



Project Reveal

APPENDIX: ANDROID CODE SNIPPETS

Martyn Williams and Niklaus Schiess

APPENDIX: ANDROID CODE SNIPPETS

The following appendix includes snippets of code from analyzed North Korean devices.

MIRAE WI-FI

The different authentication steps are implemented in the `SmartCardService.apk` app. It also implements communication with the SIM card to ready crypto material. Some other phones nowadays also seem to have such apps installed in case communication with SIM cards is required. One example are banking apps that install services on SIM cards. Therefore, the implementation of `SmartCardService.apk` is most likely similar to these apps.

The Taeyang Smart Card Service is implemented in two parts:

- `kut.it3.SmartCardService`: The Java interface within the app.¹
- `libSIA.so`: A native library that implements communication with SIM cards and all required cryptographic functions.

The service is started in the `BootCompletedBroadcastReceiver` (`kut.it3.SmartCardService.BootCompletedBroadcastReceiver`) in the `onReceive()` function as shown in the following snippet:

```
public void onReceive(Context var1_1, Intent var2_2) {  
    if (var2_2.getAction().equals("android.intent.action.BOOT_COMPLETED")) {  
        Log.i("SmartCardService", "Starting SmartCard service after boot completed");  
        var1_1.startService(new Intent(var1_1, SimCardService.class));  
        SimCardService.start(var1_1);  
        [...]  
    }  
}
```

The service then runs in the background and monitors if SIM cards are plugged into the device.

The user interface just provides a button that allows you to get the authentication status. When clicking the button, the `SimCardManager.getSimAuthStatus()` function from the native library `libSIA.so` is called:

- `Java_kut_it3_SmartCardService_SimCardManager_getSimAuthStatus()`

This is the only JNI function that is exported by `libSIA.so`.

¹ "kut" is commonly used in North Korea as an acronym for Kim Chaek University of Technology

The key material is loaded via a function `getMEKeys()` (`getSimAuthStatus()` -> `smartcard_init()` -> `getMEKeys()`). The actual data is located in the following files:

- `/system/etc/ma_cert.dat`
- `/system/etc/ma_priv.dat`

Both are then decrypted with the same key in the `encKey` array (88 bytes), which is also exported. Decryption is just simple XOR for both the certificate and the private key. The key is a ECDSA private key that is used to check signatures. The Certificate Authority (CA) private and public keys are also hardcoded in the library.

The code includes references to the following supported curves:

- `aNistX962SecgCu` - NIST/X9.62/SECG curve over a 192 bit prime field
- `aNistSecgCurveO` - NIST/SECG curve over a 224 bit prime field
- `aX962SecgCurveO` - X9.62/SECG curve over a 256 bit prime field

The actual signatures are 256 bit long. They sign the data that is returned from the SIM card via the `GET_CHALLENGE()` function. The resulting signature is then used in a challenge response authentication mechanism that checks if the SIM card and device are both authorized to be used for the Mirae WiFi.

SETTINGS.APK

In order to implement the features of the Mirae WiFi, a lot of changes have been introduced to the Settings app.

The following list includes some examples of features that have been removed:

- `auto_rotate`
- System update
- Send feedback
- Debugging menu via build number
- Settings search
- Network settings
- `tether_settings`
- `vpn_settings`
- `forceScan()` in `WifiTracker`
- `android.intent.extra.ringtone.SHOW_MORE_RINGTONES` intent in `DefaultRingtonePreference.java` (prevents changing the ringtone)
- Baseband versions
- Hides the actual battery power usage
- Wifi network notification
- Advanced Wifi settings

The changes to the settings app are not only limited to features for the Mirae WiFi. There are also many changes to remove standard features like changing USB modes or access to the debug menu.

The following sections will describe various changes that added, removed or changed features in the Settings app.

USBMODECHOOSERACTIVITY.JAVA

- Removes USB Modes:
 - Stock image: `DEFAULT_MODES = new int[] { 0, 1, 2, 4, 6, 8, 10 };`
 - Taeyang: `DEFAULT_MODES = new int[] { 0, 1, 2, 10 };`

CHANGES TO DASHBOARDSUMMARY.JAVA

- A lot of changes here compared to the stock image `Settings.apk`.
- Implements `isIccCardReady()` function which checks if `TelephonyManager.getDefault().getSimState(...)` returns either 0 or 1.
- Registers a `BroadcastReceiver()` for the `android.intent.action.SIM_STATE_CHANGED` intent and calls `isIccCardReady()` when it's received.
- `rebuildUI()` adds an additional check for Wifi:
 - `wifiEnabled = this.isIccCardReady() && !WirelessUtils.isAirplaneModeOn((Context)this.getActivity());`
 - At the end, it will enable or disable Wifi based on this check.

ADDITIONS TO THE SETTINGS APP

The Settings app adds `Identifier.java` that does not exist in the stock Android Settings app. It is an interface that loads the native library `libidentify.so` which implements the following JNI function:

- `Java_com_android_settings_deviceinfo_Identifier_getId()`

This calls the internal function `get_devid()` with the first three arguments:

- `Build.SERIAL`
- `IMEI` (returned from `getDeviceId()`²)
- `ro.com.product` from `build.prop` (code calls: `/system/bin/getprop ro.com.product`)
 - Example from Taeyang 8321: `"ro.com.product=1288321"`

The result is an ID for the given device that is unique.

STATUS.JAVA

- Various changes regarding SIM status
- Checks if multiple SIM cards are available (`isMultiSIM`)
- Adds a function that displays the IMEI:

² [https://developer.android.com/reference/android/telephony/TelephonyManager#getDeviceId\(\)](https://developer.android.com/reference/android/telephony/TelephonyManager#getDeviceId())


```

private void setPreferenceValue(final int n) {
    final Phone phone = PhoneFactory.getPhone(n);
    if (phone != null) {

        this.setSummaryText("imei",
phone.getImei());

        this.setSummaryText("imei_sv",
phone.getDeviceSvn());
    }
}

```

WIFICONFIGCONTROLLER.JAVA

The most relevant changes for the Mirae WiFi access are implemented here. The most relevant is the following line that hardcodes the WiFi SSID:

- `this.mSsidView.setText((CharSequence) "PYY1026MIRAE00007")`
-

This is the name of the Mirae WiFi which would be visible to clients in the range of it (if the SSID is not hidden). Furthermore, the authentication methods are restricted to EAP with SIM as the EAP method. This is also hardcoded in the application, the device users have no option to change any of this.

Additionally, various views have been removed from the `WifiConfigController`:

- `com.android.settings:id/security_text`
- `com.android.settings:id/security`
- `com.android.settings:id/security_wfa`
- `com.android.settings:id/wpa_security`
- `com.android.settings:id/wpa_security_wfa`
- `com.android.settings:id/wapi_security`
- `com.android.settings:id/wifi_advanced_toggle`

Generally, the changes remove a lot of features that would allow users to use the devices on other networks or to communicate with other devices over, e.g., USB. It also tries to hide technical information like the WiFi SSID by just removing the views from the Settings app. This makes it especially hard to figure out how the Mirae WiFi works for a casual user of the devices, who are not able to root and access the device.

SIGNATURE SYSTEM CORE

This section describes the inner workings of the signature system and covers aspects of the actual implementation in the native libraries that are utilized by all apps that do signature checks.

The signatures used by this system are cryptographic signatures implemented with asymmetric encryption. These are generally implemented with the RSA algorithm, although the devices also use ECDSA for other tasks (see section Mirae WiFi Access).

Describing the inner workings of these algorithms is not in the scope of this document. However, the basic principle is that asymmetric encryption uses keypairs, where the public part is used to encrypt data and only the private part can decrypt it. In the case of signatures, almost the opposite: private keys create signatures and public keys can be used to verify them. This is relevant to understand the key material that is required on devices to create or just validate signatures.

The signature system supports two types of signatures that are each used for a specific purpose. The main difference is the handling of the private and public keys.

Government Signatures (Type 1)

Also referred to as *NATISIGN* or “nation signatures”, are basically just signatures that are created by a government institution and then added to media files. The devices only have the public part of the keypair to validate those signatures. The intent is to allow the government to approve media files so that they can be consumed on tablet PCs and smartphones. This is especially useful for devices that can access the internal network and download documents or media files from official government servers.

Another use case is to control what apps can be installed on these government approved devices. These signatures are required for APK files, so that only government approved apps can be installed.

Self Signatures (Type 2)

When thinking about controlling media on such devices *NATISIGN* is the go-to signature you would want to have. It gives the government full control of media files and apps that are compatible with these devices. However, the devices are also capable of creating their own media files. Typically, they come with apps like an office suite, notepad, a camera and a microphone, so writing documents or taking pictures/videos must also be possible. For this use-case the *SELSIGN* or “self-signatures” have been implemented.

Each device has an RSA keypair which is used to create signatures for files created on this very device. For example, if you take a picture with the internal camera, the camera app will take care of appending a *SELSIGN* signature to the image on the file system transparently. You as a user won't be able to recognize that this signature exists. Only if you try to open files that do not have a valid signature, an error message will be displayed.

The key comes preinstalled on the device and is most likely not unique for specific devices (see section Signature System Bypassing Software). But the signature actually includes the IMEI and occasionally the IMSI (depending on the device) to ensure that the signature is only valid for a specific device. The signature can also include multiple device identifiers, to make them accessible on multiple devices. However, one would have to extract the private keys from one of these devices to create valid signatures.

Both types of signatures are handled the same when it comes to adding them to files and parsing them. The signatures are just appended to the files, without any special logic that cares about the file type. The type of the signature is appended at the very end in the form of an ASCII string, which is either *NATISIGN* or *SELSIGN*. When verifying the signature of a media file the code basically just seeks to the end of the file and reads the last eight bytes. Based on this data it then decides if it should be handled as a *NATISIGN* or *SELSIGN* signature. In case none of the two strings is there, it assumes the file has not been signed yet. The part where the two signatures are then handled is described in the following sections.

LIBMEDIANATSIGN.SO

This is where the *NATISIGN* signatures are implemented. This section contains some brief implementation details.

DESCRAMBLE_ANDROID()

- Function which decrypts input buffers with a XOR function in `bit_descramble()`.

CHECKPATTERN()

- Reads `/data/legal/sig/pattern.dat`
- Reads 24 bytes at a time

GETKEYFROMNV()

- Reads key from given filename from the path `/data/legal/sig/`.

LIBMEDIASELSIGN.SO

This is where the *SELSIGN* signatures are implemented.

- The key used for selfsigning is read from: `/data/legal/sig/selfsignkey.dat`
- Reads 1424 byte, but only copies 272 into `ns_SignKeyBuf2`:
 - `v12 = MSSGetSignKey(ns_SignKeyBuf, 1424, ns_SignKeyBuf2, 272u);`
- According to the symbols, the following well known crypto is used:
 - RSA2048
 - SHA256
 - Rijndael256
- Have these algorithms been modified?
 - Sboxes of Rijndael?
 - Constants of SHA256?
- The encrypted part of signatures is 520 byte
- 16 bytes at the end are

- 4 byte length
- 4 NULL bytes
- 8 bytes “SELSIGN” string

LIBMEDIAEXT.SO

This is a simple library that implements mime type checks for media files. It provides two JNI functions:

- `isFileExt()`: Check if a given file’s mime type is allowed
- `saveExtFile()`: Add additional mime types to the database

The implementation is very simple. It just reads the first 16 bytes of a given file and checks if these 16 bytes are present in the database stored at `/data/legal/sig/ext.dat`. The file just contains a list of 16 byte strings with no particular format. The following screenshot shows the contents of the default database file:

```
00000000: 504b 0304 0000 0000 0000 0000 0000 0000 PK.....
00000010: 0000 0000 0000 0000 4156 4920 4c49 5354 .....AVI LIST
00000020: 0000 01ba 0000 0000 0001 0189 c3f8 0000 .....
00000030: 0000 01ba 2100 0100 0100 0000 0000 0100 ....!.....
00000040: 0000 0000 6674 7970 3367 7000 0000 0000 ....ftyp3gp....
00000050: 3026 b275 8e66 cf11 a6d9 00aa 0062 ce6c 0&.u.f.....b.l
00000060: 464c 5601 0500 0000 0000 0000 0000 0000 FLV.....
00000070: 0000 0000 6674 7970 6973 6f6d 0000 0000 ....ftypisom...
00000080: 3026 b275 8e66 cf11 a6d9 00aa 0062 ce6c 0&.u.f.....b.l
00000090: 0000 0000 0000 0000 6d61 7472 6f73 6b61 .....matroska
000000a0: 4657 5306 0000 0000 7000 0dc0 0000 b400 FWS.....p.....
000000b0: 0000 0000 6674 7970 7174 0000 0000 0000 ....ftypqt.....
000000c0: 0000 0000 6674 7970 6d70 3432 0000 0000 ....ftypmp42....
000000d0: 0000 0000 6674 7970 6d70 3431 0000 0000 ....ftypmp41....
000000e0: 2e52 4d46 0000 0000 0000 0000 0000 0000 .RMF.....
000000f0: 1a45 dfa3 0100 0000 0000 001f 4286 8101 .E.....B...
00000100: 1f07 003f f979 7979 ff00 ff00 ff00 ff00 ...?.yyy.....
00000110: b7d8 0020 3749 da11 a64e 0007 e95e ad8d ... 7I...N...^..
00000120: 4740 1110 0042 f025 0001 c100 0000 01ff G@...B.%.
00000130: 4f67 6753 0000 0000 0000 0000 0000 0000 OggS.....
00000140: 0000 0000 6674 7970 6d6d 7034 0000 0000 ....ftypmmp4....
00000150: 0000 0000 6674 7970 3367 3261 0000 0000 ....ftyp3g2a....
00000160: 0000 0000 6674 7970 4d34 4120 0000 0000 ....ftypM4A ....
00000170: 0000 0000 6674 7970 4d34 5620 0000 0000 ....ftypM4V ....
00000180: 2321 414d 5200 0000 0000 0000 0000 0000 #!AMR.....
00000190: 0000 0000 6674 7970 4d34 4100 0000 0000 ....ftypM4A.....
```

Each line represents a mime type, starting with Zip, AVI³ and so on. The majority are media related mime types like video and audio files.

This is the only check that this library provides. There is no check for file extensions.

³ https://en.wikipedia.org/wiki/Audio_Video_Interleave

TECHNICAL SCOPE OF DEVICES AND LIBRARIES

This section describes the analysis that has been done during the project and discusses some of the technical similarities and differences between the devices. This information helps to create a baseline to compare different hardware and understand how changes in software and firmware relate to one another.

The following sections describe the devices and software libraries that were available during the analysis. The analysis was conducted on the device dumps that were available, which does not necessarily require access to the physical device.

With a few exceptions, dumps were created for all physically available devices.

AVAILABLE DEVICES

The following list contains all devices where full device dumps are available.

Smartphones:

- AP121
- Pyongyang 2406
- Pyongyang 2407

Tablet PCs:

- Taeyang 8321
- Woolim
- Ryongaksan

The AP121 smartphone and all the tablet PCs listed above are also physically available.

Currently, there are two exceptions for devices that are physically available, but we have no full dumps (yet):

- “Achim” tablet from 2017
- Pyongyang 2425 (still work in progress)

For these devices no deeper analysis has been performed so far. However, the Achim tablet shares various properties with the Woolim and Ryongaksan tablets. Also the Pyongyang 2425 phone is comparable to the Taeyang tablet PC, as it basically shares a similar platform. But it has a newer Android version that has not yet been analyzed on any North Korean smartphone.

RELEVANT DEVICE PROPERTIES

For the sake of simplicity, we are comparing the devices by some software related properties that are relevant for the signature system and other custom features. These are:

- Build dates of the devices
- Release dates of devices (in case these are available/known)
- Android version
- Kernel version
- Platform

The following table shows these properties for the devices in scope.

Device	Build Date	Android	Kernel	Platform
AP121	Thu Dec 25 18:09:48 KST 2014	4.2.1	-	MT6589
Pyongyang 2406	Wed Jul 23 18:17:20 CST 2014	4.2.2	-	MT6572
Pyongyang 2407	Mon Dec 29 12:38:55 CST 2014	4.2.2	3.4.5	MT6582
Taeyang (대양) 8321	Jul 28 16:13:34 KST 2018	6 (security patch: 2016-09-05)	3.18.19 (15-01-2018)	MT6580
Woolim	Thu Sep 10 18:26:32 EDT 2015	4.4.2	3.4.39	A33
Ryongaksan (룡악산)	2017.08.03 15:28:05 KST	4.4.2	3.4.39	A33

KERNEL INFO

The following table shows more comprehensive information about the kernel version and build properties of device dumps where this was available. It gives some basic information about the hostname it was built on, software versions that were used and the build date of the kernel. These are brief indicators for when the devices have been built and when the software has been installed/updated last.

This table only includes the devices where the information was accessible on the device dump. Other devices did not include the full information on the device dump.

Device	Kernel Info
Pyongyang 2407	Kernel: Linux version 3.4.5 (znsj@znsj-soft01) (gcc version 4.6.x-google 20120106 (prerelease) (GCC)) #1 SMP PREEMPT Mon Dec 29 12:37:11 CST 2014
Woolim	Linux version 3.4.39 (root@rainbow-SUN-SERVER-X4-2) (gcc version 4.6.3 20120201 (prerelease) (crosstool-NG linaro-1.13.1-2012.02-20120222 - Linaro GCC 2012.02)) #1 SMP PREEMPT Wed Nov 4 05:59:45 EST 2015

Device	Kernel Info
Woolim	Kernel: Linux version 3.4.39 (jml@jml) (gcc version 4.6.3 20120201 (prerelease) (crosstool-NG linaro-1.13.1-2012.02-20120222 - Linaro GCC 2012.02)) #709 SMP PREEMPT Wed Aug 2 15:16:38 CST 2017

BUILD PROPERTIES

The following table shows properties from the `build.prop`⁴ file of each device. These include information about when the Android image has been built, the base image that has been used to build it, the build system and the architecture version of the device. The build dates and base images are interesting, as it allows us to determine when the device has been finalized by the North Korean developers and the base images that they used.

The build description and fingerprint contains information that can allow you to identify the actual base image and search for it online. This has been done for the Taeyang tablet and the corresponding stock image was available for download.

Device	Properties
AP121	<p>ro.build.date=Thu Dec 25 18:09:48 KST 2014</p> <p>ro.build.description=bird89_wet_a_jb2-user 4.2.1 JOP40D eng.ubuntu-61.1419498478 test-keys</p> <p>ro.build.fingerprint=KPTCBS/bird89_wet_a_jb2/bird89_wet_a_jb2:4.2.1/JOP40D/1419498478:user/test-keys</p> <p>ro.mediatek.version.branch=ALPS.JB2.MPV1.2</p>
Pyongyang 2406	<p>ro.build.date=Wed Jul 23 18:17:20 CST 2014</p> <p>ro.build.description=gionee72_wet_jb3-eng 4.2.2 JDQ39 eng.android.1406110529 test-keys</p> <p>ro.build.fingerprint=alps/gionee72_wet_jb3/gionee72_wet_jb3:4.2.2/JDQ39/1406110529:eng/test-keys</p> <p>ro.mediatek.version.release=ALPS.JB3.MPV1</p>

⁴ <https://www.droidwiki.org/wiki/Build.prop>

Pyongyang 2407	ro.build.date =Mon Dec 29 12:38:55 CST 2014 ro.build.description =gionee82_wet_jb5-eng 4.2.2 JDQ39 eng.znsj.1419827849 test-keys ro.build.fingerprint =alps/gionee82_wet_jb5/gionee82_wet_jb5:4.2.2/JDQ39/1419827849:eng/test-keys ro.mediatek.version.release =ALPS.JB5.MPV1.6
Taeyang 8321	ro.build.date =Sat Jul 28 16:13:34 KST 2018 ro.build.description =full_along8321_tb_m_706m-user 6.0 MRA58K 1532760902 test-keys ro.build.fingerprint =alps/full_along8321_tb_m_706m/along8321_tb_m_706m:6.0/MRA58K/1532760902:user/test-keys ro.mediatek.version.release =alps-mp-m0.mp1-V2.52_along8321.tb.m_P33
Woolim	ro.build.date =Thu Sep 10 18:26:32 EDT 2015 ro.build.description =ulrim_pic-eng 4.4.2 KVT49L eng.root.20150910.181849 dev-keys ro.build.fingerprint =PIC/ulrim_pic/ulrim_pic:4.4.2/KVT49L/eng.root.20150910.181849:eng/dev-keys
Ryongaksan	ro.build.date =2017.08.03 15:28:05 KST ro.build.description =astar_y3-eng 4.4.2 KVT49L 20170803 test-keys ro.build.fingerprint =Allwinner/astar_y3/astar-y3:4.4.2/KVT49L/20170803:eng/test-keys

AVAILABLE VERSIONS OF THE SIGNATURE SYSTEM

The following table shows all the available libraries and SHA256 sums of the available device dumps. It allows one to identify which devices use the exact same versions/files of the signature system. Identical sums mean there are no differences at all.

Device Library	SHA256
pyp_2406/libmediaselfsign.so	5ce3bbd3c0c16b7e3c1b70f7dd5d0e278a3c abe4f5c248ec7f61be893306894
pyp_2406/libmediaselfsign.so	8bbb5f92d9c2efa10db7db056a5c10249621 35ed42574d1ba3aa976455083eeb
pyp_2407/libmedianatsign.so	5ce3bbd3c0c16b7e3c1b70f7dd5d0e278a3c abe4f5c248ec7f61be893306894a
pyp_2407/libmedianatsign.so	8bbb5f92d9c2efa10db7db056a5c10249621 35ed42574d1ba3aa976455083eeb
taeyang_8321/libmediaselfsign.so	8715e0686c6d669dc44e62e3fe819b5157a7 3e300d0dd986e282c7bdfc500b5f
taeyang_8321/libmediaselfsign.so	7c6950f368cc352ff7e7df29c7092103ddbde 54a1add0b73c573e43432004923
woolim/libmediaselfsign.so	344782301025b3d8806f8ee882f4a31222f 24a984dd665168e969efb8864a5f
woolim/libmediaselfsign.so	3b5df463bb07cb3da53f26d8d18c3650f76a 8ea76b0af02c5009d7610374abfb

ryongaksan/libmedianatsign.so	344782301025b3d8806f8ee882f4a31222f 24a984dd665168e969efb8864a5f
ryongaksan/libmediaselfsign.so	3b5df463bb07cb3da53f26d8d18c3650f76a 8ea76b0af02c5009d7610374abfb
ap121/libmedianatsign.so	344782301025b3d8806f8ee882f4a31222f 24a984dd665168e969efb8864a5f
ap121/libmediaselfsign.so	2ff1c9d3e6cb5ecb72b16b1f056dd828f52e0 02ced3812c0b5bd8396ecea21da

Just from the identical hashes we can conclude several interesting points:

PYONGYANG 2406 & 2407 USE THE EXACT SAME SIGNATURE SYSTEM

This is expected, as those phones are also built in the same year by the same manufacturer. Depending on the actual release date, most of the device modifications should be similar on both devices.

SIMILARITIES WITH STOCK IMAGES

One step of trying to identify modifications by North Korean developers was to diff device dumps with stock images. These were searched on the Internet with information extracted from the devices, especially from the `build.prop` files.

This section describes the process of how the dump of the Taeyang 8321 device has been compared to the following stock image we have found for the device:

- MT6580__alps__Nomi_C070010__elink8321_tb_m__6.0__alps-mp-m0.mp1-V2.52_elink8321.tb.m_P10.rar

There are various things we could learn from this analysis step:

Added Apps

Added apps are the most interesting ones as these are potentially developed by North Korean developers. We have identified that most of the added apps are actually self-developed. The following table shows the apps have been added to the device:

Name	Category	Description
AdobeReader	PDF reader	Implements signature checks in opened PDF files.
CM_Lite		
KorealIME_Star	Custom App	
LiveWallpaper1	System App	
LiveWallpaper2	System App	
LiveWallpaper3	System App	
LiveWallpaper4	System App	
LiveWallpaper5	System App	
LiveWallpaper6	System App	
NotesPad	Custom App	
OceanKingOffice12	Custom App	Office suite with implemented signature checks.
Pinyin_IME_v3_2_0	Custom App	
Provision	System App	
webview	System App	
Ocean_Launcher_V02_NoApp_Q8H	Custom App	
OneTimeInitializer	System App	
PackageInstaller	System App	
RedFlag	Custom App	The core of the signature system.
SmartCardService	Custom App	Implementation for accessing Mirae WiFi.
TraceViewer	Custom App	Frontend for functionality of RedFlag app.

Modified Apps

There are multiple reasons why apps have been modified. Some have simply been translated to Korean (e.g. Angry Birds⁵), including text in the apps but also images that contain text. It is assumed that these are popular apps that the developers wanted to make accessible on these devices but weren't available in Korean.

⁵ https://en.wikipedia.org/wiki/Angry_Birds

Another reason for modifying apps is to add compatibility with the signature system and other custom features of the North Korean devices. These include e.g. signature checks for media files in the browser app or the Adobe PDF reader app.

The following list shows the apps that have been modified. However, no further classification of the actual modifications has been done:

Name	Category	Description
Bluetooth	System App	Implements signature checks for files that are shared via Bluetooth. Excludes signature checks for vCard files (.vcf or .vcs extensions).
Camera	System App	Implements adding signatures to image and video files.
FileManager	System App	Implements signature checks for various file system tasks like file opening.
Gallery2	Media Viewer	Implements signature checks, significant changes to ensure everything is working with the signature system.
HTMLViewer	HTML Viewer	Implements check for signatures when opening files.
MtkBrowser	Browser	Implements signature checks for downloaded and opened files (include file:/// URLs).
Music	Music Player	Implements signature checks.
SoundRecorder	Sound Recorder	Implements adding signatures to recorded audio files.
DownloadProvider	System App	Implements signature checks.

ExternalStorageProvider	System App	Implements signature checks.
Settings	System App	Various changes that disable features like direct WiFi or USB option access.
SystemUI	System App	Implements signature checks.

One particularly interesting example for modified apps is the Settings app. Various changes are described in section Settings.apk.

Stock Image Content

Besides the apps it allows to also identify all the other files that might be identical on the stock image. These include e.g., configuration files or data that is preinstalled for system applications. All of the files that are identical have been excluded from the analysis.

The following table shows the apps that have not been modified:

Name	Category
BasicDreams	BasicDreams
CalendarImporter	System App
DeskClock	System App
DocumentsUI	System App
DownloadProviderUI	System App
DrmProvider	System App
KeyChain	System App
LiveWallpapers	System App
MtkCalendar	Calendar
MusicFX	System App
Settings	System App
PacProcessor	System App

PhotoTable	System App
UserDictionaryProvider	
BackupRestoreConfirmation	System App
CalendarProvider	System App
CarrierConfig	System App
DefaultContainerService	System App

Besides the apps that are available on either the North Korean devices dumps or the stock image, various core parts of the Android operating system have been analyzed. The observations are discussed in the following section.

DIFFERENCES IN ANDROID VERSIONS

Over the past years we've seen devices with different Android versions. In this section we are discussing potential differences in these versions. We focus on everything related to North Korean software implementations rather than the stock operating system.

The first devices we've analyzed in the past, including smartphones and tablet PCs, were all based on the same AllWinner architecture running (almost) the same Android Version 4.x. Even devices (that we know of) that were made in ~2017 were running this version (e.g., the Ryongaksan tablet PC). Other devices, from other vendors, then introduced MediaTek-based platforms, with newer Android versions.

Devices/Vendors with AllWinner chips:

- Woolim
- Ryongaksan
- "Achim" tablet from 2017

Devices/Vendors with MediaTek chips:

- Pyongyangphone 2406/2407
- Pyongyangphone 2425
- Taeyang 8321 tablet PC

LIMITATIONS BASED ON THE PLATFORM

The used AllWinner chips limited the supported Android versions mostly to Android 4. Based on the devices we've analyzed over the course of multiple years; this affects devices from 2013/2014 up to devices that were released in 2017 (it may even affect newer devices we've not encountered so far). However, in the year 2017 Android 8 was released, so the North Korean devices that were based on the AllWinner SoCs were shipped with obsolete Android versions for most of their selling time.

MediaTek-based chips we encountered in North Korean devices did not come with the same limitations as the AllWinner SoCs. Therefore, these devices were already shipped with newer versions of Android. It is typically not the latest version available on conventional phones outside of North Korea, but it is not too far behind. The newer phones like Pyongyang come with Android 8 or even Android 9 already.

So generally, it is assumed that the old Android versions are not a result of a decision by the North Korean developers to use these. These are rather the consequences of relying on cheap products from vendors that come with these limitations by default. The problem is then most likely that the developers cannot use newer Android versions even if they intend to do so.

The only way to ship devices with newer versions is to switch to other platforms, like some vendors did with the MediaTek chips. This trend will most likely continue in the future, as most of the newer known devices are all based on MediaTek chips.

OS FEATURE UTILIZATION IN NEWER ANDROID VERSIONS

This section highlights added or improved measures on the most recent implementation we've analyzed (Taeyang 8321).

TAEYANG BROWSER CHECK

On Woolim it was possible to bypass the signature system just by using apps that allowed to view media files but did not implement signature checks. One example was the default Android browser, which allowed you to view media files via `file:///` URLs. Due to the older Android version on Woolim, it was possible to access files from an external SD card via these URLs. However, in Taeyang this is not possible anymore. The following code shows the `loadUrl()` function, that checks if the signature system should be enabled, and if so, checks if the URL isn't a `http:///...` URL:

```
public void loadUrl(String checkAndTrimUrl, final Map<String, String> map){
    if (SystemProperties.get("ro.nation.sign").equals("1") &&
        !URLUtil.isNetworkUrl(checkAndTrimUrl)) {
        Toast.makeText(this.mContext, 2131493014, 0).show();
        return;
    }
    [...]
```

This prevents using `file:///` URLs in the default Android browser. Additionally, when downloading a file, the browser now also implements a signature check during the download process:

```

public static void onDownloadStartNoStream(Activity activity, String s, String s1, String
s2, String s3, String s4, boolean flag, long l){
    mDownloadCompleteReceiver = new BroadcastReceiver(){
        public void onReceive(Context context, Intent intent){
            if(SystemProperties.get("ro.nation.sign").equals("1") &&
Interface.isLegalFile(DownloadHandler._2D_get0()) != 1)
                (new File(DownloadHandler._2D_get0())).delete();
        }
    };
    [...]
}

```

This not only prevents loading of non-signed files from the local storage or external SD cards, but also when they are downloaded from any website. This becomes more relevant with projects like the Mirae WiFi Access, where tablet PCs and other devices will be interconnected in the future. In these scenarios files could be shared not only via peer-to-peer connections like Bluetooth, but also by universally accessible systems on this WiFi network via services like web servers.

IMPLEMENTING FEATURES IN ART

Android KitKat 4.4 introduced a new runtime as an alternative to Dalvik⁶ in 2013. The new Android Runtime (ART) is the default runtime since Android Lollipop 5.0 (2014) and it uses ahead-of-time compilation. The advantage compared to the previous runtime is that there is a central implementation of the core Android APIs that can be used by all apps on a device. This was not used in older devices like Woolim (although the Android version would have it available), but is actually used on Taeyang.

Typically, every app that should do signature checks, e. g., an image/video viewer or text editor, had to implement the signature checks on older devices like Woolim. Each of these apps had to implement calls to functions from native libraries that checked the signatures. These libraries came preinstalled on these devices and were called `libmedianatsign.so` and `libmediaselfsign.so`. On newer devices like the Taeyang 8321, which uses ART, these checks are implemented in `boot.oat` (`/system/framework/arm/boot.oat`). The explicit signature checks still use the previous native libraries but with simpler interfaces. Additional implicit measures are implemented by intercepting API calls. This new implementation is way more effective than the previous implementations, because it is more convenient to implement and is harder to bypass. It allows to implement checks at a lower level in the core Android APIs without the calling app noticing it. There is also no way for the apps to prevent these implicit checks if they rely on the core APIs.

⁶ [https://en.wikipedia.org/wiki/Dalvik_\(software\)](https://en.wikipedia.org/wiki/Dalvik_(software))

This is also a common way of implementing rootkits on Android⁷. It is assumed that the implementation is based on the same or similar research. The intended goal is also comparable to a rootkit as the custom features try to monitor and also prevent a user's actions.

Apps that implement the newer version of the signature system load the `com.Legal.Java.Interface` in the Java code and call a function called `isLegalFile()`. Anything related to `libmedianatsign.so` and `libmediaselfsign.so` then happens transparently in ART. The actual code that is executed is very similar to the older (explicit) implementations. However, they are not implemented in every app that does signature checks but in a central place where the core APIs are implemented.

The following list includes apps that use this new signature system interface on the Taeyang tablet:

- SoundRecorder.apk
- Gallery2.apk
- Music.apk
- HTMLViewer.apk
- FileManager.apk
- MtkBrowser.apk
- Bluetooth.apk
- Camera.apk
- AdobeReader.apk
- OceanKingOffice12.apk
- SystemUI.apk
- DownloadProvider.apk
- PackageInstaller.apk

The following code snippet shows an example how the signature check is implemented in the PackageInstaller app. This is a default Android system app that installs APK files.

```
[...]
if (SystemProperties.get("ro.nation.sign").equals("1")) {
    int var2 = Interface.isApkLegalFile(this.mPackageURI.getPath());
    Log.d("PackageInstaller", "PackageInstaller -- isApkLegalFile onClick
    result = " + var2);
    if (var2 != 1) {
        this.showDialogInner(7);
        this.finish();
        return;
    }
    Log.d("PackageInstaller", "PackageInstaller -- isApkLegalFile onClick sign
    complete!");
}
[...]
```

⁷ <https://securityintelligence.com/hiding-behind-android-runtime/>

The code first checks the `ro.nation.sign` property, which is located in the `build.prop` file. This is a property switch that allows to enable or disable the signature checks in various apps. However, it does not allow to fully disable the signature system because it is up to each app to check this property. Not all apps or other custom code outside of apps of the signature system implement such checks. This especially applies to implicit signature checks in the core API implementations.

If the signature checks are enabled, it continues to call the `isApkLegalFile()` function. This is part of the actual signature system that handles all the signature checks and file validation.

The checks shown above are implemented in the `PackageInstaller`'s `onCreate()` and `onClick()` functions. These are called when an instance of the `PackageInstaller` is created and also every time the app installation process is triggered. This happens, e.g., when a user tries to open an APK file.

ART is located in a file called `boot.oat` which can be converted to DEX (`oat2dex`⁸) files. The implementation is then available in the two framework files `framework.dex` and `framework-classes2.dex`. Anything related to the signature system is implemented in the latter.

For further analysis, the DEX files have been converted to JAR files via `dex2jar`⁹.

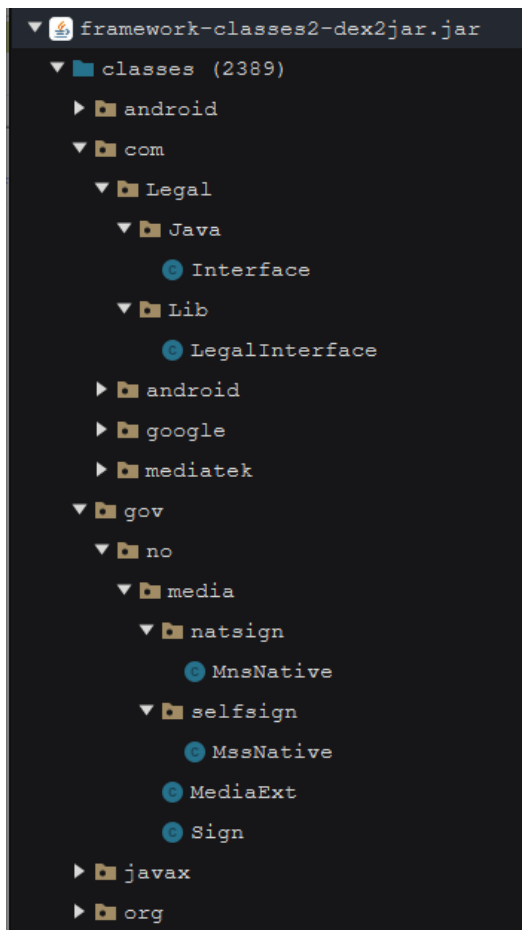
FRAMEWORK-CLASSES2.JAR

This contains the central implementation of the signature system interfaces. These are just the external interfaces that should be used in the Java code of all the apps. The actual core of the signature system is implemented in native libraries (see *section Signature System Core*).

The following image shows the structure of these files with all the relevant interfaces in the `com.Legal` and `gov.no.media` classes.

⁸ <https://github.com/testwhat/SmaliEx>

⁹ <https://github.com/pxb1988/dex2jar>



The main interface that is exposed to and used by all the apps is implemented in `com.Legal.Java`. It provides mostly the same functions as the previous versions of the signature system did in the per-app implementations. This is basically a frontend for the `gov.no.media.*` classes and an additional new interface `gov.no.media.Sign`. It provides the following functions:

- `checkMMSFile()`
- `generateTempApk()`
- `getLegalInfoSize()`
- `isApkLegalFile()`
- `isLegalFile()`
- `isMMSFile()`
- `isMagicCorrect()`
- `isMtpFile()`
- `isOtgFile()`
- `isTempApkFile()`
- `isVcfFile()`
- `isVcsFile()`
- `isVideoLegalFile()`
- `saveLegalFile()`
- `setMMSFile()`

Anything in the `gov.no.media.natsign.MnsNative` and `gov.no.media.selfsign.MssNative` packages is implemented via JNI¹⁰ in native functions¹¹. This is similar to the previous versions of the signature system.

Compared to the older versions of the signature systems it adds the `com.Legal.Lib.LegalInterface` class which uses `libLegalInterface.so` as an additional native library. The following JNI functions are provided by the `LegalInterface`:

```
package com.Legal.Lib;

public class LegalInterface
{
    static {
        System.loadLibrary("LegalInterface");
    }

    public static native int checkLegalFile(final String p0);

    public static native int generateTempApk(final String p0, final String p1,
final String p2);

    public static native int getLegalInfoSize(final String p0);

    public static native int isLegalFile(final String p0);

    public static native int isMagic(final String p0);

    public static native int isSended(final String p0);

    public static native int isTempApkFile(final String p0, final String p1);

    public static native int saveLegalFile(final String p0);

    public static native int setMMSInfo(final String p0);
}
```

The actual implementation of these functions is located in the native library `libLegalInterface.so`.

This might be the new implementation of the signature system and the successor to the previous `libmedia*` libraries. However, all of the analyzed apps still use the older implementation, which might be due to compatibility reasons. This is similar to support for the Red Star OS watermark implementation that is also available on the tablet PCs. The reason is most likely to be compatible with various different versions of the North Korean controlling mechanisms.


¹⁰ https://en.wikipedia.org/wiki/Java_Native_Interface

¹¹ <https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html>

FRAMEWORK.JAR

This is the place where the Android APIs are implemented. This has been decompiled and compared to a decompiled version of the stock image (see section TODO). The key parts are the ones that use the interfaces provided by the previously described `framework-classes2.jar`. Any occurrence of these interface calls has been added to implement signature checks and adding signatures to files for various core functionality (taking photos, creating documents, etc.).

Besides integration of the signature system, various other changes have been observed. These often disable default Android functionality to prevent users from changing certain configurations or use builtin features. The following example shows one small change in the `WifiEnterpriseConfig` class, where the password string has been removed from the `toString()` function:



The image shows a side-by-side comparison of the `toString()` method implementation in the `WifiEnterpriseConfig` class. On the left, the implementation from Taeyang is shown, where the password field is explicitly removed. On the right, the implementation from the stock image is shown, where the password field is included in the output string.

```
#Override
public String toString() {
    final StringBuffer sb = new StringBuffer();
    for (final String key : this.mFields.keySet()) {
        String str2;
        if ("password".equals(key)) {
            str2 = "XXXXXXXXXX";
        } else {
            str2 = this.mFields.get(key);
        }
        sb.append(key).append(" ").append(str2).append("\n");
    }
    return sb.toString();
}
```

```
#Override
public String toString() {
    final StringBuffer sb = new StringBuffer();
    for (final String s : this.mFields.keySet()) {
        sb.append(s).append(" ").append(this.mFields.get(s)).append("\n");
    }
    return sb.toString();
}
```

The left side shows the implementation in Taeyang and the implementation on the stock image on the right. This function is meant to show the Enterprise WiFi configuration as a string. The default implementation on the stock image just adds all available fields, including the configured password. However, Taeyang explicitly removes the password field so that it is not visible to the user.

The following section describes one major change in the Android API: adding signature checks to MTP, which is implemented in this framework file.

SIGNATURE CHECKS IN MEDIA TRANSFER PROTOCOL (MTP)

MTP is a USB mode in Android that allows the transfer of data between devices¹². It is used when connecting an Android device to a computer to copy data to/from devices. Typically, this mode has to be activated on the device, but devices may choose to use it as the default mode when plugging in a USB cable.

MTP does not open any files, it just transfers them from one device to another. However, on Taeyang the core implementation `android.mtp.MtpDatabase` in the ART has been modified to do signature checks. The following function `getObjectFilePath()` from the MTP implementation shows some parts of the added code, which checks if the given filename has a valid signature:

¹² <https://developer.android.com/reference/android/mtp/package-summary>

```

private int getObjectFilePath(final int n, final char[] array, final long[] array2) {
    Log.d("MtpDatabase", "getObjectFilePath handle = " + Integer.toHexString(n));
    [...]
    switch (Interface.isLegalFile(string)) {
        case 3: {
            cursor2 = query;
            cursor = query;
            this.deleteIllegalFile(this.mContext, string);
            break;
        }
        [...]
    }
}

```

In case the file does not have a valid signature, it will immediately be deleted from the device instead of just printing a warning message. This is different from older implementations we've analyzed like the Woolim tablet PC. On those devices files won't be deleted, the apps will just refuse to open them.

CUSTOM APP ANALYSIS

This section briefly describes the analysis of some interesting apps of Taeyang. These are implemented by North Korean developers and are not just copied from stock images.

OCEAN_LAUNCHER-V02_NOAPP_Q8H.APK

This is a central app that is pre-installed on the Taeyang tablet. It has some very interesting implementation properties. One feature is the implementation of a check for the use of wallpaper. The implementation of the features is somehow hidden in a file called `wall.jpg` which is part of the resources in the APK file. Based on the location (`resources/raw`) and filename it seems like the developers intended to hide the fact that this is actually a native library which is loaded by the following function:

```
private void loadLibrary() {  
    try {  
        InputStream var1 = this.mContext.getResources().  
            openRawResource(2131099649);  
  
        byte[] var2 = new byte[var1.available()];  
  
        var1.read(var2);  
  
        Context var3 = this.mContext;  
  
        Context var4 = this.mContext;  
  
        FileOutputStream var6 = var3.openFileOutput("wall.dat", 0);  
  
        var6.write(var2);  
  
        var6.close();  
  
        var1.close();  
  
        System.load(this.mContext.getFilesDir().getAbsolutePath() + "/wall.  
            dat");  
    } catch (Exception var5) {  
        var5.printStackTrace();  
    }  
}
```

This function copies `wall.jpg` into `wall.dat` and loads it via `System.load()`. This will then make the following JNI functions available in the app:

- `Java_com_padandroid_aplus_service_AplusServiceManager_setLicense()`
 - Gets data via function arguments and initializes a buffer with it.
- `Java_com_padandroid_aplus_service_AplusServiceManager_loadStamp()`
 - Calls `have_license()`, which just checks if the initialized buffer contains some given strings. Compare it to data from `stamp.png` file from resources.
 - Bitmap must be 320x200 and has to have a specific format set in the image header. No fancy checks, pretty simple.

These functions are used in a so-called “Aplus Service” (`com.padandroid.aplus`) which is implemented in the Java code of the app. This service seems to check the given image file when the wallpaper is changed. In case it does not meet the requirements in the checking functions, the device will refuse to change the wallpaper.