**Cui, Yi**

Q1: More on Android Push notification and alternative security. Security issues, how to authorize it.

A1: Official way is to use the Google Cloud Messaging for Android. An excellent complete overview is located at <https://developer.android.com/google/gcm/index.html>

There are two servers involved in Google Cloud Messaging, the third party / app developer's server and Google's server. Data sent from the 3rd party server to the Google server is sent over HTTPS with the 3rd party's unique API key in the POST header, so done correctly this should provide authentication of the 3rd party server, integrity and replay protection, i.e. it should not be spoofable. This relies on both the third party and Google securely implementing their services.

Google Cloud Messaging (GCM) for Android allows developers to send and receive different types of messages to and from applications installed on Android devices. A developer can, for example, send messages that contain up to 4KB of structured data from a server the developer owns through a Google-run GCM server to all user installations of the developer's GCM-enabled apps. The applications don't even have to be running on user devices as the received messages will be broadcast by the Android OS and the targeted apps will be woken up.

The GCM message data can include links, text advertisements or commands.

Alternatives solutions exist like <http://www.parse.com> for cross-platform push notifications.

**Deng, Pan**

Q1: Hybrid applications.

A1: Live demo based on the project in GitHub – *HybridApp*.

Q2: Protect apps data.

A2: Also check my presentation on SlideShare - <http://de.slideshare.net/fastlink2/droidcon-eastern-europe-2013-how-secure-is-an-androidapp> . A good idea is to have a look also at the project dexguard - <http://www.saikoa.com/dexguard> and Chinese alternatives:

* <http://www.bangcle.com/>
* <http://www.ijiami.cn/>
* <http://dev.360.cn/protect/welcome/>

Some words about the platform Google Bouncer:

* Introduced in Android 4.2
* Apps installed via Google Play are already verified on the "server side"
* Meant to protect against malicious apps installed via alternative channels:
  + Side-loaded from SDCARD, remote HTTP server, message attachment, etc.
  + Alternative app markets (such as Amazon app-store)
  + ADB installed (including through and IDE such as Eclipse)
* Enabled via Setting → Security → Verify apps
* The actual implementation is a combination of Google Play client-side app, and Google Play cloud infrastructure
* When an app is about to be installed:
  1. Google Play client collects the information about the app
  + Package/app name
  + App version
  + SHA1 digest of the app contents
  + The location (e.g. URL) from where the app is being installed
  + Information about the device (ID, IP, etc)
  1. Google Play cloud uses this information to verify the app based on some internal heuristics and responds to the client
  2. When it receives the response, the client:
* Blocks the installation if the app is found to be "dangerous"
* Gives user a warning if the app is found to be "potentially harmful or dangerous", at which point the user can cancel the installation or proceed at their own risk
* Resumes the installation without user’s intervention if the app appears to be safe

**Observations:**

* As of November 30, 2012, the Google’s App Verification Service was found to have a detection rate of only 15-20% - on a sample of 1260 apps belonging to 49 families of known Android malware (compared to 50-100% for established AV vendors)
* Most malware is distributed such that it is mutated on every download, so digest-only-based detection offers weak protection
* For more info, see An Evaluation of the Application ("App") Verification Service in Android 4.2 from NC State University - <http://www.cs.ncsu.edu/faculty/jiang/appverify/>

**Gu, Jinghui**

Q1: More about the notification manager.

A1: Please check the project *00\_Notifications* in GitHub. Marius will demo further also on Android Wear based on some examples from <http://www.slideshare.net/chuazy/android-wear-presentation> .

How to move notification to the bottom, and slide it from bottom-top?

Answer, change in PhoneStatusBar.java the getStatusBarGravity() method:

*protected int getStatusBarGravity() {*

*return Gravity.TOP | Gravity.FILL\_HORIZONTAL;*

*}*

into

*protected int getStatusBarGravity() {*

*return Gravity.BOTTOM | Gravity.FILL\_HORIZONTAL;*

*}*

maybe also:

*mNotificationPanelGravity = Gravity.START | Gravity.BOTTOM;*

also some info on <http://stackoverflow.com/questions/5971585/move-the-android-status-bar>

**Han, Junchao**

Q1: How to handle touch events. How to simulate touch events.

A1: On Android, there's more than one way to intercept the events from a user's interaction with your application. When considering events within your user interface, the approach is to capture the events from the specific View object that the user interacts with using event listners.

*@Override*

*public boolean onTouchEvent(MotionEvent event) {*

*// get pointer index from the event object*

*int pointerIndex = event.getActionIndex();*

*// get pointer ID*

*int pointerId = event.getPointerId(pointerIndex);*

*// get masked (not specific to a pointer) action*

*int maskedAction = event.getActionMasked();*

*switch (maskedAction) {*

*case MotionEvent.ACTION\_DOWN:*

*case MotionEvent.ACTION\_POINTER\_DOWN: {*

*// TODO use data*

*break;*

*}*

*case MotionEvent.ACTION\_MOVE: { // a pointer was moved*

*// TODO use data*

*break;*

*}*

*case MotionEvent.ACTION\_UP:*

*case MotionEvent.ACTION\_POINTER\_UP:*

*case MotionEvent.ACTION\_CANCEL: {*

*// TODO use data*

*break;*

*}*

*}*

*invalidate();*

*return true;*

*}*

Reference implementation:

* <http://developer.android.com/guide/topics/ui/ui-events.html>
* <http://developer.android.com/training/graphics/opengl/touch.html>

**He, Liang**

Q1: Comparison between Android and iOS.

A1: Marius will discuss free on this.

Q2: SELinux in Android. How to root a phone (Nexus 4).

A2: As part of the Android security model, Android uses Security-Enhanced Linux (SELinux) to enforce Mandatory Access Control (MAC) over all processes, even processes running with root/superuser privileges (a.k.a. Linux capabilities).

Android includes SELinux in enforcing mode and a corresponding security policy that works by default across the Android Open Source Project. In enforcing mode, illegitimate actions are prevented and all potential violations are logged by the kernel to dmesg. Android device manufacturers should gather information about errors so they may refine their software and SELinux policies before enforcing them.

In Android 4.3, SELinux was fully permissive. In Android 4.4, SELinux was made enforcing for the domains for several root processes: installd, netd, vold and zygote. All other processes, including other services and all apps, remain in permissive mode to allow further evaluation and prevent failures in Android 4.4. Still, an errant application could trigger an action in a root process that is not allowed, thereby causing the process or the application to crash.

More on <http://source.android.com/devices/tech/security/se-linux.html>

Rooting a phone usually involves finding a flaw or hole in the system. Nexus 4 was cracked after a 18.000$ bounty which leaded to the creation on the <http://towelroot.com/> app , 1 click root :).

**Huang, Yang**

Q1: How to develop a MDM.

A1: Marius will demo some slides from the “Android Security Training”.

**Li, Tingtao**

Q1: SurfaceFlinger and 3D display.

A1: There are two different methods for rendering objects to the screen in Android: View ( <http://developer.android.com/reference/android/view/View.html> ) and OpenGL ( <http://developer.android.com/guide/topics/graphics/index.html> ).

SurfaceFlinger ( source code available at <https://android.googlesource.com/platform/frameworks/native/+/android-cts-4.1_r1/services/surfaceflinger/SurfaceFlinger.cpp> ) is responsible for COMPOSITING all those surfaces. A common misunderstanding is that SurfaceFinger is for DRAWING. It is not correct. Drawing is the job of OpenGL. The interesting thing is SurfaceFlinger used openGL for compositing as well. To reduce the power and increase the performance, SurfaceFlinger also use display hardware directly for compositing, avoid the compositing of openGL. This is the so called HardWare Composer (HWC). It takes data( which is surface) from different application which could be 2D or 3D and finally combine it to obtain a main surface which will be fed to memory( which is framebuffer).

Renderscript is an API for intensive computation using heterogeneous computing. It allows developers to increase the performance of their applications at the cost of writing a greater amount of more complex code. It provides the developer three primary tools: A general purpose compute API across different system computing hardware, a compute API similar to CUDA, OpenCL or GLSL, and a familiar C99-derived language. It can also be used for 3D graphics. **As of Android 4.1, Renderscipt's experimental 3D rendering API has been deprecated, and now exists solely as a compute API.**

There are also a large number of 3D third party engines for Android - http://mobilegameengines.com/android/3d\_game\_engines

* Alien3D
* Ardor3D
* BatteryTech
* Dwarf FW
* DX Studio
* EDGELIB
* etc

Read more on:

* Android Training "Displaying Graphics with OpenGL ES" - <http://developer.android.com/training/graphics/opengl/index.html>
* Android API Guides "OpenGL ES" - <http://developer.android.com/guide/topics/graphics/opengl.html>
* Android Reference "Package android.opengl" - <http://developer.android.com/reference/android/opengl/package-summary.html>
* <https://www3.ntu.edu.sg/home/ehchua/programming/android/Android_3D.html>
* <http://obviam.net/index.php/3d-programming-with-android-projections-perspective/>

**Mu, Weiyang**

Q1: HAL

A1: Unlike traditional Linux based systems, Android applications communicate with the underlying hardware through Java APIs not by system calls. The Hardware Abstraction Layer or HAL is the glue between any device (part of the kernel) and the corresponding JNI interface.

Related material:

* Official documentation - <https://source.android.com/devices/camera/camera3_requests_hal.html>
* HAL sensors overview - <http://source.android.com/devices/sensors/>
* <http://sujaiantony.wordpress.com/2012/06/25/an-android-101-hardware-and-hal/>
* Adding support for a Camera : <http://processors.wiki.ti.com/index.php/TI-Android-DevKit-Camera_Porting_Guide>
* Patrick Brady’s Google I/O talk: <https://sites.google.com/site/io/anatomy–physiology-of-an-android>
* Extending the Android HAL - <http://www.opersys.com/blog/extending-android-hal>

**Qin, Jiajia**

Q1: How to manage the activity lifecycle. Suggestions.

A1: Marius will free discuss on this.



Q2: How to process the video pipeline (open, decode, display).

A2: Because the Android application framework is well designed to parse the media file and set up the processing pipeline automatically, the programmer don’t have to deal with the detail of the video content and processing. The framework has a unified interface to accept the media content with different codecs.

Although the media codec we used in this sample app example is H.264, it applies to all the codecs and containers that Android supported, for example, MPEG-4. Please go to this page to check the codecs and containers supported by Android.

Check the open source products: like <https://code.google.com/p/dolphin-player/>, <https://github.com/havlenapetr/FFMpeg> and closed source like Nexplayer - <http://www.nextreaming.com/>

*public class VideoPlayback extends Activity implements OnPreparedListener {*

*@Override*

*public void onCreate(Bundle savedInstanceState) {*

*super.onCreate(savedInstanceState);*

*setContentView(R.layout.media);*

*mPreview = (SurfaceView) findViewById(R.id.surface);*

*holder = mPreview.getHolder();*

*// Set up the play/pause/reset/stop buttons*

*mPlay = (ImageButton) findViewById(R.id.play);*

*mStop = (ImageButton) findViewById(R.id.stop);*

*mPause = (ImageButton) findViewById(R.id.pause);*

*mPlay.setOnClickListener(new View.OnClickListener() {*

*public void onClick(View view) {*

*if (mMediaPlayer == null){*

*prepareVideo();*

*}*

*if (!mMediaPlayer.isPlaying()) {*

*mMediaPlayer.start();*

*videoStopped = false;*

*} else {*

*}*

*}*

*});*

*mPause.setOnClickListener(new View.OnClickListener() {*

*public void onClick(View view) {*

*if (mMediaPlayer != null) {*

*mMediaPlayer.pause();*

*}*

*}*

*});*

*mStop.setOnClickListener(new View.OnClickListener() {*

*public void onClick(View view) {*

*if (mMediaPlayer != null) {*

*stopVideo();*

*}*

*}*

*});*

*}*

*private void stopVideo() {*

*videoStopped = true;*

*if (mMediaPlayer != null) {*

*mMediaPlayer.stop();*

*mMediaPlayer.release();*

*mMediaPlayer = null;*

*}*

*}*

*private void prepareVideo() {*

*// Create a new media player and set the listeners*

*try {*

*videoUri = Uri.parse("android.resource://" +*

*getPackageName() + "/" + R.raw.h264);*

*mMediaPlayer = new MediaPlayer();*

*mMediaPlayer.setDataSource(this, videoUri);*

*mMediaPlayer.setDisplay(holder);*

*mMediaPlayer.setAudioStreamType(AudioManager.STREAM\_MUSIC);*

*mMediaPlayer.prepare();*

*mMediaPlayer.setOnPreparedListener(this);*

*} catch (Exception e) {*

*Log.e(TAG, "error: " + e.getMessage(), e);*

*finish();*

*}*

*videoStopped = false;*

*}*

*public void onPrepared(MediaPlayer arg0) {*

*// TODO Auto-generated method stub*

*mMediaPlayer.start();*

*videoStopped = false;*

*}*

*}*

More resources:

* <https://software.intel.com/en-us/android/articles/video-playback-app-on-android-whitepaper>
* <http://docs.gstreamer.com/display/GstSDK/Android+tutorial+4%3A+A+basic+media+player>
* <http://docs.gstreamer.com/display/GstSDK/Android+tutorial+2%3A+A+running+pipeline>

**Qiu, Jianlin**

Q1: Interact to a service provided by google aka search engines, speech to text.

A1: Depends really on the type of services, usually we use the Google Api console but we can also use other exotic options like the search bellow. Check further also the <http://developer.android.com/guide/topics/search/index.html>

Web search:

*public void onSearchClick(View v)*

*{*

*try {*

*Intent intent = new Intent(Intent.ACTION\_WEB\_SEARCH);*

*String term = editTextInput.getText().toString();*

*intent.putExtra(SearchManager.QUERY, term);*

*startActivity(intent);*

*} catch (Exception e) {*

*// TODO: handle exception*

*}*

*}*

Voice based search:

*Intent sp=new Intent(RecognizerIntent.ACTION\_WEB\_SEARCH);*

*sp.putExtra(RecognizerIntent.EXTRA\_LANGUAGE\_MODEL, RecognizerIntent.LANGUAGE\_MODEL\_FREE\_FORM);*

*sp.putExtra(RecognizerIntent.EXTRA\_PROMPT, "Speak please");*

*startActivity(sp);*

**Ren, Shidong**

Q1: 64 bit Android.

A1: Some people say we have 64 bit in Android L only because of the ARMv8 architecture - <http://www.androidcentral.com/why-64-bit-processors-really-matter-android> !

A 32-bit processor can access a maximum of 4GB of memory. This is one reason why desktop and laptop computers and their operating systems are all 64-bit these days. If they weren't, they would never be able to use more than 4GB of memory.

Mobile phones and tablets are not yet struggling for memory in the same way as PCs and Macs are. Android phones and tablets typically have 1GB or 2GB of memory and the Samsung Note 3 and 4 are one of the few devices with 3GB, a little below the 4GB limit imposed by 32-bit processors.

As apps and mobile operating systems grow, there will come a day when 4GB of memory just isn't enough and then a 64-bit processor will be needed so phones can have more memory. Android L will bring 64 bit support.

To take full advantage of a 64-bit environment, you need 64-bit software. Apple’s plan for iOS 7-8 is similar to that which Microsoft used years ago when they rolled out 64-bit versions of their OS: enable a hybrid environment which allows both 32- and 64-bit apps to run. Those that are 64-bit should run better than the 32-bit equivalents on the same system, but the 32-bit apps should still run every bit as good as they would have on a 32-bit system. In the end, it’s a win for going 64-bit.

Android finds itself in a different situation. Since Android apps utilize the Dalvik VM, in theory, Dalvik could be “upgraded” to 64-bit, and apps would naturally run in 64-bit because they’re compiled Just In Time — that compilation can be to 32-bit, 64-bit, or some other future architecture not yet dreamed of. For example, today’s Java programs can run on a 32- or 64-bit Java virtual machine without modification.

Q2: MediaCodec in Jelly.

A2: Android has included MediaPlayer since API level 1. MediaPlayer provides a simple way to play back audio and video. However, it’s limited by the fact that it offers only three media formats: mp4, 3gpp, and mkv (beginning with Android 4.0). To play unsupported formats, many developers have used the ffmpeg software decoder with the drawback of having large compiled binaries for all the platforms.

It was added to allow direct access to the media codecs on the device. As such, it provides a rather "raw" interface. While the MediaCodec class exists in both Java and C++ sources, only the former is public.

In Android 4.3 (API 18), MediaCodec was expanded to include a way to provide input through a Surface (via the createInputSurface method). This allows input to come from camera preview or OpenGL ES rendering. Android 4.3 was also the first release with MediaCodec tests in CTS, which helps ensure consistent behavior between devices.

Android 4.3 also introduced MediaMuxer, which allows the output of the AVC codec (a raw H.264 elementary stream) to be converted to .MP4 format, with or without an associated audio stream.

Read more on:

* Encoder/decoder official documentation <http://developer.android.com/reference/android/media/MediaCodec.html>
* Sample demo apps based on MediaCodec API - <https://github.com/vecio/MediaCodecDemo>
* <http://bigflake.com/mediacodec/>
* Intel Developer Zone article - <https://software.intel.com/en-us/android/articles/android-hardware-codec-mediacodec>

**Yan, Shaopu**

Q1: What is the difference between the emulated image and the hardware.

A1: Marius will discuss free on this. For each different phone we have an unique mode of flashing, for most oft he Nexus devices we have the following steps:

* Copy the file *update.zip* in the root of the SD-CARD
* Reboot in recovery mode using the phone (power+vol up?!)
* Choose *update.zip*
* Flash the device
* Reboot the device

**Once we compile the AOSP what we do?**

*make updatepackage*

*fastboot -w update $ANDROID\_PRODUCT\_OUT/$TARGET\_PRODUCT-img-eng.$USER.zip*

* Official documentation for flashing is located at <http://source.android.com/source/building-devices.html> but a lot depends on the bootloader .
* How to build Android AOSP for Nexus 4 - <http://nosemaj.org/howto-build-android-nexus-4>
* How to build Android 4.3 for Nexus 4 - <http://nosemaj.org/build-android-4-3-nexus-4>
* Nexus 4 si Nexus 5 - <https://developers.google.com/android/nexus/drivers>
* Howto Build Android KitKat (4.4) for the Google Nexus 5 - <http://nosemaj.org/howto-build-android-kitkat-nexus-5>

**Yang, Yibin**

Q1: How to port Android on an iPhone.

A1: Easy, nobody achieved this but still easy ☺.

Android provides you with the freedom to implement your own device specifications and the drivers to support them. The hardware abstraction layer (HAL) gives you a standard way to create software hooks in between the Android platform stack and your hardware. In addition, the Android operating system is open-sourced to help you through your device's bringup.

Resources:

* Official documentation about how you can port a new device to Android - <http://source.android.com/devices/>
* Crazy ideas - <http://www.googlehunting.com/install-android-iphone-six-easy-steps/>
* <http://theiphonewiki.com/wiki/IDroid>

**Ye, Jingfu**

Q1: How to access the GPS location and how to integrate it with the maps.

A1: Based on the sample located on <http://developer.android.com/training/location/receive-location-updates.html> and <http://developer.android.com/training/location/index.html> I will demo the /Samples/LocationUpdates

**Zhang, Hongjiang**

Q1: Sniffer for Android and not only the TCPDUMP.

A1: Learn from the best :

* Android serial sniffer - <https://github.com/landswellsong/android-serial-port-api-sniffer>
* Android mobile sniffer - <http://dev.umitproject.org/projects/pm-mobile> with sourcecode located at - <https://github.com/umitproject/pm-mobile>

**Zhong, Xueliang**

Q1: System service for change the performance governor in order to perform better.

A1:

On ROMs like Cyanogenmod, there are built in settings that allow you to change CPU governors and clock speeds, setting how much power your CPU is allowed to use ( keeps it from running at full speed ).

The governors control how the CPU acts in different situations; a performance governor is going to ramp up the CPU all the time. Performance will be great, but the battery is going to suffer. Another governor I prefer is hotplug; it shuts down one core of a dual-core CPU when in sleep mode, which improves the idle battery life. There’s tons of different governors and clock speed combinations, so like changing kernels, it’s a good idea to read up on what each one does.

XDA also provides some tutorials on this.

An alternative is an application to control your CPU. Not all ROMs have CPU controls built in, so an application like that can manage the clock speeds and governors if your phone is rooted. All of the same rules apply. Higher clock speeds mean faster, more battery draining performance. Lower clock speeds mean worse performance, but less power consumption. The trick is finding a middle ground where you can live with the performance and can see the extra battery life.

And there is an app for that: One Power Guard - <http://www.onexuan.com/battery/download.php>

One Power Guard for rooted and custom ROM users gives them full control over power and battery management. This app controls just about everything any power user could want: overclocking, underclocking, I/O scheduling, CPU governor, and much more. It even includes six power saving modes which include:

**AI Mode:** It intelligently adjusts cpu frequency and optimizes kernel to greatly extend battery time according to system loading. For General User. It optimizes system parameter, charging parameters,cpu parameter,wifi parameters, etc and different from other modes.

**Powersave Mode:** It runs devices under low frequency to extend standby time. For power saver and those who use devices less frequently. It optimizes system parameter, charging parameters,cpu parameter,wifi parameters, etc and different from other modes.

**Game/Video Mode:** It provides flow game and video experience while offering satisfactory power-saving strategy. For game and video addicts. It optimizes system parameter, charging parameters,cpu parameter, wifi parameters, etc and different from other modes.

**Call Mode:** It powers up the core function of phone conservation. For business persons trapping by phone. It optimizes system parameter, charging parameters,cpu parameter,wifi parameters, etc and different from other modes.

**Standby Mode:** It lowers system frequency and prolongs stand-by time to the extreme. For sleepers and dreamers. It optimizes system parameter, charging parameters,cpu parameter,wifi parameters, etc and different from other modes.

**Custom Mode:** It indulges pros to customize cpu frequence, scheduling and io adjustment, and to choose parameters of their own. For advanced users. It optimizes system parameter, charging parameters,cpu parameter,wifi parameters, etc and different from other modes.

Alternatives:

1. SetCPU - <http://www.setcpu.com/documentation.html#governor> and <http://www.setcpu.com/index.html>
2. <http://onexuan.com/blog/2013/08/onepowerguard-pro-cpu-governor-introduction/>
3. CPU frequencies are configured in code: <https://www.codeaurora.org/cgit/quic/la/kernel/msm/tree/arch/arm/mach-msm/acpuclock-8064.c?h=jb_2.6>
4. <http://bamboopuppy.com/android-cpu-frequency-using-cpufreq-ondemand-governor/>
5. <http://icrontic.com/discussion/95140/android-cpu-governors-and-you-setcpu-system-tuner-tegrak>
6. Voltage Control App - <https://play.google.com/store/apps/details?id=com.darekxan.voltagecontrol&hl=en>
7. CPU Tunner - <https://play.google.com/store/apps/details?id=ch.amana.android.cputuner&feature=search_result>
8. No frills CPU App - <https://play.google.com/store/apps/details?id=it.sineo.android.noFrillsCPU&hl=en>
9. Simple CPU - <http://forum.xda-developers.com/showthread.php?p=21625211>
10. Antuntu CPU App - <https://play.google.com/store/apps/details?id=com.antutu.CpuMaster&hl=en>

Check the tech paper from Google <https://source.android.com/devices/tech/power.html>

Check the official platform implementation of Battery tab in Settings app, the project app is located at <https://github.com/android/platform_packages_apps_settings> and the class doing all the magic is <https://github.com/android/platform_packages_apps_settings/blob/master/src/com/android/settings/BatteryInfo.java>

Check the Intel PowerTop open source project at <https://01.org/powertop/> and sources at: <https://github.com/fenrus75/powertop>

The BatteryInfo.java has an included BroadcastReceiver for Intent.ACTION\_BATTERY\_CHANGED and is taking from the passed Intent most of

the informations need it like temperature, etc.

Check the Power Tutor application website at : <http://ziyang.eecs.umich.edu/projects/powertutor/> and the source code is available at: <https://github.com/msg555/PowerTutor>

Check also the SysPower app website <https://code.google.com/p/syspower/> and the source code is available at: <https://code.google.com/p/syspower>

Also command line alternatives + java code alternatives:

*adb shell dumpsys batteryinfo*

*public static void dumpBatteryInfo() {*

*try {*

*String cmd = "dumpsys battery";*

*Process script = Runtime.getRuntime().exec(cmd);*

*BufferedReader in = new BufferedReader(*

*new InputStreamReader(script.getInputStream()));*

*String line = null;*

*while ((line = in.readLine()) != null) {*

*Log.i ("BATTERY","Battery stats: " + line);*

*}*

*} catch (Exception ex) {*

*}*

*}*

*<uses-permission android:name="android.permission.DUMP" />*

**Li, Pan**

Q1: Why Android choose Linux kernel and how the kernel adjusted itself for the Android.

A1: There is no short answer. Here is the long one ☺.

Although the platform built on top of the Linux kernel, it is not a Linux distribution. It does not support the X-windowing system nor does it use the standard Linux libraries including the Gnu C Library.

Not many people in the mainstream understand that there is a difference between the Linux kernel and a Linux distribution. Android will be a Linux ―flavor whether it wants to be or not.

Google did take Android in a direction that wasn't compatible with the mainstream Linux kernel. As Greg Kroah-Hartman, the maintainer of the stable Linux kernel for the Linux Foundation and head of the Linux Driver Project, wrote in Android and the Linux kernel community, "The Android kernel code is more than just the few weird drivers that were in the drivers/staging/android subdirectory in the kernel. In order to get a working Android system, you need the new lock type they have created, as well as hooks in the core system for their security model. In order to write a driver for hardware to work on Android, you need to properly integrate into this new lock, as well as sometimes the bizarre security model. Oh, and then there's the totally-different framebuffer driver infrastructure as well." That flew like a lead balloon in Android circles.

This disagreement sprang from several sources. One was that Google's Android developers had adopted their own way to address power issues with WakeLocks. The other cause, as Google open source engineering manager Chris DiBona pointed out, was that Android's programmers were so busy working on Android device specifics that they had done a poor job of co-coordinating with the Linux kernel developers.

Must read articles:

1. <https://lwn.net/Articles/472984/>
2. <http://elinux.org/Android_Kernel_Features>
3. <http://www.unixmen.com/why-is-android-built-on-linux-kernel/>

The pieces that exist in the Linux-next staging directory now are:

* **Binder**, the interprocess communication mechanism used within Android. Binder could conceivably be replaced with a standard IPC mechanism or, perhaps, with D-bus, but it has a number of unique features (zero-copy message transmission, thread management, credential passing) that are hard to replace in a straightforward manner. (See this article for a detailed look at various Linux IPC mechanisms, binder included).
* **Logger** is the kernel piece of the Android logging system. It implements a completely separate path for Android-specific log messages, which do not mix with normal kernel messages in any way. Other than adding a "facility" concept to kernel logging, it's not clear what this component offers, but it is also relatively self-contained and should not be too controversial.
* **The "low memory killer"** implements Android's interesting approach to application management. In the Android world, applications never choose to exit. They hang around until memory gets tight, at which point kernel starts to kill them off. It's a small piece of code that works using the "shrinker" mechanism, a standard way to register functions to be called when the kernel would like to free up some memory. So, even though it is memory-management code, it is relatively unintrusive and will not affect systems where it is not used.
* "**Pmem**" is Android's answer to the age-old problem of allocating large, physically-contiguous buffers after the system has been running for a while. It works in the usual way: a range of memory is set aside at boot time. One difference with pmem is that it exports a device to user space, allowing buffers to be allocated directly by applications and passed to drivers. That, in turn, leads to things like camera drivers being written with the assumption that user space can give them physically-contiguous buffers for video frames, something that would not be possible in a mainline kernel.
* Approaches like CMA seem like a better solution to this particular problem - if and when CMA is merged into the mainline. Meanwhile, however, applications have been written using pmem, so that interface is unlikely to go away in the near future.
* **The "RAM console"** saves log data to a special region of memory where it can be found and recovered after a reboot. It is a debugging tool.
* **"Timed GPIO"** is a simple mechanism whereby the kernel can schedule a specific setting for a GPIO line at some point in the future. An example use would be to ensure that the vibrator gets turned off regardless of what happens to the application that turned it on.