Table of Contents

|  |  |  |
| --- | --- | --- |
| Section | Content | Page No |
| 1 | Introduction to Chrome OS | 1 |
| 2 | About Chrome OS | 2 |
| 3 | Architectural Overview and performance | 18 |
| 4 | Design decisions | 24 |
| 5 | Summary | 52 |

Chrome Operating System

Introduction to the Chrome Operating System

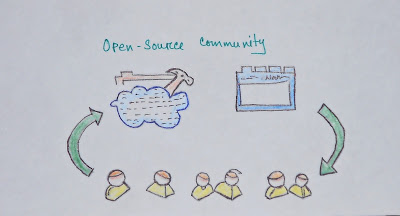
Introduction

Chrome OS is a Linux kernel-based operating system designed by Google. It is derived from the free software Chromium OS and uses the Google Chrome web browser as its principal user interface.

Google announced the project in July 2009, conceiving it as an operating system in which both applications and user data reside in the cloud: hence Chrome OS primarily runs web applications. Source code and a public demo came that November. The first Chrome OS laptop, known as a Chromebook, arrived in May 2011. Initial Chromebook shipments from Samsung and Acer occurred in July 2011.

Chrome OS has an integrated media player and file manager. It supports Chrome Apps, which resemble native applications, as well as remote access to the desktop. Android applications started to become available for the operating system in 2014, and in 2016, access to Android apps in the entire Google Play Store was introduced on supported Chrome OS devices. Reception was initially skeptical, with some observers arguing that a browser running on any operating system was functionally equivalent. As more Chrome OS machines have entered the market, the operating system is now seldom evaluated apart from the hardware that runs it.

Chrome OS is only available pre-installed on hardware from Google manufacturing partners, but there are unofficial methods that allow it to be installed in other equipment. Its open source upstream, Chromium OS, can be compiled from downloaded source code. Early on, Google provided design goals for Chrome OS, but has not otherwise released a technical description.



The core is Linux but unlike it, you cannot install Full Linux Apps. Instead, it let you install Web Apps that run like native apps, and now it encloses Android OS inside a container. So technically, you can now install Android Apps as well. In the future, Google is planning to let users install Full Linux Apps through a special and secure Linux Container. This is more innovative because the apps will have more power and control than Web apps or Android Apps but since they are inside a container, they won’t harm the core OS.

History

Google announced Chrome OS on July 7, 2009, describing it as an operating system in which both applications and user data reside in the cloud. To ascertain marketing requirements, the company relied on informal metrics, including monitoring the usage patterns of some 200 Chrome OS machines used by Google employees. Developers also noted their own usage patterns. Matthew Papakipos, former engineering director for the Chrome OS project, put three machines in his house and found himself logging in for brief sessions: to make a single search query or send a short email.

Chrome OS was initially intended for secondary devices like netbooks, not as a user's primary PC. While Chrome OS supports hard disk drives, Google has requested that its hardware partners use solid-state drives "for performance and reliability reasons" as well as the lower capacity requirements inherent in an operating system that accesses applications and most user data on remote servers. In November 2009 Matthew Papakipos, engineering director for the Chrome OS, claimed that the Chrome OS consumes one-sixtieth as much drive space as Windows 7. The recovery images Google provides for Chrome OS range between 1 and 3 GB.

On November 19, 2009, Google released Chrome OS's source code as the Chromium OS project. At a November 19, 2009, news conference, Sundar Pichai, at the time Google's vice president overseeing Chrome, demonstrated an early version of the operating system. He previewed a desktop which looked very similar to the Chrome browser, and in addition to the regular browser tabs, also had application tabs, which take less space and can be pinned for easier access. At the conference, the operating system booted up in seven seconds, a time Google said it would work to reduce.

Additionally, Chris Kenyon, vice president of OEM services at Canonical Ltd, announced that Canonical was under contract to contribute engineering resources to the project with the intent to build on existing open source components and tools where feasible.

Early Chromebooks

In 2010, Google released the unbranded Cr-48 Chromebook in a pilot program. The launch date for retail hardware featuring Chrome OS was delayed from late 2010 until the next year. On 11 May 2011, Google announced two Chromebooks from Acer and Samsung at Google I/O. The Samsung model was released on 15 June 2011, but the Acer was delayed until mid-July. In August 2011, Netflix announced official support for Chrome OS through its streaming service, allowing Chromebooks to watch streaming movies and TV shows via Netflix. At the time, other devices had to use Microsoft Silverlight to play videos from Netflix. Later in that same month, Citrix released a client application for Chrome OS, allowing Chromebooks to access Windows applications and desktops remotely. Dublin City University became the first educational institution in Europe to provide Chromebooks for its students when it announced an agreement with Google in September 2011.

Expansion

By 2012, demand for Chromebooks had begun to grow, and Google announced a new range of devices, designed and manufactured by Samsung. In so doing, they also released the first Chromebox, the Samsung Series 3, which was Chrome OS's entrance into the world of desktop computers. Although they were faster than the previous range of devices, they were still underpowered compared to other desktops and laptops of the time, fitting in more closely with the Netbook market. Only months later, in October, Samsung and Google released a new Chromebook at a significantly lower price point ($250, compared to the previous Series 5 Chromebooks' $450). It was the first Chromebook to use an ARM processor, one from Samsung's Exynos line. In order to reduce the price, Google and Samsung also reduced the memory and screen resolution of the device. An advantage of using the ARM processor, however, was that the Chromebook didn't require a fan. Acer followed quickly after with the C7 Chromebook, priced even lower ($199), but containing an Intel Celeron processor. One notable way which Samsung reduced the cost of the C7 was to use a laptop hard disk rather than a solid-state drive.

In April 2012, Google made the first update to Chrome OS's user interface since the operating system had launched, introducing a hardware-accelerated window manager called "Aura" along with a conventional taskbar. The additions marked a departure from the operating system's original concept of a single browser with tabs and gave Chrome OS the look and feel of a more conventional desktop operating system. "In a way, this almost feels as if Google is admitting defeat here", wrote Frederic Lardinois on TechCrunch. He argued that Google had traded its original version of simplicity for greater functionality. "That’s not necessarily a bad thing, though, and may just help Chrome OS gain more mainstream acceptance as new users will surely find it to be a more familiar experience." Lenovo and HP followed Samsung and Acer in manufacturing Chromebooks in early 2013 with their own models. Lenovo specifically targeted their Chromebook at students, headlining their press release with "Lenovo Introduces Rugged ThinkPad Chromebook for Schools".

When Google released Google Drive, they also included Drive integration in the next version of Chrome OS (version 20), released in July 2012. While Chrome OS had supported Flash since 2010, by the end of 2012 it had been fully sandboxed, preventing issues with Flash from affecting other parts of Chrome OS. This affected all versions of Chrome including Chrome OS.

Chrome book Pixel

Up to this point, Google had never made their own Chrome OS device. Instead, Chrome OS devices were much more similar to their Nexus line of Android phones, with each Chrome OS device being designed, manufactured, and marketed by third party manufacturers, but with Google controlling the software. However, in February 2013 this changed when Google released the Chromebook Pixel. The Chromebook Pixel was a departure from previous devices. Not only was it entirely Google-branded, but it contained an Intel i5 processor, a high resolution (2,560x1,700) touchscreen display, and came at a price point more competitive with business laptops.

Controversial popularity

By the end of 2013, analysts were undecided on the future of Chrome OS. Although there had been articles predicting the demise of Chrome OS since 2009, Chrome OS device sales continued to increase substantially year-over-year. In mid-2014, Time Magazine published an article titled "Depending on Who's Counting, Chromebooks are Either an Enormous Hit or Totally Irrelevant", which detailed the differences in opinion. This controversy was further spurred by the fact that Intel seemed to decide Chrome OS was a beneficial market for it, holding their own Chrome OS events where they announced new Intel-based Chromebooks, Chromeboxes, and an all-in-one from LG called the Chromebase.

Seizing the opportunity created by the end of life for Windows XP, Google pushed hard to sell Chromebooks to businesses, offering significant discounts in early 2014.

Pwnium competition

In March 2014, Google hosted a hacking contest aimed at computer security experts called "Pwnium". Similar to the Pwn2Own contest, they invited hackers from around the world to find exploits in Chrome OS, with prizes available for attacks. Two exploits were demonstrated there, and a third was demonstrated at that year's Pwn2Own competition. Google patched the issues within a week.

Material Design and App Runtime for Chrome

Although the Google Native Client has been available on Chrome OS since 2010, there originally were few Native Client apps available, and most Chrome OS apps were still web apps. However, in June 2014, Google announced at Google I/O that Chrome OS would both synchronise with Android phones to share notifications and begin to run Android apps, installed directly from the Google Play Store. This, along with the broadening selection of Chromebooks, provided an interesting future for Chrome OS.

At the same time, Google was also moving towards the then-new Material Design visual language for its products, which it would bring to its web products as well as Android Lollipop. One of the first Material Design items to come to Chrome OS was a new default wallpaper, though Google did release some screenshots of a Material Design experiment for Chrome OS that never made it into the stable version.

Chromebox for Meetings

In an attempt to expand its enterprise offerings, Google released the Chromebox for Meetings in February 2014. The Chromebox for Meetings is a kit for conference rooms containing a Chromebox, a camera, a unit containing both a noise-cancelling microphone and speakers, and a remote control. It supports Google Hangouts meetings, Vidyo video conferences, and conference calls from UberConference. Several partners announced Chromebox for Meetings models with Google, and in 2016 Google announced an all-in-one Chromebase for Meetings for smaller meeting rooms.

Hardware

Laptops running Chrome OS are known collectively as "Chromebooks". The first was the CR-48, a reference hardware design that Google gave to testers and reviewers beginning in December 2010. Retail machines followed in May 2011. A year later, in May 2012, a desktop design marketed as a "Chromebox" was released by Samsung. In March 2015 a partnership with AOPEN was announced and the first commercial Chromebox was developed.

In early 2014, LG Electronics introduced the first device belonging to the new all-in-one form factor called "Chromebase". Chromebase devices are essentially Chromebox hardware inside a monitor with built-in camera, microphone and speakers.

The Chromebit is an HDMI dongle running Chrome OS. When placed in an HDMI slot on a television set or computer monitor, the device turns that display into a personal computer. The device was announced in March 2015 and shipped that November. Chrome OS supports dual-monitor setups, on devices with a video-out port.

Applications

Initially, Chrome OS was almost a pure web thin client operating system that relied primarily on servers to host web applications and related data storage. Google gradually began encouraging developers to create "packaged applications", and later, Chrome Apps. The latter employs HTML5, CSS, Adobe Shockwave, and JavaScript to provide a user experience closer to a native application.

In September 2014, Google launched App Runtime for Chrome (beta), which allowed certain ported Android applications to run on Chrome OS. Runtime was launched with four Android applications: Duolingo, Evernote, Sight Words, and Vine. In 2016, Google made the Google Play Store available for Chrome OS, making most Android apps available for supported Chrome OS devices.

Google announced in 2018 that Chrome OS would be getting support for desktop Linux apps. This capability was released to the stable channel with Chrome 69 in October 2018, but was still marked as beta.

Chrome Apps

Google has encouraged developers to build not just conventional Web applications for Chrome OS, but Chrome Apps (formerly known as Packaged apps). From a user perspective, Chrome Apps resemble conventional native applications: they can be launched outside of the Chrome browser, are offline by default, can manage multiple windows, and interact with other applications. Technologies employed include HTML5, JavaScript, and CSS.

Integrated media player, file manager

Google integrates a media player into both Chrome OS and the Chrome browser, enabling users to play back MP3s, view JPEGs, and handle other multimedia files while offline. It supports DRM videos.

Chrome OS also includes an integrated file manager, resembling those found on other operating systems, with the ability to display directories and the files they contain from both Google Drive and local storage, as well as to preview and manage file contents using a variety of Web applications, including Google Docs and Box. Since January 2015, Chrome OS can also integrate additional storage sources into the file manager, relying on installed extensions that use the File System Provider API.

Remote application access and virtual desktop access

In June 2010, Google software engineer Gary Kačmarčík wrote that Chrome OS will access remote applications through a technology unofficially called "Chromoting", which would resemble Microsoft's Remote Desktop Connection. The name has since been changed to "Chrome Remote Desktop", and is like "running an application via Remote Desktop Services or by first connecting to a host machine by using RDP or VNC". Initial roll-outs of Chrome OS laptops (Chromebooks) indicate an interest in enabling users to access virtual desktops.

Android applications

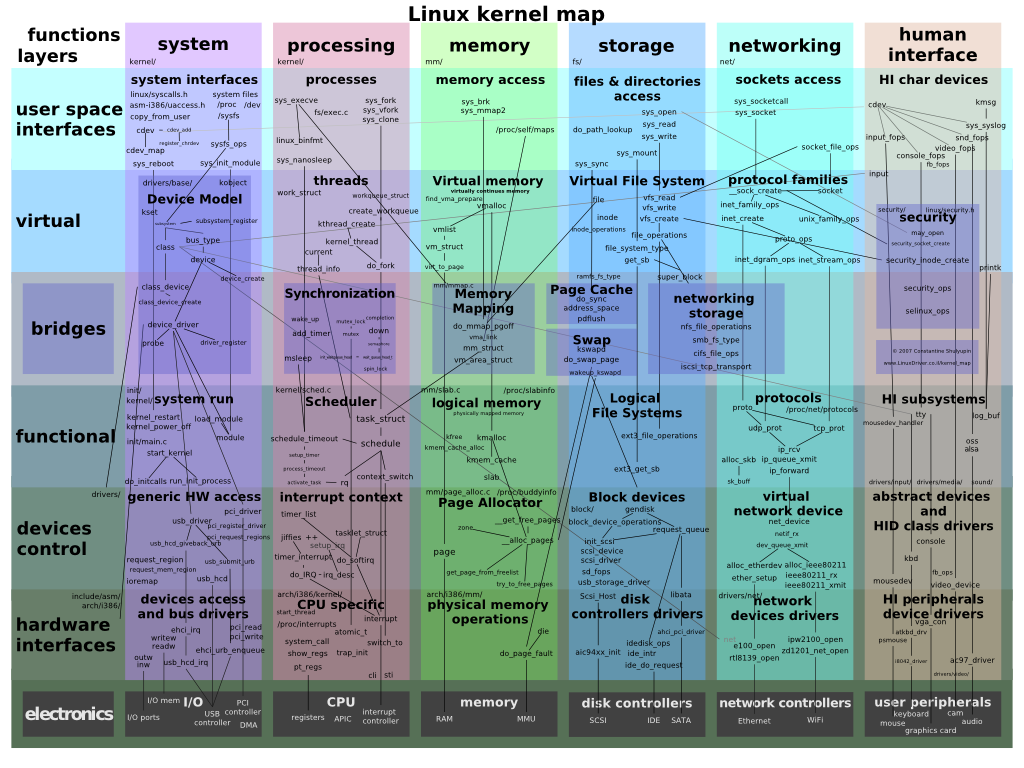
At Google I/O 2014, a proof of concept showing Android applications, including Flipboard, running on Chrome OS was presented. In September 2014, Google introduced a beta version of the App Runtime for Chrome (ARC), which allows selected Android applications to be used on Chrome OS, using a Native Client-based environment that provides the platforms necessary to run Android software. Android applications do not require any modifications to run on Chrome OS, but may be modified to better support a mouse and keyboard environment. At its introduction, Chrome OS support was only available for selected Android applications.

In 2016, Google introduced the ability to run Android apps on supported Chrome OS devices, with access to the entire Google Play Store. The previous Native Client-based solution was dropped in favor of a container containing Android's frameworks and dependencies (initially based on Android Marshmallow), which allows Android apps to have direct access to the Chrome OS platform, and allow the OS to interact with Android contracts such as sharing. Engineering director Zelidrag Hornung explained that ARC had been scrapped due to its limitations, including its incompatibility with the Android Native Development Toolkit (NDK), and that it was unable to pass Google's own compatibility test suite.

Linux Apps

Since 2013 it has been possible to run Linux applications in Chrome OS through the use of Crouton, a third-party set of scripts that allows access to a Linux distribution such as Ubuntu. However, in 2018 Google announced that desktop Linux apps were officially coming to Chrome OS. The main benefit claimed by Google of their official Linux application support is that it can run without enabling developer mode, keeping many of the security features of Chrome OS. It was noticed in the Chromium OS source code in early 2018. Early parts of Crostini were made available for the Google Pixelbook via the dev channel in February 2018 as part of Chrome OS version 66, and it was enabled by default via the beta channel for testing on a variety of chromebooks in August 2018 with version 69.

Kernel map



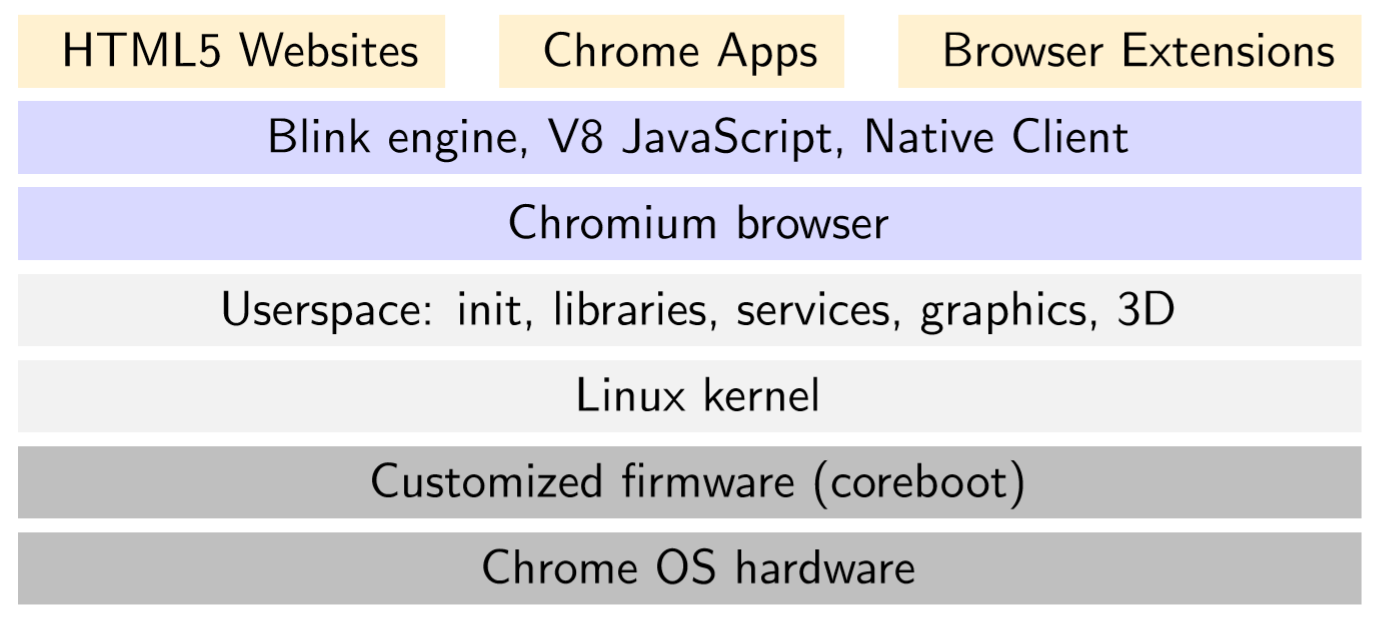
Architecture

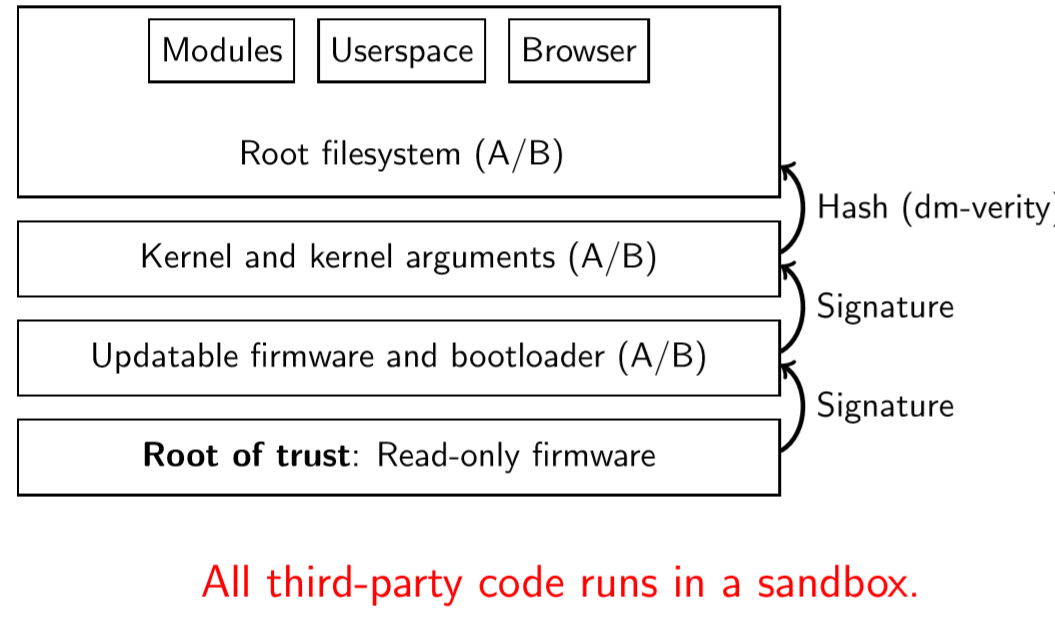
Google's project for supporting Linux applications in Chrome OS is called Crostini, named for the Italian bread-based starter, and as a pun on Crouton. Crostini runs a virtual machine through a virtual machine monitor called crosvm, which uses Linux's built-in KVM virtualization tool. Although crosvm supports multiple virtual machines, the one used for running Linux apps, Termina, contains a basic Chrome OS kernel and userland utilities, in which it runs containers based on Linux containers (specifically LXD).

Architecture of the chrome OS:

Chrome OS is built on top of the Linux kernel. Originally based on Ubuntu, its base was changed to Gentoo Linux in February 2010. In preliminary design documents for the Chrome OS open source project, Google described a three-tier architecture: firmware, browser and window manager, and system-level software and userland services.

The firmware contributes to fast boot time by not probing for hardware, such as floppy disk drives, that are no longer common on computers, especially netbooks. The firmware also contributes to security by verifying each step in the boot process and incorporating system recovery. System-level software includes the Linux kernel that has been patched to improve boot performance. Userland software has been trimmed to essentials, with management by Upstart, which can launch services in parallel, respawn crashed jobs, and defer services in the interest of faster booting. The window manager handles user interaction with multiple client windows much like other X window managers.





Chrome OS’s architecture has three distinct components, all of which have been tuned to make the OS fast:

1.    The firmware: The firmware provides the quick boot time and incorporates system recovery, if and when it's needed. The firmware also contributes to security by verifying each step in the boot process.  
  
2.    The System-level/user-land software: Chrome OS's system-level software includes a Linux kernel that has been patched to increase the speed and performance of the system. User land software has been trimmed to essentials, with management by Upstart, which can launch services in parallel, re-spawn crashed jobs, and defer services in the interest of faster booting.  
  
3.    The Window Manager: Window manager cum browser handles the user interaction and is also used to log on and use the internet services and various Google apps.

Security:

In March 2010, Google software security engineer Will Drewry discussed Chrome OS security. Drewry described Chrome OS as a "hardened" operating system featuring auto-updating and sandbox features that will reduce malware exposure. He said that Chrome OS netbooks will be shipped with Trusted Platform Module (TPM), and include both a "trusted bootpath" and a physical switch under the battery compartment that actuates a developer mode. That mode drops some specialized security functions but increases developer flexibility. Drewry also emphasized that the open source nature of the operating system will contribute greatly to its security by allowing constant developer feedback.

At a December 2010 press conference, Google claimed that Chrome OS would be the most secure consumer operating system due in part to a verified boot ability, in which the initial boot code, stored in read-only memory, checks for system compromises.

Chrome OS includes the Chrome Shell, or "crosh", which documents minimal functionality such as ping and SSH at crosh start-up.

Shell Access

In developer mode, a full-featured bash shell (which is supposed to be used for development purposes) can be opened via VT-2, and is also accessible using the crosh command shell. To access full privileges in shell (e.g. sudo) a root password is requested. For some time the default was "chronos" in Chrome OS and "facepunch" in Chrome OS Vanilla and later the default was empty, and instructions on updating it were displayed at each login.

Open Source

Chrome OS is partially developed under the open source Chrome OS project. As with other open source projects, developers can modify the code from Chrome OS and build their own versions, whereas Chrome OS code is only supported by Google and its partners and only runs on hardware designed for the purpose. Unlike Chrome OS, Chrome OS is automatically updated to the latest version.

Chrome OS on windows

On Windows 8 exceptions allow the default desktop web browser to offer a variant that can run inside its full-screen "Metro" shell and access features such as the Share charm, without necessarily needing to be written with Windows Runtime. Chrome's "Windows 8 mode" was previously a tablet-optimized version of the standard Chrome interface. In October 2013, the mode was changed on Developer channel to offer a variant of the Chrome OS desktop.

Design

Early in the project, Google provided publicly many details of the Chrome OS's design goals and direction, although the company has not followed up with a technical description of the completed operating system.

User Interface

Design goals for Chrome OS's user interface included using minimal screen space by combining applications and standard Web pages into a single tab strip, rather than separating the two. Designers considered a reduced window management scheme that would operate only in full-screen mode. Secondary tasks would be handled with "panels": floating windows that dock to the bottom of the screen for tasks like chat and music players. Split screens were also under consideration for viewing two pieces of content side-by-side. Chrome OS would follow the Chrome browser's practice of leveraging HTML5's offline modes, background processing, and notifications. Designers proposed using search and pinned tabs as a way to quickly locate and access applications.



New window manager

On April 10, 2012, a new build of Chrome OS offered a choice between the original full-screen window interface and overlapping, re-sizable windows, such as found on Microsoft Windows and Apple's macOS. The feature was implemented through the Ash window manager, which runs atop the Aura hardware-accelerated graphics engine. The April 2012 upgrade also included the ability to display smaller, overlapping browser windows, each with its own translucent tabs, browser tabs that can be "torn" and dragged to new positions or merged with another tab strip, and a mouse-enabled shortcut list across the bottom of the screen. One icon on the task bar shows a list of installed applications and bookmarks. Writing in CNET, Stephen Shankland argued that with overlapping windows, "Google is anchoring itself into the past" as both iOS and Microsoft's Metro interface are largely or entirely full-screen. Even so, "Chrome OS already is different enough that it's best to preserve any familiarity that can be preserved".

Printing

Google Cloud Print is a Google service that helps any application on any device to print on supported printers. While the cloud provides virtually any connected device with information access, the task of "developing and maintaining print subsystems for every combination of hardware and operating system—from desktops to netbooks to mobile devices—simply isn't feasible." The cloud service requires installation of a piece of software called proxy, as part of the Chrome OS. The proxy registers the printer with the service, manages the print jobs, provides the printer driver functionality, and gives status alerts for each job.

In 2016, Google included "Native CUPS Support" in Chrome OS as an experimental feature that may eventually become an official feature. With CUPS support turned on, it becomes possible to use most USB printers even if they do not support Google Cloud Print.

Link handling

Chrome OS was designed with the intention of storing user documents and files on remote servers. Both Chrome OS and the Chrome browser may introduce difficulties to end users when handling specific file types offline; for example, when opening an image or document residing on a local storage device, it may be unclear whether and which specific Web application should be automatically opened for viewing, or the handling should be performed by a traditional application acting as a preview utility. Matthew Papakipos, Chrome OS engineering director, noted in 2010 that Windows developers have faced the same fundamental problem: "Quicktime is fighting with Windows Media Player, which is fighting with Chrome."

Release channels and updates

Chrome OS uses the same release system as Google Chrome: there are three distinct channels: Stable, Beta, and Developer preview (called the "Dev" channel). The stable channel is updated with features and fixes that have been thoroughly tested in the Beta channel, and the Beta channel is updated approximately once a month with stable and complete features from the Developer channel. New ideas get tested in the Developer channel, which can be very unstable at times. A fourth canary channel was confirmed to exist by Google Developer Francois Beaufort and hacker Kenny Strawn, by entering the Chrome OS shell in developer mode, typing the command shell to access the bash shell, and finally entering the command update\_engine\_client -channel canary-channel -update. It is possible to return to verified boot mode after entering the canary channel, but the channel updater disappears and the only way to return to another channel is using the "powerwash" factory reset.

Reception

At its debut, Chrome OS was viewed as a competitor to Microsoft, both directly to Microsoft Windows and indirectly the company's word processing and spreadsheet applications—the latter through Chrome OS's reliance on cloud computing. But Chrome OS engineering director Matthew Papakipos argued that the two operating systems would not fully overlap in functionality because Chrome OS is intended for netbooks, which lack the computational power to run a resource-intensive program like Adobe Photoshop.

Some observers claimed that other operating systems already filled the niche that Chrome OS was aiming for, with the added advantage of supporting native applications in addition to a browser. Tony Bradley of PC World wrote in November 2009:

“ We can already do most, if not all, of what Chrome OS promises to deliver. Using a Windows 7 or Linux-based netbook, users can simply not install anything but a web browser and connect to the vast array of Google products and other web-based services and applications. Netbooks have been successful at capturing the low-end PC market, and they provide a web-centric computing experience today. I am not sure why we should get excited that a year from now we'll be able to do the same thing, but locked into doing it from the fourth-place web browser.”

After this 2009 statement Chrome browser rose to become the number one browser used worldwide.

By 2016, Chromebooks had become the most popular computer in the US K–12 education market.

Relationship to Android

Google's offering of two open source operating systems, Android and Chrome OS, has drawn some criticism despite the similarity between this situation and that of Apple Inc's two operating systems, macOS and iOS. Steve Ballmer, Microsoft CEO at the time, accused Google of not being able to make up its mind. Steven Levy wrote that "the dissonance between the two systems was apparent" at Google I/O 2011. The event featured a daily press conference in which each team leader, Android's Andy Rubin and Chrome's Sundar Pichai, "unconvincingly tried to explain why the systems weren't competitive." Google co-founder Sergey Brin addressed the question by saying that owning two promising operating systems was "a problem that most companies would love to face". Brin suggested that the two operating systems "will likely converge over time." The speculation over convergence increased in March 2013 when Chrome OS chief Pichai replaced Rubin as the senior vice president in charge of Android, thereby putting Pichai in charge of both.

The relationship between Android and Chrome OS became more substantial at Google I/O 2014, where developers demonstrated native Android software running on Chrome OS through a Native Client based runtime. In September 2014, Google introduced a beta version of the App Runtime for Chrome (ARC), which allows selected Android applications to be used on Chrome OS, using a Native Client-based environment that provides the platforms necessary to run Android software. Android applications do not require any modifications to run on Chrome OS, but may be modified to better support a mouse and keyboard environment. At its introduction, Chrome OS support was only available for selected Android applications. In October 2015, The Wall Street Journal reported that Chrome OS would be folded into Android so that a single OS would result by 2017. The resulting OS will be Android, but it will be expanded to run on laptops. Google responded that while the company has "been working on ways to bring together the best of both operating systems, there's no plan to phase out Chrome OS."

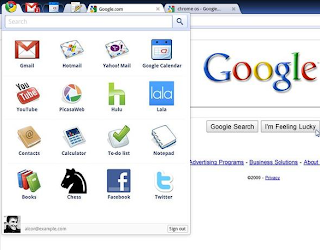
In 2016, Google introduced the ability to run Android apps on supported Chrome OS devices, with access to the entire Google Play Store. The previous Native Client-based solution was dropped in favor of a container containing Android's frameworks and dependencies (initially based on Android Marshmallow), which allows Android apps to have direct access to the Chrome OS platform, and allow the OS to interact with Android contracts such as sharing. Engineering director Zelidrag Hornung explained that ARC had been scrapped due to its limitations, including its incompatibility with the Android Native Development Toolkit (NDK), and that it was unable to pass Google's own compatibility test suite.

Installation

While Chrome OS runs on specially optimized hardware, it comes preinstalled. There is a simple setup procedure, however. It’s recommended that you sign in using a Google account. If you'd prefer, you can opt for the Guest mode. Guest mode in Chrome OS has the Chrome browser's trackless browsing mode, called Incognito. Incognito prevents guest users from leaving any traces of their session, as well as keeping them from making any changes to your apps and other settings.

Interface

The user interface of the Chrome OS simply resembles a web browser. You can run an application in one tab and browse the web in another. There's an actual desktop that looks a bit cribbed from Windows 7, with Chrome browser pinned to the far left of the Launcher, and other apps pinned right next to it. The browser also has a set of ‘Application Tabs’ which are pinned in the web-browser where user can access popular Google apps and services Google Talk, Gmail, Picasa, YouTube, Google+, Google docs which are pre-loaded and just a click away if you wish to use them. User has the option to add more. Besides the Application Tab, you will find the chrome menu. Other applications like contacts and calculator can be accessed from this menu. Activities like chatting and file handling and sharing are accomplished through Panels. For instance, if the user wants to view the contents of a USB device, he plugs it in the machine and he will see the contents in the pop-up panels.



The lower-right corner shows your Google account avatar to indicate who's logged in. Click the avatar to show shutdown options and reveal more information and settings.

Features

•    100GB Google Drive.  
•    Based on Linux: supports multiple workspaces.  
•    Auto updating and translation.  
•    Supports PDF, PPT, DOC, ZIP, XLS and RAR, and PPTX.   
•    Printing with Google Cloud Print. Cloud Print does now come with access to

FedEx stores in the United States, which is a nice improvement for remote

printing.   
•    Cookies, image management, JavaScript, plug-ins, pop-ups, location information,

and notifications can be adjusted.   
•    A new feature: "verified boot." Chrome OS will check its own integrity when

booting, and if it detects any changes, it will allow user to restore a last-known

good configuration.  
•    Supports external Memory Devices. Ex. SD Card.  
•    Explores external devices in Chrome Browser itself.  
•    A media player is integrated enabling users to play back MP3s, view JPEGs, and

handle other multimedia files while offline. Embedded Video Player is provided as

a feature of HTML 5.  
•    Chatting feature is readily available.  
•    V8 JavaScript engine is used to increase browser speed.  
•    Developer mode is included with a switch mode button. It allows developer to

install their own builds of Chrome OS  
•    File sharing is easier. If user wants to send a file from a USB drive to a friend with

whom he is chatting, he can simply drag and drop the file between the two panels

(the chat and the USB panel).





**Architectural Overview & Performance**

Google Chrome OS is not a conventional operating system that you can download or buy on a disc and install. As a consumer, the way you will get Chrome OS is by buying a Chrome-book that has Chrome OS installed by the OEM. However, one can download the Chrome OS which requires just 1GB disk space, 256 MB RAM and supports most of today's graphics cards and support both Intel, AMD processors.

Cr-48 prototype hardware

Google announced the Cr-48 laptop, a reference hardware design to test the Chrome OS operating system. The laptop's design broke convention by replacing the caps lock key with a dedicated search key and the default settings for the hot keys. The Cr-48 was intended for testing only, not retail sales. To test it, they used a high-powered Lenovo T400 laptop, running on an Intel Core 2 Duo T9400 at 2.53GHz, with 3GB of RAM.

The Cr-48 notebooks have additional unused hardware components, including a Bluetooth 2.1 controller. The USB port only acts as a keyboard, mouse, Ethernet adapter, or USB storage port and will not work as a printer port as there is no print stack on the operating system.

Commercial hardware

Chrome OS is initially intended for secondary devices like net books, called the Chrome-books. Currently only two Hardware companies ACER and SAMSUNG are authorized to design this Chrome-book. Chrome OS is hardware dependent currently supports x86-based PCs and devices built around the ARM processor. Chrome-books do not have HDDs but instead they use flash-based memory solutions or solid-state drives (SSD) for that.

On May 29, 2012, Google and Samsung introduced the Second-Generation Chrome-book and a new compact desktop Chromebox, which was said to be a clone of the Apple Mac Mini.

Hardware-specific developer information

|  |  |  |  |
| --- | --- | --- | --- |
| **Release date** | **Manufacturer** | **Model** | **Project code name** |
| December 2010 | Google | Cr-48 | Mario |
| June 2011 | Samsung | Series 5 Chromebook | Alex |
| July 2011 | Acer | AC700 Chromebook | ZGB |
| May 2012 | Samsung | Series 3 Chromebox | Stumpy |
| May 2012 | Samsung | Series 5 550 Chromebook | Lumpy |
| October 2012 | Samsung | ARM Chromebook | Snow (aka Daisy) |
| November 2012 | Acer | C7 Chromebook | Parrot |

Google is currently working with manufacturers to elaborate a new set of netbook hardware which comply with their speed and security needs. Also, Google requests the hardware partners for the netbooks to be made a bit larger to accommodate a standard size keyboard and to have better mouse pads. Later Google Chrome OS will also be available on desktops, serving as an ideal machine for a public Internet terminal, according to Google.

Advantages

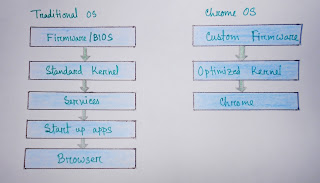
Chrome OS does a lot of things and its almost custom made for regular internet users but all that can be done on other current OS too, so why then should to switch to Chrome OS is because of the below reasons



Chrome is intended to provide the core features of an OS which is speed, simplicity and security

Speed

Chrome OS tends to diminish the irrelevant process time such as hardware detection, BIOS, OS Primitives etc. Chrome offers incredibly quick boot time, and is all customized for quick access. Google claims that even the fastest computer takes about 25 seconds to boot. Booting on a Chrome OS device takes hardly 7 seconds to get the user to the log-in screen. It is just like Television, Flip a switch and it is ON.

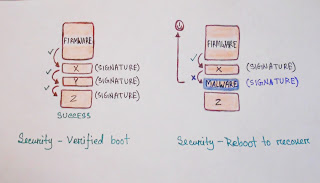


Simplicity

Just a browser with tabs  
-    Requires Google ID to log in  
-    Customizable as per users requirement   
-    For any application User is not supposed to buy and install any app on her PC instead just get subscription to suitable web application and consume it.

Security

1. Small list of verified programs : signed and verified before each use
2. File System is Locked down :
3. Read only Root File System in EEPROM
4. Self Automatic updates for entire OS
5. All apps in Chrome OS are web apps which are sandboxed. This means that they can’t interfere with each other and your device. Therefore the chances of getting a virus are minimal.
6. The root system is read only. All user data is encrypted and fully secure. On boot up, the OS checks itself and incase any improper functionality is found; the OS automatically downloads patches and reinstalls them.
7. Data is stored in cloud - As such, computers running Chrome OS will be very dependent on a solid internet connection to function optimally. Since data is stored in the cloud, you’ll be able to access your data remotely in a process Google is calling “chromoting”.
8. Data is always synced with cloud to avoid data loss
9. Digital Signatures are provided for every firmware update
10. If malware found System recovers itself to clean image

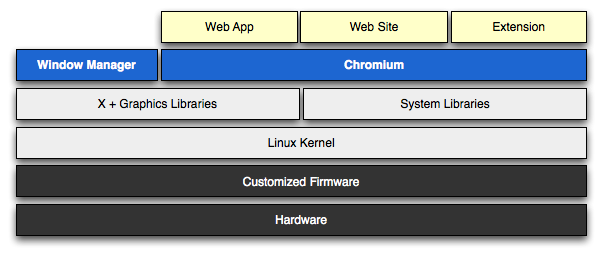




Software architecture

Chrome OS consists of three major components:

* The Chrome-based browser and the window manager
* System-level software and user-land services: the kernel, drivers, connection manager, and so on
* Firmware

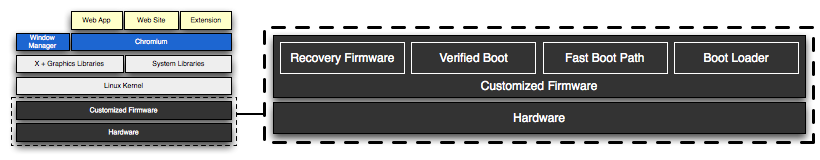


Firmware

The firmware plays a key part to make booting the OS faster and more secure. To achieve this goal we are removing unnecessary components and adding support for verifying each step in the boot process. We are also adding support for system recovery into the firmware itself. We can avoid the complexity that's in most PC firmware because we don't have to be backwards compatible with a large amount of legacy hardware. For example, we don't have to probe for floppy drives.

Our firmware will implement the following functionality:

* [**System recovery**](https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery)**:** The recovery firmware can re-install Chrome OS in the event that the system has become corrupt or compromised.
* [**Verified boot**](https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot)**:**  Each time the system boots, Chrome OS verifies that the firmware, kernel, and system image have not been tampered with or become corrupt. This process starts in the firmware.
* [**Fast boot**](https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery)**:** We have improved boot performance by removing a lot of complexity that is normally found in PC firmware.



Firmware Boot and Recovery

* The layout and structure of firmware for Chrome OS is designed for security, recovery and development.
* All firmware will contain a recovery code path, which will restore the machine to its original Chrome OS state. This recovery code path will be initiated either when any chain in the boot path is not verified or when a user manually triggers recovery mode, likely via an explicit recovery button on the device.
* Chrome OS wants to support developers as well. Developers are provided with a means of running alternate software. In the alternate boot paths, the user is notified that they are not running a boot path provided as part of Chrome OS.
* The boot and recovery procedures outlined will be implemented and required for all Chrome OS platforms regardless of architecture (ARM/Intel/etc...).

**Problems faced in developing the OS**

Firmware

1. Incomplete update: An update of the firmware is interrupted. This leaves the portion of the firmware which was being updated in an unknown or corrupt state. For example, if the update is interrupted after a firmware block is erased but before it is reprogrammed, that block is empty.
2. Attack: An attacker compromises the software and is able to reprogram the firmware. For example, an exploit of an unpatched kernel vulnerability. In this case, both the main and backup firmware may be compromised.
3. Corruption: The EEPROM holding the firmware becomes corrupted in the sectors containing writable/updatable firmware.

Software

1. Incomplete update: An update of the software on the drive is interrupted. This leaves the rootfs partition in an unknown state.
2. Attack: An attacker compromises the software and is able to rewrite the data on the drive (rootfs or partition table).
3. Malicious user: A malicious user installs developer mode onto the device, leaving behind a trojan, then returns the device.
4. Corruption: The drive becomes corrupted in the partition table or rootfs partition.
5. Crash: Device crashes on boot due to bad software. For example, the device is updated with the wrong image. This prevents the normal autoupdate process from running.

Other problem

1. An attacker with physical access to the device opens the device and alters its innards. This includes:
   * Jumpering the EEPROM to gain write access to the read-only portion.
   * Replacing the EEPROM.
   * Otherwise altering the circuit board (adding piggyback chips, etc).
2. Exploits involving the normally-writable data partition.
   * Changing bookmarks to point to sites which download HTML5 malware, etc.
   * Malformed bookmarks file or image which causes a buffer overrun when parsed
   * Changing the preferred wireless network to a malicious one which logs/alters packets
3. Exploits involving other processors (embedded controller, modem processor, etc.)

## Design decisions

### Boot needs to start securely

In order to attempt a secure boot, the initial boot stub needs to perform a minimum level of initialization to verify the next piece of boot code before handing off to that code.

To prevent accidental or intentional corruption of the known-good boot stub, this code must be in a portion of memory which is not field-writable. Many EEPROM devices have an external pin (WP) which can be pulled low to write protect the upper portion of the EEPROM. This has a number of benefits:

* Devices are writable at time of manufacture (as opposed to true ROMs, which are fixed at time of ROM manufacture).
* Devices can be made writable for firmware development by simple hardware modification.
* Both readable and read-only ROM are provided by a single device. Simpler board design, fewer parts, lower cost.
* Any attacker who can open the case and modify the hardware to write to the protected upper portion of ROM could also simply replace a true ROM with a reprogrammed part, so this isn't significantly less secure than a true ROM.

On ARM platforms, the initial boot ROM may be in the same package as the processor. This is even more secure compared to a separate EEPROM.

### Writable firmware should have a backup

To protect against a failed firmware update, the writable portion of the firmware (responsible for doing the remainder of chipset and storage setup and then bootstrapping the kernel off the storage device) should exist in two copies. In the event the first copy is corrupt, the device can boot normally off the second copy. This is similar to the design for the file system, which has two copies of the root partition.

### Recovery firmware must be read-only

Recovery firmware must be able to take over the boot process if the boot stub determines that the normal writable firmware is corrupt, or if the user manually boots the device into recovery mode.

To prevent the recovery firmware from being corrupted by a firmware update or a software-based attack, it must be in the same read-only portion of the EEPROM as the boot stub.

### Recovery firmware does not need to access the network

The recovery process should not require firmware-level network access by the device being recovered.  The recovery procedure can involve a second computer, which is used to create recovery media (for example, a USB drive or SD card). It is assumed that second computer has network access.

This simplifies the recovery process, since the recovery firmware only needs to bring up enough of the system to bootstrap a Linux image from local storage. That image can then take care of reflashing the EEPROM and reimaging.

It is not necessary to implement a full network stack with WiFi configuration in the recovery firmware to support PXE boot. PXE boot introduces a number of complications:

* Need to initialize more hardware to bring up wireless, keyboard, etc.
* Need to implement a full network stack.
* Makes recovery an interactive process, including the user entering their SSID and WPA key, which the user may not know.
* Unlikely to work for public WiFi access points; these often redirect http access to a login screen, navigation of which which would necessitate a full browser in the recovery firmware.
* Unlikely to work for cellular networks (3G/4G/etc...), if that requires more complicated drivers or configuration.

All of these issues would need to be resolved, and the resulting firmware must be correct at the time the device ships, because recovery mode firmware can't be updated in the field. It is unacceptable to ship a mostly-working PXE solution, assuming that the user can fall back on a second computer in the event PXE recovery fails. The only time the user would discover PXE recovery didn't work is when the user is relying on it to repair their computer.

More information on wireless network boot is here: <http://etherboot.org/wiki/wirelessboot>.

### Recovery firmware should tell the user how to recover

If the recovery firmware finds a USB drive/SD card with a good recovery image on it, it should boot it and use that to recover. The software in that recovery image will have its own user interface to guide the user through recovery.

If the recovery firmware does not find a good recovery image, it needs to tell the user how to use a second computer to build that recovery image.

The preferred way to do this is to initialize the screen and show recovery instructions to the user, including a URL to go to in that second computer's web browser. Note that recovery instructions need to be displayed in the correct language for the user.

It is desirable for the recovery instructions and/or recovery URL to include a code for the device model. This allows the destination website to:

* Provide the proper recovery image for that device model.
* Describe the recovery procedure specific to that device model. For example, if the device has a SD card but no USB port, the recovery procedure would indicate that a SD card is necessary, and would not discuss the possibility of recovering using USB.
* Display graphics appropriate for the device model. For example, showing the location of the USB or SD card port.

### Users must be able to manually trigger recovery mode

If the writable firmware and/or rootfs have valid signatures but don't work (for example, the user somehow managed to get an ARM kernel on an x86 device), the user needs to be able to force recovery mode to run.

This can be done by having a physical reset button somewhere on the device. When this button is pressed during power-on, the device goes straight to the recovery firmware without even looking at the writable firmware or file system.

Some options for the recovery button:

* It could be a button attached to a GPIO on the main processor. In this case, the boot stub would initialize the GPIO and read its state at boot time.
* It could be a button attached to a subprocessor such as the Embedded Controller (EC). In this case, the boot stub would need to request the button state from the EC at boot time.
* It could be one of the keys on the keyboard, though this creates the undesirable possibility of accidentally entering recovery mode.
  + This is undesirable if the only interface to the keyboard is USB, because USB firmware is more complex and the USB hardware interface can take a significant amount of time to initialize.
  + Some devices use a subprocessor to read the keyboard. In this case, initiating recovery mode is much like the previous option.
  + The keyboard could bring out the press-state of one of its keys to a separate I/O line, which could be attached to a GPIO on the processor or to a subprocessor.

Since the recoverability of Chrome OS is one of its key features, we seek to have a dedicated "recovery mode" button.

### Support developers / l33t

### Users installing their own software

To provide a relatively trustable boot for the majority of users, we need to control all the read-only firmware at the beginning of the boot process.

To support developers, at some point during the boot process, we need to hand off to code self-signed by someone else.

The initial devices will allow hand off at the point the kernel is loaded and its embedded signature is checked.  This can produce one of three results:

* Trusted kernel: The kernel has a valid signature, and the signing key is known to the firmware.  This is the normal case for production devices.
* Developer kernel: The kernel has a valid signature, but the key used to sign the kernel is not known to the firmware.  This is the case when a developer builds and self-signs their own kernel.
* Corrupt kernel: The kernel fails its signature check.

Once the chain of trust departs from the standard Chrome OS boot chain, we need to indicate this clearly to the user of the device. This prevents malicious attackers from giving users a modified version of Chrome OS without the user knowing. We likely will need to show a warning screen which includes the following elements:

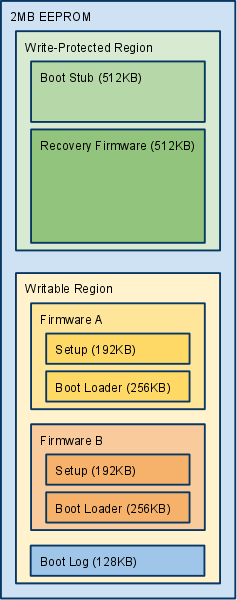
* A warning that the standard image of Chrome OS is not running
* A means of reverting back to the standard Chrome OS image
* A means of allowing the user/developer to proceed down the "untrusted" path

It is desirable for the warning screen to have a timeout, so that Chrome OS devices with developer images can be used in unattended applications (for example, as a media server).  The timeout should be sufficiently long that a user can read and respond to it - for example, at least 30 seconds.

Since language settings will not be available at this stage of the boot process, any messaging will likely need to be internationalized and displayed in all possible languages.

## EEPROM map

Here is an early guess at sizes and layout of the EEPROM. These sizes may change as we progress with implementation.



## Firmware block descriptions

### Boot stub

Must be at the top of EEPROM, since most processors jump to the top of memory (0xFFFFFFF0) after internal initialization.

Contains the "root key" - the official public key used to verify the signature of the next stage of firmware. The private key is not contained on the device, and must be protected from all unauthorized access. To reduce exposure of the private root key, the private root key will be used to sign a second date-limited or numbered key stored in the rewritable firmware, which is then used to sign that firmware. Validation of the date or key number could be done via a TPM module.

**Pseudocode**

1. Initialize processor and RAM (and implicitly, those parts of the north bridge necessary to initialize RAM), using conservative timings.
2. If user-forced recovery mode, skip to attempt loading recovery firmware. (The recovery button asserts an I/O line that can be measured by the firmware.)
3. Check the non-volatile register or memory region for a recovery mode cookie value.  If this is set, some later stage of the boot process must have failed and requested recovery mode, so skip to attempt loading recovery firmware.
4. Attempt loading Firmware A.
   1. Copy Firmware A to RAM.
   2. Verify signature of Firmware A using public key stored in boot stub.
   3. If signature is valid, jump to Firmware A Setup.
5. Firmware A is bad. Repeat for Firmware B.
6. Both A and B are bad. Attempt loading recovery firmware to RAM and verify signature. If valid, jump to recovery firmware.
7. All recovery options have been exhausted. Catch fire / emit POST code / etc.

### 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 Chrome 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.

### Boot log

A boot log will be stored at the bottom of the writable section of firmware. This will store the following types of events:

* Each time recovery firmware is run, with information on what triggered it (manual, bad firmware, bad root filesystem, etc.).
* When a recovery completes, including which actions were taken.

It does not store information on successful boots. This removes the need to support EEPROM writing in the normal boot process, and conserves space for log entries dealing with real errors.

The boot log can be uploaded as part of the autoupdate process. It can then be cleared and reused for new log entries.

### 

**Recovery firmware**

This firmware attempts to recover from bad firmware or rootfs by loading a recovery image off an external storage device.

**Pseudocode**

1. Write a log entry to the boot log indicating why recovery mode was activated (manual, bad writable firmware, user triggers recovery mode, etc.)
2. Initialize enough of the chipset to be able to access a storage device (USB drive, SD card, etc.)
3. Check for an inserted storage device.  If no storage device is present, skip to displaying recovery instructions.
4. Verify the signature of the recovery image, using a public key stored in the recovery firmware.  If the signature is invalid, skip to displaying recovery instructions.
5. Load and run the recovery image.
6. If we're still here, either no storage device was inserted, or the storage device did not contain a valid image.  Display recovery instructions:
   1. Initialize the display.
   2. Display instructions.  These instructions must be internationalized and rendered in multiple languages.  The instructions must include:
      * A warning that the computer is in recovery mode
      * How to obtain a trusted recovery image (for example, instructions to go to a second computer and browse to a specified URL for further instructions)
      * An indication if an already-inserted storage device does not contain a valid recovery image (sorry, try again).
7. Wait for a storage device to be inserted, then go back to step 3.

Most of the recovery work is left to the recovery image loaded from the storage device. This allows for publishing updated recovery images and instructions. Because the recovery firmware is etched in stone (well, as electrons in floating gates) at the time the device is launched, it needs to be as simple and robust as possible.

## 

**Recovery image**

This is the software loaded onto a storage device (USB drive, SD card, etc.) which does the bulk of the recovery work.

The recovery image will be available for download.

We will need to provide a downloadable installer for users to use to install the recovery image on the storage device. This installer should be supplied for all popular operating systems (Chrome OS, Windows, Mac OS X, Ubuntu Linux). The installer will:

* Prompt the user to select a destination for the recovery image
  + Ideally, the user will only be able to select destination devices appropriate for recovery of the laptop model they are attempting to recover.
  + Also, the user should only be able to select removable devices. We don't want them trashing their hard disk.
* Warn the user that this will erase whatever is on that destination, and prompt for confirmation
* Reformat the destination storage device
* Install the recovery image on the storage device
* Prompt the user to insert the storage device into the Chrome OS device

Ideally, the installer would be able to back up the current data on the destination device, before reformatting and writing the recovery image. When recovery is complete, the user could re-insert the destination device and have the original contents restored.

The recovery image should contain an entire clean copy of the firmware and rootfs. This way, a user can make a recovery device ahead of time. For example, if a user is going to be somewhere with poor connectivity, they could make a recovery USB drive at home and keep it in their bag. Alternately, the system could come with a recovery device (though users might end up reformatting it and filling it with images of kittens).

The recovery image on the storage device would do something like the following:

1. Initialize the display and tell the user what's going on. Reassure them that their Chrome OS system is trying to recover.
2. If developer firmware is detected, give the user a choice of "JUST FIX IT" or "Scary Settings for L33t H4x0rs". If the user picks the latter:
   1. Give the developer more control over each stage in the rest of the recovery.
   2. For example, a developer might not want to wipe and replace rootfs or stateful data, or might want to make a backup of that data.
3. Run system tests. It's possible that the boot problems are due to bad hardware. Now's the time to detect that.
   1. If hardware is sufficiently bad that the rest of recovery can't be run, show the user information on how to return it.
4. Verify both firmware A and firmware B are up-to-date and valid. Load known good firmware as needed.
5. Verify rootfs-A and rootfs-B are up-to-date and valid. Load known good rootfs images as needed.
6. Wipe the stateful data. Anything the user cares about should be in the cloud anyway. If the user's been hacked, wiping the data will put the device back in a known good state.
   1. Optionally, it can inform the user that data on the system will be erased.
   2. This message should emphasize that most data is saved to the cloud, so that the user is more likely to proceed with recovery.
   3. It is less desirable to give the user a choice whether or not to delete the stateful data. Most users would pick the less scary but also less secure option of NOT deleting data - but this leaves them vulnerable to persistent hacks on the data partition, such as manipulating their /etc/hosts file or bookmarks.
7. If recovery mode was manually triggered, ask the user why they're running it.
   1. Couldn't even log in; crash during boot
   2. Kept crashing after boot
   3. Worried my system was hacked
   4. Hey, what's this button do?
8. Write an entry to the boot log describing repairs performed.

### Recovery via Chrome OS

The recovery installer should run on a healthy Chrome OS system. That is, it should be able to download the recovery image, reformat the storage device, and copy the recovery image to it.

* This enables a healthy Chrome OS system to create its own recovery device in advance.
  + Perhaps we should periodically advise users to update their recovery device?
* It also enables one Chrome OS system to download a recovery image for a different model. (A user has a corrupt system, so they go to their friend's house and use the friend's Chrome OS system to make a recovery image).

### Making recovery images read-only

If the recovery image is inserted into a powered-on and hacked system, the hacked software could delete or corrupt it. This won't cause the recovery firmware to load that corrupt image, because the corrupt image will fail the signature check. It will be annoying to the user, who will need to reflash the storage device.

This is more of an issue if we want to include an internal recovery image (for example, on an internal storage device).

The SD standard specifies a physical write-protect notch for SD cards, similar to those on a 3.5" floppy disk. Unfortunately, the implementation of the write-protection is purely software, so pwned drivers can choose to ignore the write-protect detect signal.

Some USB drives have a write-protect switch on them. In this case, the protection is handled by the USB drive itself.

Some eMMC chips have a number of protection mechanisms including:

* an external LOCK# pin which can be used to make the device read-only
* a write-once bit which makes the device read-only until reboot

Since these chips come in sizes up to 2GB (~$10 at stores), they provide a possible place to store a recovery image. Some system architectures may be able to use the eMMC drive to hold the main firmware image also.

If we have an internal storage device, it's possible we could wire it so it's only enabled if the recovery button is pressed at boot time. For example, we could use that button to latch up a circuit which powers that device, so that it will remain powered during that boot only.

### Using recovery mode to load developer mode firmware/software

If the rootfs image on the SD card is signed by someone else, the recovery image will display a screen similar to the Developer Mode screen.  Instructions on this screen should include:

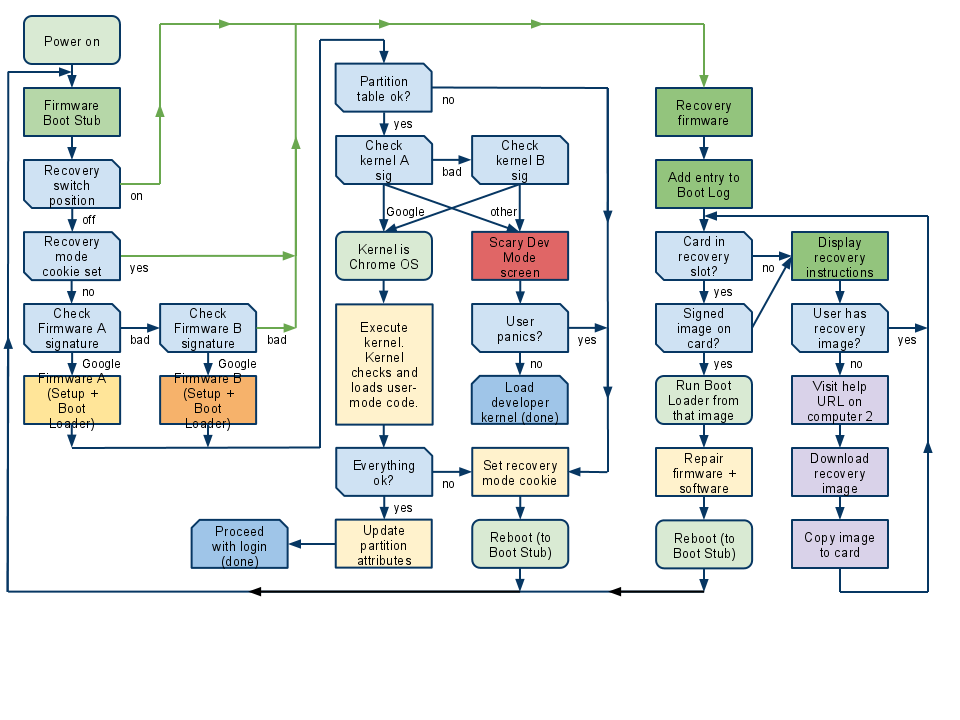
* Notice that the recovery image is signed by someone else
* How to obtain a trusted recovery mode image
* Instructions on installing a trusted recovery image
* A means of allowing the user/developer to proceed down the "untrusted" path

Note that this screen must also be internationalized / rendered in multiple languages. This is potentially much easier, since by the time we're doing this the recovery image has booted a full OS with GUI.

## Boot flowchart

The three main downward flows in this chart show:

* Start of boot in read-only firmware
* Normal continuation of boot, loading kernel from internal drive
* Recovery mode

[](https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery/ChromeOSFirmwareBootandRecoveryPUBLISHED.png?attredirects=0)**Other Points**

### Verification of the rest of the rootfs

The firmware boot process above describes a way to verify all code from the start of boot through hand off to the kernel.

The kernel is responsible for verifying the rest of the data in the rootfs - for example, user-mode drivers. If the kernel determines that the rootfs has been compromised, it can force recovery mode to run by setting the recovery mode cookie in a non-volatile register and rebooting into recovery mode.

Alternately, a kernel which determines its rootfs is corrupt can commit suicide by marking its partition as inactive and then rebooting. The next boot will do one of the following:

* If the other root partition is good, the boot loader will run the other root partition.
* If neither root partition is good, the boot loader will trigger the recovery mode firmware, which can put the rootfs back in a known state.

### When to verify the boot process

We should develop under the initial assumption that we can verify every boot, since this provides the most security.

Signature-checking code does impact startup time. However, this is likely not an issue for the firmware (< 1 MB) and kernel (< 10 MB) than it is for the remainder of the rootfs. We assume that we will verify the entire firmware and kernel every boot.

### Handling returned systems

When a Chrome OS system is returned to the store, the store should check to make sure it still boots.

* If it boots normally, the store can follow their standard process for dealing with returns.
* If it displays the Scary Dev Mode screen, the store can either run recovery mode or refuse the return (in which case the original owner can go home and run recovery mode, then come back and returns the system again).
* If it fails to boot, it gets shipped back to the manufacturer.

Note that it's advisable for stores to run recovery mode on returned computers anyway, to put them back into a clean state and destroy any user data still on the device.

### Testing

The comprehensive test suite for Chrome OS should include tests of each stage of the verification process and the recovery procedure, including all decision points in the pseudocode/flowchart. This will require actual hardware for each firmware image being tested.

Testing should also attempt to write to the read-only portion of the EEPROM, to ensure that it is in fact not writable.

**Verified Boot**

* The Chrome OS team is implementing a verified boot solution that strives to ensure that users feel secure when logging into a Chrome 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 Chrome 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](https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery), [Verified Boot Crypto](https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-crypto), and [Verified Boot Data Structures](https://www.chromium.org/chromium-os/chromiumos-design-docs/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 Chrome-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.

## Goals of verified boot

The primary attacker in this model is an opportunistic attacker. This means that the attacker has accessed the system using:

* a remote vector, such as the Chrome-based browser or a browser plugin
* a local vector, such as booting to a USB drive and changing files (but **not** by replacing the write-protected firmware)

If we assume attackers access the system via a remote vector and bypass all run-time defenses, then they will have access to modify the root partition (kernel, modules, browser, ...), update read-write firmware, inject code into SMRAM, and so on. In addition, the attacker can now access any data in the currently-logged-in user's account such as locally stored media and website cookies. The attacker may collect passwords when typed by the user into the Chrome-based browser or the screenlocker.

An opportunistic *local* attacker will have a completely different level of access. Access will be achieved using a USB boot drive, or other out-of-band bootable material supported by the firmware. Once the system is running the attacker's operating system, she will be able to modify the root file system and encrypted user-data blobs. She won't have any visibility into the encrypted information but may copy it or modify it.

While Chrome OS does as much as possible to guard against such remote and local breaches, no software system is impervious to successful attacks. Therefore, it is important that the attacker cannot continue to "own" a machine through permanent, local changes. To that end, on boot, the firmware and other accessible regions of the system internals are verified to be in a known good state. If they aren't, then the firmware recovery process will be initiated (or the user can request permission to proceed, which would make sense in the case of a development install, for example).

The important factor to consider with the attackers considered above is that if an attacker gains access via the Chrome browser, they can presumably modify the Chrome browser's startup (or bookmarks or server-side settings) to re-attack the machine at next reboot. This is why it is important to be able to ensure that a safe recovery/reinstall is possible outside of what can be done by an attacker on the machine. (Obviously, this is no deterrent for a **determined** attacker willing to modify the system physically.)

## Getting to the kernel safely

As outlined in the Firmware Boot and Recovery document, verification will occur in several places. Initially, the small read-only stub code will compute a SHA-2 hash (either with internal code or using a provided SHA-2 accelerator) of the read-write portion of the firmware. An RSA signature of this hash will then be verified using a permanently stored public key (of, [ideally](http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf), 2048 bits or more).

The read-write firmware is then responsible for computing hashes of any other non-volatile memory and the kernel that will be executed. It will contain its own *subkey* and a list of cryptographic hashes for the data to be verified: kernel, initrd, master boot record, and so on. These additional hashes will be signed by the subkey so that they may be updated without requiring the write-protected key to be used for every update. (Note, the kernel+initrd signed hashes may be stored with the kernel+initrd on disk to avoid needing a firmware update when they change.)

Once we're in the kernel, we've successfully performed a verified boot.

## Extending verification from the kernel on upward

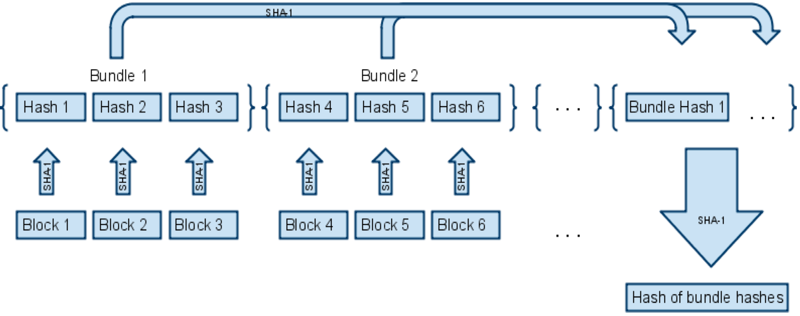
In general, once we're running, integrity measurements become less useful. We can ensure that the Chrome browser that we execute has not been tampered with, but we can't guarantee that the same attack that compromised it the first time won't compromise it a second time without updating.

To ensure that an update is possible, the executables, modules, and configuration files that allow the system to receive updates must be authentic and untampered with. Getting to that point requires network access and a running autoupdater. Given that Chrome OS keeps a very minimal root file system, it's easier to just verify everything on it.

While that sounds great in theory, in practice it is hard to guarantee an intact file system without paying the cost for upfront checks. If the read-write firmware were to verify the entire root partition before proceeding to boot, it would add at least 5 seconds to the boot-time on current netbooks. This delay is untenable.

Instead of performing full file system verification in advance, it can be done on demand from a verified kernel. A transparent block device will be layered between the run-time system and all running processes. It will be configured during kernel startup using either in-kernel code changes or from a firmware-verified initial ramdisk image. Each block that is accessed via the transparent block device layer will be checked against a cryptographic hash which is stored in a central collection of hashes for the verifiable block device.  This may be in a standalone partition or trailing the filesystem on the verifiable block device.

Initially, blocks will be 4KB in size. For a root file system of roughly ~75MB, there will be roughly 19,200 SHA-1 hashes. On current x86 and ARM based systems, computing the SHA-1 hash of 4KB takes between 0.2ms and 0.5ms. There will be additional overhead (TBD) incurred by accessing the correct block hash and comparing the cryptographic digests.  Once verified, blocks should live in the page cache above the block layer. This will mean that verification does not occur on every read event. To further amortize time-costs, the block hashes will be segmented into logical bundles of block hashes. Each bundle will be hashed. The subsequent list of bundle hashes will then be hashed. This layering can be repeated as needed to build a tree.  The final, single hash will be hard coded into the kernel or supplied through a device interface from a trusted initial RAM disk.

[](http://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/verified-boot/diag2.png?attredirects=0)

Note that SHA-1 is considered to be [unsafe after 2010 by NIST](http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html) for general use. The biggest risk here is that specific block collisions can be found and made such that they provide an alternate execution path. We could use any hashing algorithm supported by the Linux kernel in our implementation. SHA-1 is just a specific example.

## Known weaknesses of verified boot

While verified boot can ensure that the system image (i.e. firmware, kernel, root file system) are protected against tampering by attackers, it can't protect data that must inherently be modifiable by a running system. This includes user data, but also system-wide state such as system configuration (network, time zone, keyboard layout, etc.), cached data maintained by the system (VPD contents, metrics and crash reporting data, etc.). This state is generally kept on the writable stateful file system. In some cases, it is consumed by the boot process and may affect the behavior of the (verified) software. If an attacker manages to place malicious data on the stateful file system that will cause the verified code to "take a wrong turn", they may cause inadvertent side effects that may ultimately lead to the system getting exploited and thus defeating verified boot. The [Hardening against malicious stateful data](https://www.chromium.org/chromium-os/chromiumos-design-docs/hardening-against-malicious-stateful-data) document discusses details and mitigations.

## Mitigating potential bottlenecks

Loading hashes off the same disk as the data for each block would affect performance during a read. Right now, the plan is to read in signed bundles of hashes as blocks in that bundle range are accessed. Once a bundle is loaded into memory, it is kept there. If we assume that we're looking at something like 20k-hashes, then that will require around 400KB of memory. Even if the needed hashes grow to twice that, allocating 1MB of kernel memory doesn't seem too onerous. In addition, keeping the block hashes in memory will provide for easy linear addressing of the hashes since they will be in block-order.

## Handling updates

For Chrome OS, the autoupdater will update the collection of hashes specific to the partition it is updating.  In general, the complete collection of hashes for a specific partition will be generated by running a lightweight utility directly on the filesystem image. It will walk the origin block device and emit an image file that contains the properly formatted hash collection. In addition, it will emit the SHA-1 hash of the bundle hashes. This will be the authoritative hash that will need to be either signed or stored in a signed/verified location, such as in the kernel. The resulting file can either be appended to the filesystem image or stored in a standalone partition (hash partition).    On update, a direct difference of the new hash collection can be taken using bsdiff against the last versions.  However, it may be that more efficient difference generation approaches may be used as long as the end result is the same.

## The implementation

Post-firmware verified boot will most likely be implemented as a device mapper target. It will provide the transparent, verifying block layer. Initially, it will be assumed that there will be a verified initrd that can be used to set up the root partition using the dm device. A simple utility tool will be written that will directly hash a given block device and emit a compatible binary blob that contains the collection of hashes. It will take the format:

  hashblock\_1  . . . hashblock\_n  
  hashbundle\_1 . . . hashbundle\_m

This data will live either in its own partition or be appended to the verified partition (aligned on a block boundary). Its location, the hash algorithm used, and the hash of bundle hashes should be passed in as arguments to the device mapper setup process (either using dmsetup from an initrd or directly in the kernel).

## Other issues, ideas, and notes

* All verification performed after the kernel is running should be independent of the firmware verification. This allows for developers to run their own builds as well as for the boot-time verification to be compatible with a TCG-style boot.
* The partition table should be validated to some extent by the read-write firmware, but if there are any kernel/firmware partition parsing bugs, we may be able to catch them with audits as well.
* Verified Boot can play nicely with the TPM. Once a signed kernel is up, it can initialize the TPM's PCR registers and use those for measurement tracking. In particular, we have the option of using it (Linux-IMA style) to perform disk encryption key protection and possibly even other pre-login state protection where the key becomes available only if we've booted without modification.
* The hash partition will be subject to replay attack unless the kernel version that pairs with it is included in the file and the kernel is upgraded when the hashes are. However, the kernel+initrd will suffer the same attack for the next level up. Avoiding replay/rollback attacks is non-trivial since the firmware can't guarantee the local clock is not changed. A local TPM tickstamp blob would need to be included in the signed hash payloads to solve the problem to some degree. If autoupdates were customized per-download, this may be possible, but at present, this is not planned.
* Key management is of utmost importance for the key used to sign the read-only firmware. That key should only be used on the R/W firmware which should be updated much less frequently than the rest of the system. If possible, this key should never be exposed on a network-enabled machine, but that is out of scope for this design.
* Having a fully tamper-evident root file system means that if desired, a manifest of service-specific public keys/certificates could be stored on the root partition. These keys could then be used to verify the authenticity and integrity any data stored on the stateful partition that was signed by a remote server (Google or another provider).
* There is no plan to support any remote inspection of whether a Chrome OS installation is using a 'verified boot' or a development version.
* The autoupdate process currently has a file-centric view. This means that it could be possible for file system layout to diverge across machines. If this is the case, block hashes may still be used, but a more file-centric view may be needed. If the updater moves towards file system image differencing, this design will work as is.

## Attack cases

This section only discusses the current threat model, but many of the attacks can be generalized to other attack vectors. In addition, these scenarios ignore all other attack mitigation techniques not included in the document above. In reality, various system-level and Chrome-level defenses should aid in making run-time attacks difficult and unreliable.

*Vector*: Opportunistic local attacker with a USB stick or bootable SD card.  
*Scenario*: Attacker boots the system off of an external boot device. The attacker then changes files and copies the entire system.  
*Coverage*: Verified Boot will detect this tampering. Encrypted user data will still be protected.  
*Exposure:*None. User will need to recover their system.

*Scenario*: Attacker boots the system off of an external boot device and leaves the system running a "fake" Chrome OS to phish user data.  
*Coverage*: Verified Boot will not impact this *unless*the user reboots the system before logging in.  
*Exposure:*None to complete depending on if the user reboots prior to logging in. If the user left the machine at the screenlocker, a fake screenlocker could be used in the phishing OS since it is unlikely a user will reboot before unlocking. This may be addressed in the future with clever authentication use (PCR+TPM, ?). However, a paranoid user that left their machine in an unsafe place may just want to reboot to be safe.

*Vector*: Determined local attacker with a USB drive and a toolkit  
*Scenario*: Attacker opens the system up and enables the write-pin on the write-protected firmware. The attacker then boots the system off of an external boot device. The root file system is changed along with the formerly write-protected and read-write firmware.  
*Coverage*: Verified Boot will operate normally and will not detect any variance.  
*Exposure:*Complete; a determined attacker that will physically modify the machine cannot be easily stopped. They may also install hardware keyboard sniffers, microphones, cameras, etc.

*Vector*: Chrome or a Chrome plugin  
*Scenario*: No superuser privileges are gained, but the attacker can modify Chrome data. The attacker changes bookmarks, starting pages, marks their pages as ok for popups, and disables safe browsing. In addition, cookies and stored passwords are harvested and posted to a remote server.  
*Coverage*: Verified Boot has no impact.  
*Exposure:*Only the initially compromised user is exposed.

*Scenario*: Attacker gains superuser privileges. The attacker remounts root partition read-write directly. The attacker then replaces /usr/bin/chromeos-chrome with their own build of Chrome that includes malware/illegal ad revenue and password/credit card sniffing.  
*Coverage*: Verified Boot will detect after the next reboot (not after a suspend to RAM).  
*Exposure:*Until reboot, any user that logs in is exposed (password, cookies, encrypted data).

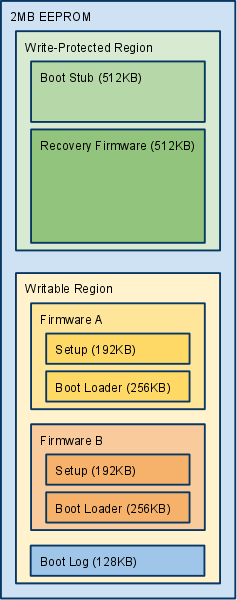
*Scenario*: Attacker gains superuser privileges. The attacker remounts root partition read-write directly. The attacker then adds a kernel module in the form of a rootkit for the system to load on next reboot.  
*Coverage*: Verified Boot will detect after the next reboot.  
*Exposure:*Until reboot, any user that logs in is exposed (password, cookies, encrypted data).

*Scenario*: Attacker gains superuser privileges. The attacker then modifies the encrypted file system metadata which exploits a file system bug in the kernel on next login.  
*Coverage*: Verified Boot has no direct impact. However, if the autoupdater runs before next login, the vulnerability may be patched.  
*Exposure*: On next login, the tampered with encrypted file system metadata attack will trigger.

*Scenario*: Attacker gains superuser privileges. The attacker then replaces the hash-partition with an older version and replaces system image with one that has known vulnerabilities (which may be easier to exploit reliably than the vector used for attack). The attacker will then change the current user's configuration to auto-open an attack URL to re-exploit the system immediately after reboot. If the attacker can gain superuser privileges repeatedly, then it will be difficult for autoupdate to repair.  
*Coverage*: Verified Boot will not be able to detect hash-partition replay attacks easily. It may be possible to retroactively detect then by the autoupdater after the network is up, but an attacker will always be able to the system appear to just be out-of-date.  
*Exposure*: Any user that logs in is exposed across reboots.  
*Notes*: Downgrade/replay attacks of this nature will be less dangerous if autoupdater is able to run prior to Chrome starting, but there will most likely be a race between the two. It may make sense to include a version check early in the Chrome startup process to detect seriously out-of-date browsers/systems prior to their opening dangerous pages.

## EEPROM map

The below is the figure of EEPROM. These sizes may change as we progress with implementation.

[](http://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery/eeprom-map.png?attredirects=0)

## Firmware block descriptions

### Boot stub

Must be at the top of EEPROM, since most processors jump to the top of memory (0xFFFFFFF0) after internal initialization.

Contains the "root key" - the official public key used to verify the signature of the next stage of firmware. The private key is not contained on the device, and must be protected from all unauthorized access. To reduce exposure of the private root key, the private root key will be used to sign a second date-limited or numbered key stored in the rewritable firmware, which is then used to sign that firmware. Validation of the date or key number could be done via a TPM module.

**Pseudocode**

1. Initialize processor and RAM (and implicitly, those parts of the north bridge necessary to initialize RAM), using conservative timings.
2. If user-forced recovery mode, skip to attempt loading recovery firmware. (The recovery button asserts an I/O line that can be measured by the firmware.)
3. Check the non-volatile register or memory region for a recovery mode cookie value.  If this is set, some later stage of the boot process must have failed and requested recovery mode, so skip to attempt loading recovery firmware.
4. Attempt loading Firmware A.
   1. Copy Firmware A to RAM.
   2. Verify signature of Firmware A using public key stored in boot stub.
   3. If signature is valid, jump to Firmware A Setup.
5. Firmware A is bad. Repeat for Firmware B.
6. Both A and B are bad. Attempt loading recovery firmware to RAM and verify signature. If valid, jump to recovery firmware.
7. All recovery options have been exhausted. Catch fire / emit POST code / etc.

### 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 Chrome 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.

### Boot log

A boot log will be stored at the bottom of the writable section of firmware. This will store the following types of events:

* Each time recovery firmware is run, with information on what triggered it (manual, bad firmware, bad root filesystem, etc.).
* When a recovery completes, including which actions were taken.

It does not store information on successful boots. This removes the need to support EEPROM writing in the normal boot process, and conserves space for log entries dealing with real errors.

The boot log can be uploaded as part of the autoupdate process. It can then be cleared and reused for new log entries.

### Recovery firmware

This firmware attempts to recover from bad firmware or rootfs by loading a recovery image off an external storage device.

**Pseudocode**

1. Write a log entry to the boot log indicating why recovery mode was activated (manual, bad writable firmware, user triggers recovery mode, etc.)
2. Initialize enough of the chipset to be able to access a storage device (USB drive, SD card, etc.)
3. Check for an inserted storage device.  If no storage device is present, skip to displaying recovery instructions.
4. Verify the signature of the recovery image, using a public key stored in the recovery firmware.  If the signature is invalid, skip to displaying recovery instructions.
5. Load and run the recovery image.
6. If we're still here, either no storage device was inserted, or the storage device did not contain a valid image.  Display recovery instructions:
   1. Initialize the display.
   2. Display instructions.  These instructions must be internationalized and rendered in multiple languages.  The instructions must include:
      * A warning that the computer is in recovery mode
      * How to obtain a trusted recovery image (for example, instructions to go to a second computer and browse to a specified URL for further instructions)
      * An indication if an already-inserted storage device does not contain a valid recovery image (sorry, try again).
7. Wait for a storage device to be inserted, then go back to step 3.

Most of the recovery work is left to the recovery image loaded from the storage device. This allows for publishing updated recovery images and instructions. Because the recovery firmware is etched in stone (well, as electrons in floating gates) at the time the device is launched, it needs to be as simple and robust as possible.

## Recovery image

This is the software loaded onto a storage device (USB drive, SD card, etc.) which does the bulk of the recovery work.

We will need to provide a downloadable installer for users to use to install the recovery image on the storage device. This installer should be supplied for all popular operating systems (Chrome OS, Windows, Mac OS X, Ubuntu Linux). The installer will:

* Prompt the user to select a destination for the recovery image
  + Ideally, the user will only be able to select destination devices appropriate for recovery of the laptop model they are attempting to recover.
  + Also, the user should only be able to select removable devices. We don't want them trashing their hard disk.
* Warn the user that this will erase whatever is on that destination, and prompt for confirmation
* Reformat the destination storage device
* Install the recovery image on the storage device
* Prompt the user to insert the storage device into the Chrome OS device

Ideally, the installer would be able to back up the current data on the destination device, before reformatting and writing the recovery image. When recovery is complete, the user could re-insert the destination device and have the original contents restored.

The recovery image should contain an entire clean copy of the firmware and rootfs. This way, a user can make a recovery device ahead of time. For example, if a user is going to be somewhere with poor connectivity, they could make a recovery USB drive at home and keep it in their bag. Alternately, the system could come with a recovery device (though users might end up reformatting it and filling it with images of kittens).

The recovery image on the storage device would do something like the following:

1. Initialize the display and tell the user what's going on. Reassure them that their Chrome OS system is trying to recover.
2. If developer firmware is detected, give the user a choice of "JUST FIX IT" or "Scary Settings for L33t H4x0rs". If the user picks the latter:
   1. Give the developer more control over each stage in the rest of the recovery.
   2. For example, a developer might not want to wipe and replace rootfs or stateful data, or might want to make a backup of that data.
3. Run system tests. It's possible that the boot problems are due to bad hardware. Now's the time to detect that.
   1. If hardware is sufficiently bad that the rest of recovery can't be run, show the user information on how to return it.
4. Verify both firmware A and firmware B are up-to-date and valid. Load known good firmware as needed.
5. Verify rootfs-A and rootfs-B are up-to-date and valid. Load known good rootfs images as needed.
6. Wipe the stateful data. Anything the user cares about should be in the cloud anyway. If the user's been hacked, wiping the data will put the device back in a known good state.
   1. Optionally, it can inform the user that data on the system will be erased.
   2. This message should emphasize that most data is saved to the cloud, so that the user is more likely to proceed with recovery.
   3. It is less desirable to give the user a choice whether or not to delete the stateful data. Most users would pick the less scary but also less secure option of NOT deleting data - but this leaves them vulnerable to persistent hacks on the data partition, such as manipulating their /etc/hosts file or bookmarks.
7. If recovery mode was manually triggered, ask the user why they're running it.
   1. Couldn't even log in; crash during boot
   2. Kept crashing after boot
   3. Worried my system was hacked
   4. Hey, what's this button do?
8. Write an entry to the boot log describing repairs performed.

### Recovery via Chrome OS

The recovery installer should run on a healthy Chrome OS system. That is, it should be able to download the recovery image, reformat the storage device, and copy the recovery image to it.

* This enables a healthy Chrome OS system to create its own recovery device in advance.
  + Perhaps we should periodically advise users to update their recovery device?
* It also enables one Chrome OS system to download a recovery image for a different model. (A user has a corrupt system, so they go to their friend's house and use the friend's Chrome OS system to make a recovery image).

### Making recovery images read-only

If the recovery image is inserted into a powered-on and hacked system, the hacked software could delete or corrupt it. This won't cause the recovery firmware to load that corrupt image, because the corrupt image will fail the signature check. It will be annoying to the user, who will need to reflash the storage device.

This is more of an issue if we want to include an internal recovery image (for example, on an internal storage device).

The SD standard specifies a physical write-protect notch for SD cards, similar to those on a 3.5" floppy disk. Unfortunately, the implementation of the write-protection is purely software, so pwned drivers can choose to ignore the write-protect detect signal.

Some USB drives have a write-protect switch on them. In this case, the protection is handled by the USB drive itself.

Some eMMC chips have a number of protection mechanisms including:

* an external LOCK# pin which can be used to make the device read-only
* a write-once bit which makes the device read-only until reboot

Since these chips come in sizes up to 2GB (~$10 at stores), they provide a possible place to store a recovery image. Some system architectures may be able to use the eMMC drive to hold the main firmware image also.

If we have an internal storage device, it's possible we could wire it so it's only enabled if the recovery button is pressed at boot time. For example, we could use that button to latch up a circuit which powers that device, so that it will remain powered during that boot only.

### Using recovery mode to load developer mode firmware/software

If the rootfs image on the SD card is signed by someone else, the recovery image will display a screen similar to the Developer Mode screen.  Instructions on this screen should include:

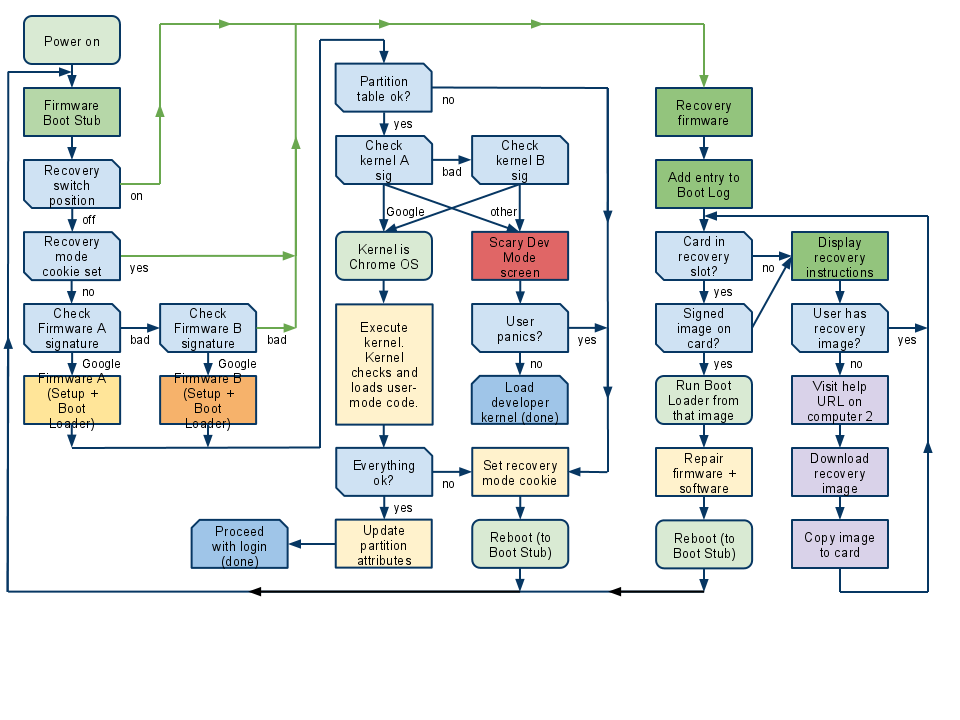
* Notice that the recovery image is signed by someone else
* How to obtain a trusted recovery mode image
* Instructions on installing a trusted recovery image
* A means of allowing the user/developer to proceed down the "untrusted" path

Note that this screen must also be internationalized / rendered in multiple languages. This is potentially much easier, since by the time we're doing this the recovery image has booted a full OS with GUI.

## Boot flowchart

The three main downward flows in this chart show:

* Start of boot in read-only firmware
* Normal continuation of boot, loading kernel from internal drive
* Recovery mode

[](https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery/ChromeOSFirmwareBootandRecoveryPUBLISHED.png?attredirects=0)

Summary:

Important points on the Chrome OS

1. Operating system from Google based on the Chrome browser
2. Designed around web apps
3. Browser, Gmail, Google Docs, YouTube, Netﬂix, games
4. Google Drive, Chrome Sync, and persistent app state
5. Synced, backed up, and updated automatically

Important points on the Chromium OS

1. Built from publically available Open Source code
2. Only runs on devices in developer mode
3. Allows shell and root access
4. No Flash, Netﬂix, DRM

Difference between Chromium OS and Chrome OS

|  |  |
| --- | --- |
| Chromium OS | Chrome OS |
| Built from publicly available Open Source code | Digital signature from Google |
| Only runs on devices in developer mode | Runs on systems in production mode |
| Allows shell and root access | Branding |
| No Flash, Netflix, DRM | Flash, Netflix, and DRM |

Chrome OS firmware

1. Based on coreboot and u-boot
2. Coreboot provides the framework for hardware initialization
3. “depthcharge”: u-boot as coreboot payload
   * 1. Provides ﬂexible boot of Linux from various media
4. Read-only ﬁrmware for root of trust and recovery mode
5. A/B read-write ﬁrmware available for fallbacks during updates
6. Includes SeaBIOS to boot arbitrary OSes
7. Open Source ﬁrmware for embedded controller
8. Implements veriﬁed boot procedure
9. Enforces developer-mode switch requirements

i) Physical presence (switch or keyboard)

ii) Wiping local state when switching

Developer mode

1. Physical switch on older hardware
2. Esc-Refresh-Power on newer hardware(Refresh-Power is instant hard reset)
3. Allows bypassing veriﬁed boot via explicit keyboard interaction
4. Enforced in ﬁrmware or embedded controller
5. Not changeable from OS
6. Wipes stateful partition, after enforced delay
7. Allows booting USB or BIOS

Chrome OS kernel

1. Extensively patched Linux kernel
   1. Backported drivers and improvements
   2. Security enhancements and hardening
   3. Not new APIs
2. A/B copies for redundancy during updates
3. Stored on dedicated partitions to simplify depthcharge
4. Wrapped in veriﬁed boot container, with kernel command line
5. Veriﬁcation information for dm-verity on kernel command line
6. Edit formatted kernel and command line via vbutil\_kernel

Chrome OS userspace

1. Linux distribution
2. Based on Gentoo
   1. -O99 -funroll-loops -fomit-instructions -ftw
3. Uses the Portage build system and packaging infrastructure
4. Pulls in many packages from Gentoo, and adds patches
5. Adds its own chromiumos-overlay with the Chrome OS core and additional packages
6. Adds board-speciﬁc overlay for each target board
7. Notable divergence from Gentoo: Upstart

Verified BOOT:

* Run-time attacks against unsignable data: Chromium configuration
* Persisted attacks by defaulting Chromium to launch the attack site on next start
* Logged-in user data is exposed immediately after compromise.
  + With partial signing, a Chromium replacement would result in cross-user exposure after reboot.
  + With full signing, other system users would be notified prior to exposure
* Metadata attacks against the kernel will not be caught by signing per-file.
* Downgraded manifest file attacks are difficult to detect since there is no current way to encoded tamperproof system time into manifest files.