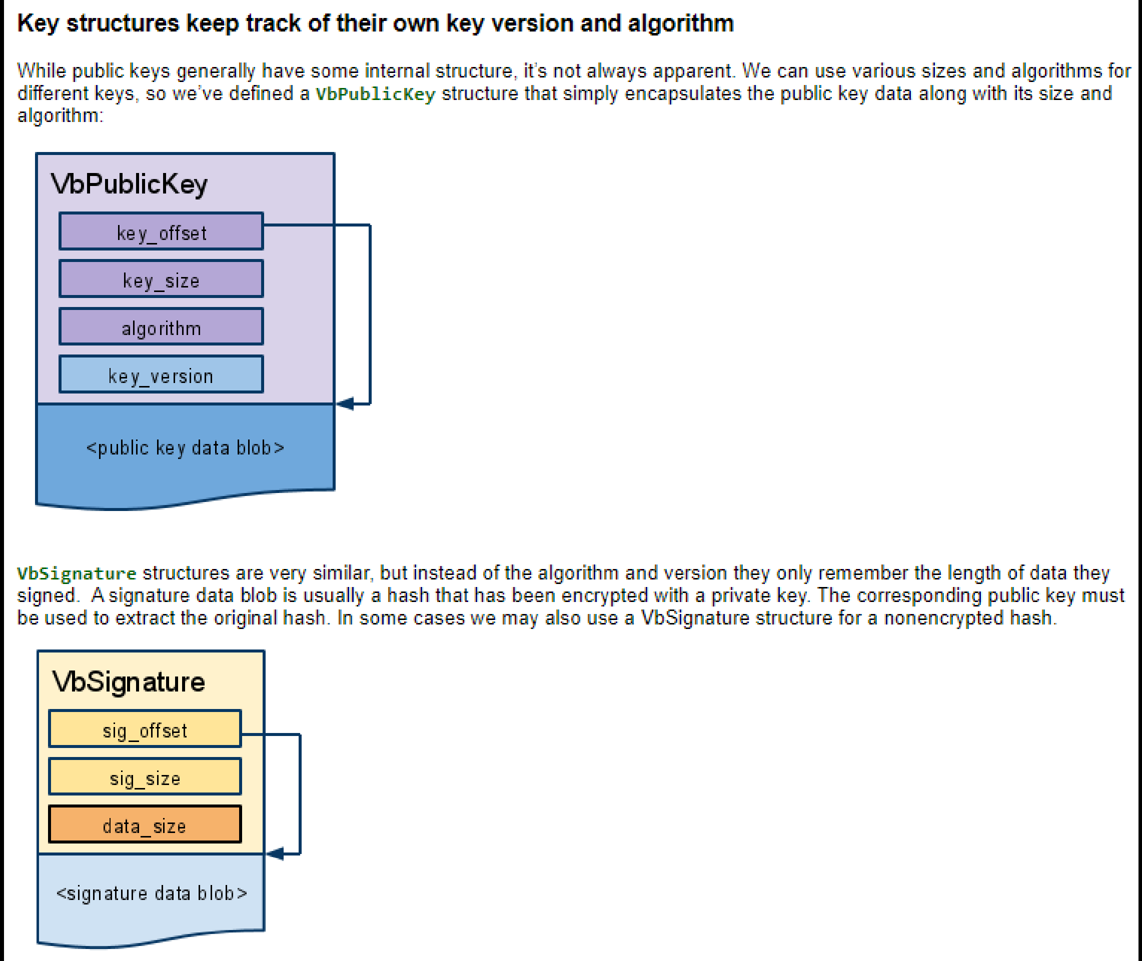
48. Examples of such a license record include a cryptographic signature or key or a cryptographic hash or hash tree:

> During the download, the updater client hashes the downloaded bytes and when the download finishes, it checks the payload signature (located at the end of the payload). If the signature cannot be verified, the update is rejected.
>
> After the inactive partition is updated, the entire partition is re-read, hashed and compared to a hash value passed in the metadata to make sure the update was successfully written into the partition.
>
> In the next step, the Postinstall process (if any) is called. The postinstall reconstructs the dm-verity tree hash of the ROOT partition and writes it at the end of the partition (after the last block of the file system). The postinstall can also perform any board specific or firmware update tasks necessary. If postinstall fails, the entire update is considered failed.
>
> Then the updater client goes into a state that identifies the update has completed and the user needs to reboot the system. At this point, until the user reboots (or signs out), the updater client will not do any more system updates even if newer updates are available. However, it does continue to perform periodic update checks so we can have statistics on the number of active devices in the field.
>
> After the update proved successful, the inactive partition is marked to have a higher priority (on a boot, a partition with higher priority is booted first). Once the user reboots the system, it will boot into the updated partition and it is marked as active. At this point, after the reboot, The updater client calls into the `chromeos-setgoodkernel` program. The program verifies the integrity of the system partitions using the dm-verity and marks the active partition as healthy. At this point the system is basically updated successfully.

https://chromium.googlesource.com/aosp/platform/system/update_engine/#Life-of-an-A_B-Update.

> **The public key and signature headers know how to find their data**
>
> Given a pointer to the `VbPublicKey` or `VbSignature` header, you can find the corresponding data, which may not be immediately following the header itself. This lets us put the header structs as sub-structs in the major structures (preambles, etc.), followed by the data, without needing to pass in data pointers separately.
>
> Functions should take header pointers rather than raw data wherever possible.
>
> **Common structures are reused**
>
> Keys, signatures, and key blocks are used by firmware and kernel algorithms. Common code for generating and parsing is reused.

https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures



Key structures keep track of their own key version and algorithm

While public keys generally have some internal structure, it's not always apparent. We can use various sizes and algorithms for different keys, so we've defined a `VbPublicKey` structure that simply encapsulates the public key data along with its size and algorithm:

`VbSignature` structures are very similar, but instead of the algorithm and version they only remember the length of data they signed. A signature data blob is usually a hash that has been encrypted with a private key. The corresponding public key must be used to extract the original hash. In some cases we may also use a VbSignature structure for a nonencrypted hash.

https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures.

The public key and signature headers know how to find their data

Given a pointer to the `VbPublicKey` or `VbSignature` header, you can find the corresponding data, which may not be immediately following the header itself. This lets us put the header structs as sub-structs in the major structures (preambles, etc.), followed by the data, without needing to pass in data pointers separately.

Functions should take header pointers rather than raw data wherever possible.

Common structures are reused

Keys, signatures, and key blocks are used by firmware and kernel algorithms. Common code for generating and parsing is reused.

https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures.

Abstract

- The Chromium OS team is implementing a verified boot solution that strives to ensure that users feel secure when logging into a Chromium OS device. Verified boot starts with a read-only portion of firmware, which only executes the next chunk of boot code after verification.
- Verified boot strives to ensure that all executed code comes from the Chromium OS source tree, rather than from an attacker or corruption.
- Verified boot is focused on stopping the opportunistic attacker. While verified boot is not expected to detect every attack, the goal is to be a significant deterrent which will be improved upon iteratively.
- Verification during boot is performed on-the-fly to avoid delaying system start up. It uses stored cryptographic hashes and may be compatible with any trusted kernel.

This document extends and expands on the Firmware Boot and Recovery, Verified Boot Crypto, and Verified Boot Data Structures documents.

Verified Boot should provide a mechanism that aids the user in detecting when their system is in need of recovery due to boot path changes. In particular, it should meet these requirements:

- Detect non-volatile memory changes from expected state (rw firmware)
- Detect file system changes relevant to system boot (kernel, init, modules, fs metadata, policies)
- Support functionality upgrades in the field

It is important to note that restraining the boot path to only Chromium-project-supplied code is not a goal. The focus is to ensure that when code is run that is not provided for or maintained by upstream, that the user will have the option to immediately reset the device to a known-good state. Along these lines, there is no dependence on remote attestation or other external authorization. Users will always own their computers.

Contents

1 Abstract
2 Goals of verified boot
3 Getting to the kernel safely
4 Extending verification from the kernel on upward
5 Known weaknesses of verified boot
6 Mitigating potential bottlenecks
7 Handling updates
8 The implementation
9 Other issues, ideas, and notes
10 Attack cases

http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot.

49. Other examples include x509 and/or root certificate authority:

The certificates are categorised as "Key" and "Content" certificates. Key certificates are used to verify public keys which have been used to sign content certificates. Content certificates are used to store the hash of a boot loader image. An image can be authenticated by calculating its hash and matching it with the hash extracted from the content certificate. The SHA-256 function is used to calculate all hashes. The public keys and hashes are included as non-standard extension fields in the X.509 v3 certificates.

https://chromium.googlesource.com/chromiumos/third_party/arm-trusted-firmware/+/v1.2-rc0/docs/trusted-board-boot.md.

50. Once the verification structure has been set up in the BIOS, the Pixelbook is configured by Google to apply the update to the inactive partition or slot, and upon reboot, load the OTA update into its volatile memory (e.g., RAM), and then use the at least one license record from the BIOS to verify the OTA update as part of its secure or verified boot process prior to launching the operating system:

Firmware (A/B) setup

This firmware sets up a minimal set of hardware components so that the boot loader can load the kernel from the normal boot drive. For example, the SATA or eMMC controller.

Pseudocode

1. Initialize chipset / file system sufficiently to jump to Boot Loader code.
2. Jump to Boot Loader code.

Firmware (A/B) boot loader

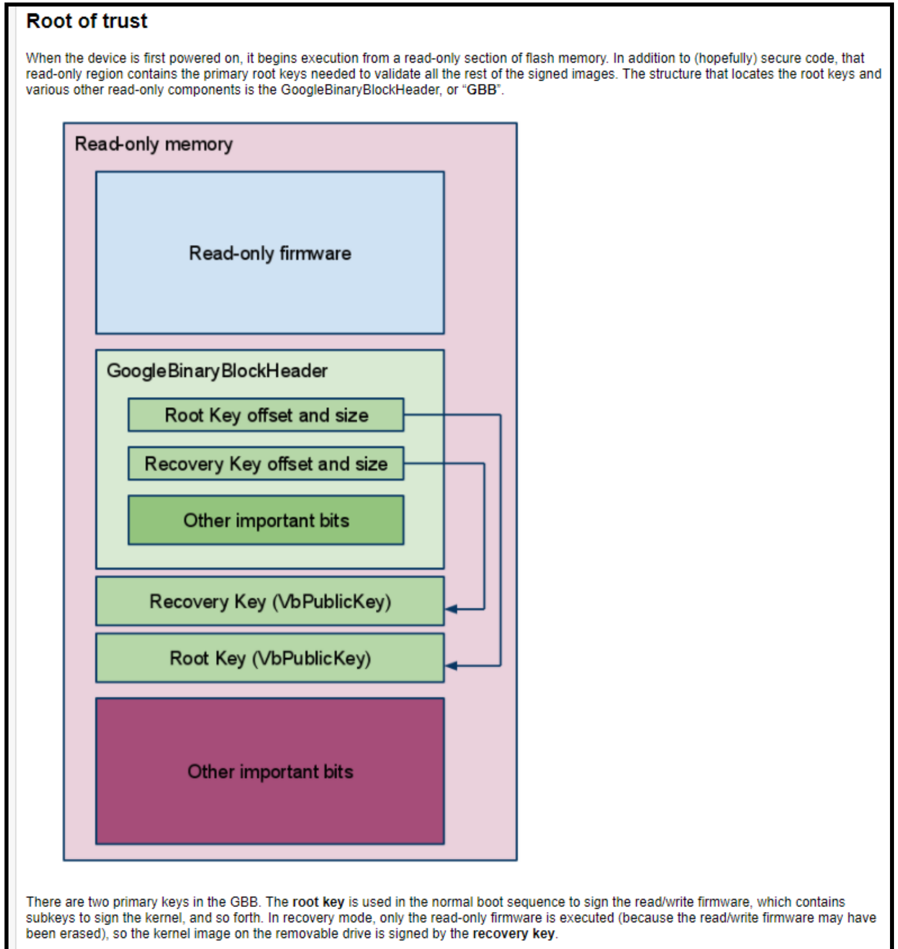
The boot loader is only designed to load Chromium OS. We can go directly from firmware bootstrap to the kernel in the disk.

Pseudocode

1. Verify the partition table on the disk looks sane.
2. Load kernel A from the disk.
3. Verify signature of kernel.
4. If signature is invalid:
    1. If this is kernel A, retry with kernel B.
    2. Else this is kernel B. Both kernels are bad, so set the recovery-mode cookie non-volatile register and reboot into recovery firmware.
5. If kernel was signed with a public key not known to the boot loader, this is a developer kernel:
    1. Initialize the display.
    2. Display scary developer mode warning to user. For example: "Google Chrome OS is not installed. Press space bar to repair."
    3. Wait for keypress or 30-second delay before continuing.
    4. If key pressed was Space bar, Enter, or Esc, jump to Recovery Firmware.
    5. If key pressed was Control+D, dismiss screen.
    6. Ignore other key presses.
6. Continue booting the kernel.

https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery.

51. If the OTA update is verified, the Pixelbook is further configured to load and execute the update.


Root of trust

When the device is first powered on, it begins execution from a read-only section of flash memory. In addition to (hopefully) secure code, that read-only region contains the primary root keys needed to validate all the rest of the signed images. The structure that locates the root keys and various other read-only components is the GoogleBinaryBlockHeader, or "**GBB**".

There are two primary keys in the GBB. The **root key** is used in the normal boot sequence to sign the read/write firmware, which contains subkeys to sign the kernel, and so forth. In recovery mode, only the read-only firmware is executed (because the read/write firmware may have been erased), so the kernel image on the removable drive is signed by the **recovery key**.

https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures.

52. Further, during the infringing time period, Google performed or caused to be performed each of the Claim 1 steps identified above by providing an OTA update to each Accused Product.

53. In addition, and as described above for the Google Nexus 6P and Pixelbook, during the infringing time period Google performed or caused to be performed each of the Claim 1 steps identified and described above when it transmitted an OTA update to other Accused Products such as the Chromecast or Nest Thermostat, Google performed and/or caused devices like the Chromecast and Nest Thermostat to perform each element of Claim 1 as part of its Google-specified, pre-configured software update process, including as reported by various third parties.

Updates to your Chromecast

To enjoy the latest and greatest features available on Chromecast, your device may need to update to the most recent software version. This is done automatically as part of setup so there's nothing you need to do to receive the update.

Note: During the update, Chromecast will not be available to cast. Please wait until the update is complete to try to cast.

Here's an overview of what you should expect during the update:

- Your Google Home app will provide indication of the setup progress.
- You can check the status of your update on the TV.
- You can use the LED on the side of the Chromecast to verify it is still receiving the update.
- The update typically takes up to 10 minutes. If the update is taking much longer than expected, please check if the device is still updating by checking the LED status. It will be pulsing red while receiving the update.
- If needed, rebooting the Chromecast by unplugging it from the power source, waiting one minute, and plugging it back in will automatically restart the update.

https://support.google.com/chromecast/answer/6292664?hl=en-IN.

Find your device's firmware version

Firmware is the software installed on Google Nest or Home speaker or display. When a firmware update is available, your device will automatically download the update via an Over-the-Air (OTA) update.

Your speaker or display must be set up and connected to the internet to receive the firmware update.

https://support.google.com/googlenest/answer/7188572?co=GENIE.Platform%3DAndroid&hl=en-AU.

How to get software updates

Nest products will **keep themselves automatically updated** as long as they're connected to Wi-Fi.

But a few things can temporarily delay an update. For example, if you're actively changing a setting while your product is trying to update itself, it might need to wait for you to finish.

For more details, pick a specific Nest product below.

https://support.google.com/googlenest/answer/9335964?hl=en&co=GENIE.Platform=Android#zippy=%2Chow-to-get-software-updates

Find your device's firmware version

Firmware is the software installed on Google Nest or Home speaker or display. When a firmware update is available, your device will automatically download the update via an Over-the-Air (OTA) update.

Your speaker or display must be set up and connected to the internet to receive the firmware update.

https://support.google.com/googlenest/answer/7188572?co=GENIE.Platform%3DAndroid&hl=en-AU.

Nest thermostat software update history

Important: Some software updates may take a few weeks to make it to all Nest thermostats that are connected to Wi-Fi.

Select the thermostat you have:

Nest Thermostat Nest Thermostat E

Nest Learning Thermostat

https://support.google.com/googlenest/answer/9263516?hl=en#zippy=%2Cnest-thermostat-e-and-nest-learning-thermostat.

Ways to Update IoT Devices OTA

There are three main methods to update IoT devices OTA, no matter how many devices you're updating in the fleet.

1. Edge-to-Cloud OTA Updates

With this method, the IoT device is connected to the Internet and receives the update from a remote server directly. The device can handle updates to both its firmware and the software it runs. Most consumer IoT devices are updated this way because they are close to Wi-Fi signals in people's homes and small commercial locations.

Update Method Example: Amazon Echo Dot update, Google Home update, Nest thermostat update

https://dzone.com/articles/the-best-ways-to-update-iot-devices-over-the-air.

Unlike regular phone Android, Android Things is not customizable by third-parties. All Android Things devices use an OS image direct from Google, and Google centrally distributes updates to all Android Things devices for three years. Android Things doesn't really have an interface. It's designed to get a device up and running and show a single app, which on the smart displays is the Google Smart Display app. Qualcomm's "Home Hub" platform was purposely built to run Android Things and this Google Assistant software—the SD624 is for smart displays, while the less powerful SDA212 is for speakers.

https://arstechnica.com/gadgets/2018/10/google-home-hub-under-the-hood-its-nothing-like-other-google-smart-displays/.

Type of System	Update System	Authentication of Update source	Integrity (transit)	Confidentiality (transit)	Code signing	Type of encryption
Papers	Nilsson and Larson[22]	✓	✓	✓	✓	asymmetric/symmetric
	Costa et al.[6]	✓	✓	✗	✓	asymmetric
	Itani et al.[15]	✓	✓	✗	✗	symmetric
	Katzir and Schwartzman[16]	✓	-	-	-	-
	CAO et al.[4]	✓	✓	✓	✗	symmetric / none
Proprietary	Home routers[14]	✗	✗	✗	✗	✗
	Android OS[3]	✓	✓	✗	✓	asymmetric
	NEST thermostat[21]	✓	✓	✓	✓	asymmetric
	Google OnHub router[11]	✓	✓	✓	✓	asymmetric

Table 5: Summary of guarantees provided by systems in related works

https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997256539/MEIC-ist173009-Tomas-Pinho-resumo_alargado.pdf.

54. Further, Google expressly conditions participation in the OTA update process and the receipt of the benefit of a software update on the performance of each of the above steps.

55. Primarily, as described above, Google pre-configures/programs each Accused Product to perform the above described steps upon receiving an OTA update from Google.

56. Further, Google not only set the time and conditions under which a user could receive and install an OTA update, but Google required all users to accept and install such updates.

57. For example, Google stated the following in its Terms of Service (applicable from April 14, 2014, to October 15, 2017):

> **About Software in our Services**
>
> When a Service requires or includes downloadable software, this software may update automatically on your device once a new version or feature is available. Some Services may let you adjust your automatic update settings.

https://policies.google.com/terms/archive/20140414.

58. As another example, Google stated the following in its March 25, 2014 Google Chrome OS Terms of Service:

> 4. Software updates
>
> 4.1 The Software may automatically download and install updates from time to time from Google. These updates are designed to improve, enhance and further develop the Software and may take the form of bug fixes, enhanced functions, new software modules and completely new versions. You agree to receive such updates (and permit Google to deliver these to you) as part of your use of the Software.

https://www.google.com/chromebook/termsofservice.html#index.

59. As another example, Google stated the following in its Nest End User License Agreement (EULA):

> **3. Automatic Software Updates.**
>
> Nest may from time to time develop patches, bug fixes, updates, upgrades and other modifications to improve the performance of the Product Software and related services ("Updates"). These may be automatically installed without providing any additional notice or receiving any additional consent. You consent to this automatic update. If you do not want such Updates, your remedy is to stop using the Product. If you do not cease using the Product, you will receive Updates automatically. You acknowledge that you may be required to install Updates to use the Product and the Product Software and you agree to promptly install any Updates Nest provides. Your continued use of the Product is your agreement to this EULA.

https://nest.com/legal/eula/.

60. As another example, Google stated that Google Nest Wifi and Google WiFi receives automatic software updates "to make sure you always have the latest security features and protection from recently discovered security threats":

> **Automatic updates**
>
> Google Nest Wifi and Google Wifi receive automatic software updates to make sure you always have the latest security features and protection from recently discovered security threats. These updates may include open source components and go through several rigorous reviews.
>
> All software updates are signed by Google. Google Nest Wifi and Google Wifi can't download or run any software that isn't signed and verified.

https://support.google.com/googlenest/answer/6309220?hl=en&ref_topic=9832039.

61. Further, Google emphasizes the benefits associated with using OTA updates to update the software of its Accused Products, including by advertising that, in one analysis, 87% of all Nexus owners were "running the latest security update after a month," and that "it takes about one and a half calendar weeks for the OTA to reach every Google device": https://source.android.com/devices/tech/ota/ab/ab_faqs; https://source.android.com/security/bulletin/pixel.

62. As another example, Google stated that severe security vulnerabilities required critical or urgent action to apply OTA software updates as soon as possible, explaining: "The most severe of these issues are Critical security vulnerabilities in device-specific code that could enable arbitrary code execution within the context of the kernel, leading to the possibility of a local permanent device compromise, which may require reflashing the operating system to repair the device." https://source.android.com/security/bulletin/2016-12-01.

63. As another example, Google advertises the "great new features" and other benefits associated with updating Accused Products to new versions of the Android Operating System. *See* https://developer.android.com/about/versions/pie/android-

9.0; https://developer.android.com/about/versions/oreo; https://developer.android.com/about/versions/nougat; https://developer.android.com/about/versions/marshmallow.

64. Google also identified the specific benefits associated with each OTA update: https://source.android.com/security/bulletin; https://source.android.com/security/bulletin/pixel; https://developer.android.com/about/versions; https://chromereleases.googleblog.com/search/label/Chrome%20OS; https://support.google.com/googlenest/answer/9263516?hl=en&ref_topic=9361783; .

65. As another example, Google explains that its automatic updates to its Nest devices, such as the Nest Thermostat, "improve the performance of the Product Software": https://nest.com/legal/eula/.

66. As another example, Google states that its automatic updates to Chrome OS devices, such as its Pixelbook, "are designed to improve, enhance and further develop the Software and may take the form of bug fixes, enhanced functions, new software modules and completely new versions": https://www.google.com/chromebook/termsofservice.html.

67. Further, Google controlled the manner in which each OTA update could be performed, including by pre-configuring each Accused Product such that, upon receiving an OTA update from Google, the device would automatically perform each remaining step of the claimed method.

68. Google also controlled the timing of the performance of such method by determining when to utilize its OTA servers/software to set up a verification structure in each Accused Product.

69. For example, Google uses Google Cloud servers to transmit data to and support the Accused Products, including, on information and belief, its OTA servers to provide OTA updates to Accused Products.

70. Various third parties also have reported such functionality:

BY MICHAEL GROTHAUS 1 MINUTE READ

If you're a Google user, you probably noticed some trouble last night when trying to access Google-owned services. Last night, Google reported several issues with its Cloud Platform, which made several Google sites slow or inoperable. Because of this, many of Google's sites and services–including Gmail, G Suite, and YouTube–were slow or completely down for users in the U.S. and Europe.

However, the Google Cloud outage also affected third-party apps and services that use Google Cloud space for hosting. Affected third-party apps and services include Discord, Snapchat, and even Apple's iCloud services.

But an especially annoying side effect of Google Cloud's downtime was that Nest-branded smart home products for some users just failed to work. According to reports from Twitter, many people were unable to use their Nest thermostats, Nest smart locks, and Nest cameras during the downtime. This essentially meant that because of a cloud storage outage, people were prevented from getting inside their homes, using their AC, and monitoring their babies.

https://www.fastcompany.com/90358396/that-major-google-outage-meant-some-nest-users-couldnt-unlock-doors-or-use-the-ac.

If the above scenarios does not apply to your case, check out these additional steps.

- Try changing your network's 'DNS server' - This can help the network connecting to a closer or more optimized DNS servers, eliminating checking multiple redundant DNS servers. From the **Google Home app** > tap Advanced Networking > It will re-direct you to the **Google Wifi app.**

- From the **Google Wifi app** > Settings Tab (3 dots and a gear) > Network & General > Advanced Networking > **DNS** > select either **"ISP's DNS"** or **"Custom"** if you select **Custom** we recommend putting these - on primary server put in **8.8.8.8**, on the secondary server put in **8.8.4.4** > Save.

- Toggle 'Google cloud services'.

https://support.google.com/googlenest/thread/55022253/new-google-wifi-firmware-12871-57-12-bandwidth-speed-cut-in-half?hl=en.

71. Google also had the right and ability to stop or limit infringement simply by not performing the initial step of using its OTA servers/software to set up a verification structure in each Accused Product. Absent this action by Google, the infringement at issue in this lawsuit would not have occurred.

72. Google's infringement has caused damage to Ancora, and Ancora is entitled to recover from Google those damages that Ancora has sustained as a result of Google's infringement.

DEMAND FOR JURY TRIAL

73. Ancora hereby demands a jury trial for all issues so triable.

PRAYER FOR RELIEF

WHEREFORE, Plaintiff prays for judgment as follows:

A. Declaring that Google has infringed United States Patent No. 6,411,941 in violation of 35 U.S.C. § 271;

B. Awarding damages to Ancora arising out of this infringement, including enhanced damages pursuant to 35 U.S.C. § 284 and prejudgment and post-judgment interest, in an amount according to proof;

C. Awarding such other costs and relief the Court deems just and proper, including any relief that the Court may deem appropriate under 35 U.S.C. § 285.

Date: July 16, 2021

/s/ Andres Healy

Andres Healy (WA 45578)
SUSMAN GODFREY LLP
1201 Third Avenue, Suite 3800
Seattle, Washington 98101
Tel: (206) 516-3880
Fax: 206-516-3883
ahealy@susmangodfrey.com

Lexie G. White (TX 24048876)
SUSMAN GODFREY LLP
1000 Louisiana Street, Suite 5100
Houston, Texas 77002
Tel: (713) 651-9366
Fax: (713) 654-6666
lwhite@susmangodfrey.com

Charles Ainsworth
State Bar No. 00783521
Robert Christopher Bunt
State Bar No. 00787165
PARKER, BUNT & AINSWORTH, P.C.
100 E. Ferguson, Suite 418
Tyler, TX 75702
903/531-3535
E-mail: charley@pbatyler.com
E-mail: rcbunt@pbatyler.com

COUNSEL FOR PLAINTIFF ANCORA TECHNOLOGIES, INC.