REVERSE ENGINEERING LG MLT ON ANDROID

Mike Williamson
Calgary Police Service Digital Forensics Team
pol4439@calgarypolice.ca

ABSTRACT

MLT (also MPT) is logging software for the Android operating system whose presence was identified in the course of a forensic examination of an LG cell phone by the author of this paper. MLT was created by LG and is specific to LG devices. MLT logs software and hardware events in SQLite databases. These logs are forensically valuable because they persist beyond a factory reset (or "wipe") of the device. Even when a device hasn't been reset, the information it contains could be useful in corroborating findings from other areas of the device. The challenge with MLT data is that there is no documentation or even descriptive column names to explain what information each database table represents. Internet research showed that no published work existed documenting the purpose and source of each table and column.

The purpose of this research therefore was to identify a means of sourcing the database files, tables and columns found in the various databases created by MLT. This was accomplished by reverse engineering the MLT application obtained directly from the cell phone being examined. However, with further collaboration it appears that MLT is ubiquitous for LG devices running Android. It is both preinstalled and enabled by default. The findings of this research is of potential value to anyone conducting a forensic analysis of an LG device, especially one that has been factory reset.

1 INTRODUCTION

MLT, like most Android applications, is found on LG devices as an APK file (Android Package Kit - see section 2.1). MLT runs almost entirely in the background and listens for system events (known in Android as 'broadcasts'. It saves these events to several SQLite databases, occasionally duplicating information. The reason for having multiple databases with duplicitous data is presently unknown. The SQLite databases are stored on a separate partition, aptly labelled 'mpt'. This separate partition is the reason the logs are not removed in a factory reset operation.

The tables and field names are numeric (as an example: Table 't123' with columns 'f001', 'f002', and 'f003'). No explanation or documentation is provided as to the purpose of each table or column. In some cases, the purpose of a column is obvious by looking at the data found within it. For obvious reasons this type of assertion is far too speculative and would be insufficient for the purposes of court testimony, where the standard of proof is much higher.

A non-exhaustive, plain language list of types of events tracked by MLT includes: operating system version upgrades, app installation, update and deletion, device boot-up and charging events, Screen usage, Bluetooth usage, WiFi usage, GPS usage, cell tower data, error logging. Log entries are all datestamped.

Discussion on this subject on the internet was mostly non-existent. There was some forum discussion that notes the existence of MLT including suggestions that LG is responsible for installing "spyware" on their devices which tracks device usage without their knowledge or consent. Some guides exist on how to permanently disable MLT to improve privacy, device performance, or both.

As previously mentioned, the catalyst for this research was the discovery of the MLT databases during forensic examination of a wiped device. The majority of the research will be conducted on the MLT sample obtained from that device, which happens to be MLT version 2.5.4. There are numerous versions of MLT however ranging from 2.4.1 to 2.5.8 and possibly much more. There are some distinct differences between the versions, including the names of the database files, the number of tables within them, and so on, but the same approach and methodology should be the same for various iterations of the software.

2 BACKGROUND

In order to explain the inner workings of MLT, first we will review how Android binaries are stored on the device and what happens when they are executed on the device.

2.1 APK FILES

APK files are compressed container files that contain: compiled code, user interface files, application resources, metadata and code signing certificates. When a user downloads a new application from the Google Play store, they are likely downloading an APK, which is how the app will be stored on the system. MLT is not available on the Google Play store however as it ships with LG's flavor of Android. There are public tools available (namely *apktool*) which will extract the contents of an APK file and make them more accessible to reverse engineering efforts. Android apps are usually coded in Java, and then compiled into machine code before being packed into the APK.

Prior to Android 5.0, a virtual machine known as 'dalvik' handled all code execution. This was replaced in Android 5.0 with Android Runtime (ART). Despite the change in runtime environments, the terminology for machine code in Android seems to remain 'dalvik bytecode'. Dalvik bytecode is machine-code, programmatic instructions for a theoretical (virtual) processor.

When an APK file is run the Zygote process - a skeleton process with all native libraries pre-loaded - creates a copy of itself within which the new process will be run. The classes.dex file (dalvik bytecode) is converted into native machine code which is stored on the device as an OAT file.

In addition to extracting all the files within an APK, *apktool* also disassembles the classes.dex file using a process called baksmali (Icelandic for disassembler). The resuling .smali files are a slightly more human readable form of dalvik bytecode. Interpretation of these smali files is the primary source of information for this research. We will also use an application called Bytecode Viewer which attempts to convert dalvik bytecode back into Java, which can be much easier to make sense of than smali.

2.1.1 Later Versions of MLT

Regarding later versions of MLT, such as 2.5.7; In some cases, the MLT APK file container does not contain the classes.dex file. It is not totally clear what might cause this state, whether it is intentional for anti-reversing purposes, or if it is purely for system optimization (if the classes.dex file isn't inside the APK, it doesn't need to be decompressed).

Some of the naming schemes does change with later versions of MLT - in the case of 2.5.7, the only log file is called LDB_MainData.db but still resides on the \mpt\ partition, as with earlier versions.

2.2 SMALI

Smali is a language that exists halfway between machine code and human authored code. It is open source, authored by JesusFreke, who describes it as follows:

"smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android's Java VM implementation. The syntax is loosely based on Jasmin's/dedexer's syntax, and supports the full functionality of the dex format (annotations, debug info, line info, etc.)

The names "smali" and "baksmali" are the Icelandic equivalents of "assembler" and "disassembler" respectively. Why Icelandic you ask? Because dalvik was named for an Icelandic fishing village." [3]

There remains a significant learning curve when it comes to being able to understand smali. Those with an understanding of ASM whether as a programmer, reverse engineer, or both will have an advantage. There are a number of websites that cover the basics effectively, the links to which are provided in the references section of this paper.

One of the most important things about reading smali is understanding that primitive variables, known in languages such as C++ as bool, void, byte, short, char, int, long, float, and double are represented by single capital letters as follows:

Z	bool	V	void
B	byte	S	short
C	char	I	int
J	long	F	float
D	double		

An example function taking two integers (I) and a boolean (Z) as parameters and returning an integer (I) for example, would be shown as:

```
.method public someFunction(IIZ)I
```

Given that dalvik bytecode is derived from its original composition in Java, which is object-oriented and class-based, classes is also a requirement. Classes in smali are always prefixed with an 'L' and suffixed with a ';' character. The namespace is also included, separated with '/' characters instead of the traditional '.' separator. An example method passing a String object and returning void (V) would be:

```
.method public someFunction(Ljava.lang.String;)V
```

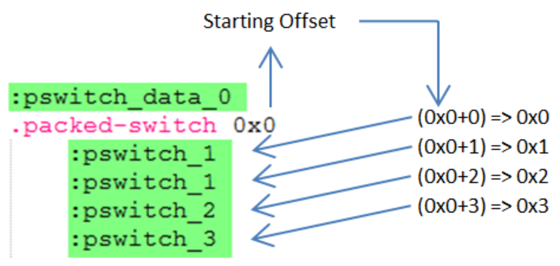
2.2.1 Packed Switches (pswitch)

Pswitch, short for Packed Switch, is a form of switch statement found in smali. Switch statements are a selection control mechanism found in many programming languages, used to allow the value of a variable or expression to change the control flow of a program execution via a multiway branch. [5]

Packed switches vary from those seen in C or Java because the logic is 'packed' into a section at the bottom of a function identified as 'pswitch_data' and corresponds to code labels. The values are relative to their index in the packed switch and must be incremental. This is perhaps best explained with an example. Consider the following smali statement:

```
packed-switch v3, :pswitch_data_0
```

This statement expresses that the block ':pswitch_data_0' will decide, depending on the current value of *v3*, where in the code to go next. The pswitch_data block in this case is shown below.


```
:pswitch_data_0
.packed-switch 0x0
    :pswitch_1
    :pswitch_1
    :pswitch_2
    :pswitch_3
```

The starting offset specified at the start of the pswitch_data block must be added to the 0-based row index in order to determine each value comparison. Because in this case the starting offset is 0x0, the first row represents a value for v3 of 0x0. Notice that rows 0 and 1 both have the ":pswitch_1" label. This means that, if v3 is either 0 or 1, the code found under that label should be executed. A hypothetical, C-style switch statement equivalent to the packed switch shown above would appear as follows:

```
switch (v3)
{
    case 0:
    case 1:
        //:pswitch_1
        break;
    case 2:
        //:pswitch_2
        break;
    case 3:
        //:pswitch_3
        break;
}
```

2.3 Android API Documentation

When reverse engineering Windows applications, MSDN, specifically Microsoft's online Windows API documentation is crucial to being able to understand API calls, including invocation parameters, symbolic constants, and more. Similarly in Android, the online API reference documentation, hosted on android.com, provides invaluable assistance. The URL for the documentation is http://developer.android.com/reference/

We will rely on the Android API documentation heavily when understanding the inner workings of MLT. It's important to note that when working with the API documentation, you can select the Android API version number. The API version number for Android 4.4.2 which the research specimen is running, is 17.

2.4 Tools Used

- Notepad++ [55], using smali syntax highlighting [60]. Notepad++ offers a powerful 'Find in Files' mechanism which is highly useful in the reversing process.

- apktool [1] is a tool for extracting the contents of apk files and performing baksmali on the dalvik bytecode (machine code) contained within.

- Bytecode Viewer [65] is a bytecode to Java disassembler which is useful in situations where the smali is difficult to understand due to more complicated control flow manipulation (loops, complex conditionals, complex method invocations). It occasionally fails however so an understanding of smali is still critical.

3 MLT.APK

To reiterate, this research was conducted using an MLT.apk sample obtained from an evidence device - an LG D802 cell phone running Android 4.4.2. Unlike apps downloaded by the user, MLT.apk itself resides on the system partition under /priv-app/.

3.1 APKTOOL Results

Using *apktool* on our MLT.apk results in 127 files being generated, organized into folders according to their pupose. A file at the root called apktool.yml is generated which contains JSON data representing application metadata. The version number is found here under the versionInfo.versionName property - in the case of our research sample this value is 2.5.4. 115 of our 127 files are .smali files containing disassembled dalvik bytecode. Each .smali file represents a corresponding class in the original Java source code. The .smali filenames (and thus classes they represent) do have descriptive filenames, for example MptMainLogProvider.smali. Unfortunately when it comes to subclasses (classes within a class) and subclasses of subclasses, the useful descriptions are usually (but not always) replaced with a numeric notation prefixed with a '$' character, for example:

MptOnWifiWatcher.smali

MptOnWifiWatcher$1.smali

MptOnWifiWatcher$2.smali

MptOnWifiWatcher$WiFiEventTrigger.smali

The numeric subclasses are best understood by looking at the content of those files and observing which classes they derive from - often being derived from superclasses which are well documented in the Android or Java reference documentation.

3.2 Application Instantiation

According to the Android API documentation, application instantiation is based upon the definitions provided in the manifest file (AndroidManifest.xml). Based on the manifest for MLT, we can see that *MltApplication* and *MltMonitorService* classes will be instantiated by the runtime upon initialization. There are also references to 'Providers', including MptMainLogProvider, MptCommonLogProvider, MptIncallLogProvider and MptBasicLogProvider. These providers correspond to the database filenames found on the *mpt* partition. The LogProvider classes all derive from *android.content.ContentProvider*, which the documentation describes as '*one of the primary building blocks of Android applications, providing content to applications. They encapsulate data and provide it to applications through the single ContentResolver interface.*' [105] See section 3.3 for more information.

Given that this application is designed to function almost completely in the background, it certainly fits the reference documentation's description for when to use a Service. The next section will describe how MltMonitorService is structured and operates.

3.2.1 MltMonitorService

MltMonitorService derives from android.app.Service, described in the documentation as *'an application component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use.*' [110]

MLTMonitorService is complex, having 5 direct descendant classes (and 5 more nested descendants) including:

- MLTMonitorService$1 (Object)

- MLTMonitorService$2 (Object)
- MLTMonitorService$3 (Object)
    - MLTMonitorService$3$1 (java.util.TimerTask)
- MLTMonitorService$BlobManager (java.lang.Object)
    - MLTMonitorService$BlobManager$1 (java.util.TimerTask)
    - MLTMonitorService$BlobManager$2 (java.util.TimerTask)
    - MLTMonitorService$BlobManager$3 (java.util.TimerTask)
- MLTMonitorService$DbFlushManager (java.lang.Thread)
    - MLTMonitorService$DbFlushManager$1 (android.os.Handler)

The documentation further states that upon creation (*<init>*), android.app.Service will call its onCreate method and then call its onStartCommand method with arguments supplied by the client. The service will continue to exist until it is stopped, at which point its onDestroy method will be called. onCreate for MltMonitorService instantiates many 'watcher' properties, each of which have an 'm' prefix, for example mScreenWatcher, which is of type com.lge.mlt.MptOnScreenWatcher.

Each of the m*Watcher properties are initialized in the *handleStart* method of *MltMonitorService*, usually by way of invoking the m*Watcher's *startListen* method. The *startListen* method demands a specific interface be passed as a parameter, and thanks to polymorphism that requirement is always fulfilled by the MltMonitorService instance itself (see 3.4).

MltMonitorService contains dozens of methods which are directly responsible for preparing data to be inserted into the SQLite database. These methods are prefixed with 'update' and 'insert' and 'on'. Examples include updateAppAccumulateLog, insertProcessInfoLog, and onGpsInfoSaved. Within these functions, the instruction 'const-string' is used before each piece of information is put onto the stack

3.3 LogProvider Classes

As described in 3.2, the LogProvider classes for this version of MPT include:

- MptBasicLogProvider
- MptCommonLogProvider
- MptIncallLogProvider
- MptMainLogProvider

The *LogProvider classes serve an important purpose, including:

- Specify the filename and path for the SQLite database on disk (e.g. /mpt/MPT_MainData.db)
- The ability to create a new SQLite database if it does not exist, with all associated tables, columns, and triggers. (Triggers are used exclusively to cap the maximum number of rows for each table)
- Provide static, immutable references in the form of URIs that will be referenced by other files when specifying the destination of new data.
- Provide query, insert, update and delete methods which are responsible for the construction of the actual SQL that is then executed against the target database.

From a reverse engineering perspective, the LogProviders therefore provide great information, including plain language names for each of the tables found within the SQLite databases.

I created a C# script to parse the URIs and associate them with database table names as a reference. The results of that script, and the script itself, will be included in the appendices of this research paper.

It appears that, while the crux of data falls in the 'MainLog', there are circumstances where duplicate data ends up in the tertiary logs - Basic, Common, and Incall. I do not at this point know the reason for this. One theory I have is that it may be that the tables are number capped using SQLite triggers, so by having alternate files, greater volumes of certain types of logs may be kept. As far as I can tell, there are no datatypes that are exclusive to any of the tertiary logs. That is, whatever information can be viewed in the tertiary logs is usually duplicitious.

3.4 Trigger Interfaces

In Java, interfaces are similar to classes with the exception that they only guarantee that a class will possess certain properties or method signatures. The interface does not contain the code for these methods.

A class uses the *.implements* statement to incorporate an interface. Then, to successfully compile, that class must have code matching the property/method signatures defined in the interface. The purpose of this centers around polymorphism - being able to pass around objects of different class types as the interface.

The MltMonitorService class implements a total of 24 interfaces. I am calling them 'trigger interfaces' because they are all subclasses of Receiver/Watcher classes, as an example: *MptOnWifiWatcher$WiFiEventTrigger*. (Recall that this notation means that *WiFiEventTrigger* is a subclass of *MptOnWifiWatcher*.)

Although there are a large number of Watcher -> Trigger -> Changed event chains, they tend to follow the same basic process. In the next section we will map the process for a specific example. If the process for one is understood then understanding all of them is not a large leap.

3.5 WiFi Watcher Example

So by implementing this *WiFiEventTrigger* interface, what methods are guaranteed to be implemented by MltMonitorService? Only one: *onWifFiInfoChanged(BILjava/lang/String;)V*
There are 3 arguments here: byte,int,string and the function returns void. We can see MltMonitorService's implementation of this at line 8131, where we get additional plain language clues as to the purpose of each argument - 'eventType', 'mState', and 'str'. The three arguments are used to populate an MptItem and subsequently invoke *MltMonitorService$DbFlushManager*'s *sendDbTransaction* method with a hard-coded dbTransactionType of 0x10. *sendDbTransaction* then crafts an *android.os.Message,* assigning the *dbTransactionType* to the *what* property of the message.

Next it invokes the *sendMessage* function of *MltMonitorService$DbFlushManager$1* (which derives from *android.os.Handler*). This enqueues the *android.os.Message* for processing on a different thread than the rest of the MltMonitorService. (asynchronous).

The *sendMessage* function is pre-defined by *android.os.Handler*, but *handleMessage* is not; it has to be defined by the class deriving from it, per the Android API documentation. This means that MltMonitorService$DbFlushManager$1 must implement *handleMessage(Message)* and it does at line 36. *handleMessage* is an extensive but straightforward function that performs a pswitch (see 2.2.1) on the value of the Message's *what* property, and invokes a specific method based on said the value. In our example, the dbTransactionType was 0x10. If we find the pswitch label for 0x10 (:pswitch_e in this case), we see that ultimately *MltMonitorService*'s *insertWiFiLog* function is called, which takes an MptItem (created in the earlier *onWiFiInfoChanged* method) as a parameter.

insertWiFiLog creates an instance of the *android.content.ContentValues* class which is essentially a keyed data dictionary. The important concept is that *ContentValues* is an expected argument for the android.database.sqlite.SQLiteDatabase class's insert method. [10] This dictionary is then populated with 'f002', 'f003', and so on being used as dictionary keys. These map to our SQLite column names. The function also retrieves the table URIs for two databases: *MptCommonLogProvider*'s *CONTENT_WIFI_PROPERTY_URI* and *MptMainLogProvider*'s *CONTENT_ACC_WIFI_PROPERTY_URI* . The constant values are *content://com.lge.mlt.common/t012* and *content://com.lge.mlt.main/t318* respectively.

Finally, *insertWiFiLog* calls the LogProvider's *insert* command, passing the table URI and *ContentValues* object as parameters. The LogProviders then invoke the associated SQLiteDatabase insert commands which is where the SQL is finally generated and executed on the SQLite file on disk.

I created a C# script to parse the possible values of *what* and create a chart of *what* values to method invocations, as well as the ultimately affected database table.

3.6 Diagram

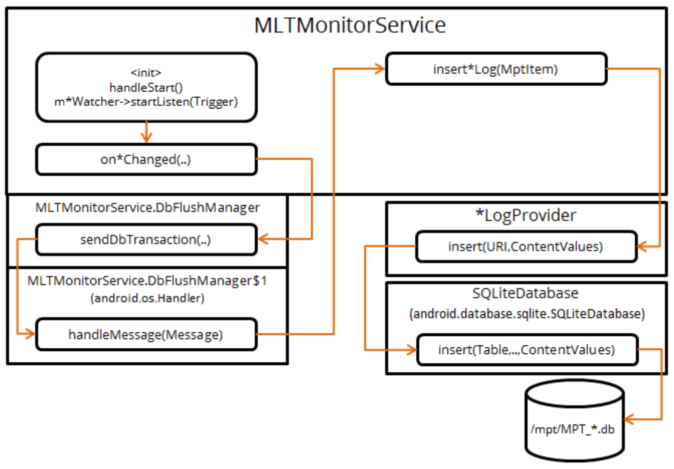

MLTMonitorService
<init>
handleStart()
m*Watcher->startListen(Trigger)
on*Changed(..)
insert*Log(MptItem)
MLTMonitorService.DbFlushManager
sendDbTransaction(..)
MLTMonitorService.DbFlushManager$1
(android.os.Handler)
handleMessage(Message)
*LogProvider
insert(URI,ContentValues)
SQLiteDatabase
(android.database.sqlite.SQLiteDatabase)
insert(Table,...,ContentValues)
/mpt/MPT_*.db

Figure 1: Process Flow Diagram

3.7 Database Schema Commonalities

It appears that for all tables, the first column, F001, is used to represent the ROWID (primary key). The second column, F002, is used to represent the log entry timestamp. F003 and beyond is where the variance begins.

4 MPT_MAINDATA.DB

/mpt/MPT_MainData.db has the largest collection of log types, so we will start there first. In this section we will identify the purpose of each numeric table (with help from the MPTMainLogProvider - see 3.3) in the database as well as each field within those tables. MPT_MainData.db has a total of 27 such numerically named tables ranging from 't301' to 't328'.

Of note, each table consistently uses F001 as a ROWID and F002 as getSampleTime(), a function that returns a numeric value associated to the event occurrence timestamp. After F002, the purpose of fields diverges greatly by table.

4.1 t301 - Basic Info

This table keeps track of information about the device including unique identifiers, OS version, system configuration and cell network information.

This data is associated with MptOnBasicWatcher (and its subclasses) and MptBasicItem. The corresponding methods in MltMonitorService are insertBasicInfoLog and onBasicInfoChanged. The URI for t301 is CONTENT_BASIC_INFO_URI. There are a total of 21 fields, including:

F001	ROWID
F002	MptBasicItem->getSampleTime()

F003	MptBasicItem->getDeviceid()
	android.telephony.TelephonyManager->getDeviceId()
	Returns the unique device ID, for example, the IMEI for GSM and the MEID or ESN for CDMA phones. Returns null if device ID is not available. Requires the READ_PHONE_STATE permission. [15]

F004	MptBasicItem->getDeviceid_Type()
	android.telephony.TelephonyManager->getPhoneType()
	PHONE_TYPE_NONE = 0
	PHONE_TYPE_GSM = 1
	PHONE_TYPE_CDMA = 2
	PHONE_TYPE_SIP = 3
	[15]
	However only valid values are 1 or 2.

F005	MptBasicItem->getSuffix()
	android.telephony.TelephonyManager->getSimOperator()
	Returns the MCC+MNC (mobile country code + mobile network code) of the provider of the SIM. 5 or 6 decimal digits.[15]

F006	MptBasicItem->Serial_no() android.os.Build->SERIAL A hardware serial number, if available. Alphanumeric only, case-insensitive. [20]

F007	MptBasicItem->getOs_ver()
	android.os.SystemProperties->get("ro.build.version.release")

F008	MptBasicItem->getHw_ver()
	android.os.SystemProperties->get("ro.revision")

F009	MptBasicItem->getSw_ver()
	android.os.SystemProperties->get("ro.lge.swversion")

F010	MptBasicItem->getModel_name() android.os.Build->MODEL The end-user-visible name for the end product. [20]
F011	MptBasicItem->getBuild_no() android.os.Build->FINGERPRINT A string that uniquely identifies this build. [20]

F012	MptBasicItem->getImsi()
	android.telephony.TelephonyManager->getSubscriberId()
	Returns the unique subscriber ID, for example, the IMSI for a GSM phone.[15]

F013	MptBasicItem->getRegistration()
F014	MptBasicItem->getPhoneType() See F004.

F015	MptBasicItem->getNetworkType()
	android.telephony.TelephonyManager.getNetworkType()
	The NETWORK_TYPE_xxxx for current data connection. [15]
	NETWORK_TYPE_GPRS = 1
	NETWORK_TYPE_EDGE = 2
	NETWORK_TYPE_UMTS = 3
	NETWORK_TYPE_CDMA = 4
	NETWORK_TYPE_EVDO_0 = 5
	NETWORK_TYPE_EVDO_A = 6
	NETWORK_TYPE_1xRTT = 7
	NETWORK_TYPE_HSDPA = 8
	NETWORK_TYPE_HSUPA = 9
	NETWORK_TYPE_HSPA = 10
	NETWORK_TYPE_IDEN = 11
	NETWORK_TYPE_EVDO_B = 12
	NETWORK_TYPE_LTE = 13
	NETWORK_TYPE_EHRPD = 14
	NETWORK_TYPE_HSPAP=15
	NETWORK_TYPE_GSM = 16
	NETWORK_TYPE_TD_SCDMA = 17
	NETWORK_TYPE_IWLAN = 18

F016	MptBasicItem->getScreen_density() android.util.DisplayMetrics->density The logical density of the display. [25]
F017	MptBasicItem->getScreen_width() android.util.DisplayMetrics->widthPixels The absolute width of the available display size in pixels. [25]
F018	MptBasicItem->getScreen_height() android.util.DisplayMetrics->heightPixels The absolute height of the available display size in pixels. [25]

F019	MptBasicItem->getScreen_orient()
	android.content.res.Configuration->orientation
	ORIENTATION_UNDEFINED = 0
	ORIENTATION_PORTRAIT = 1
	ORIENTATION_LANDSCAPE = 2
	ORIENTATION_SQUARE = 3

F020	MptBasicItem->getRam_size()
	com.lge.mlt.MptOnBasicWatcher->getTotalMem()
	This function parses the contents of /proc/meminfo

F021	MptBasicItem->getMpt_version()
	MptOnBasicWatcher->mContext->getString(0x7f05000a)
	Retrieves the value of the resource with offset 0x7f05000a, found in /res/values/public.xml. In this case, a string with name 'str_MLT_VERSION'. In /res/values/strings.xml, an entry with key 'str_MLT_VERSION' shows a value of 2.5.4.

4.2 t302

For reasons unknown, there is no t302. It is skipped over in the sequence of table numbers.

4.3 t303 - Rooting History

The purpose of this table remains unclear and further work would be required. It monitors MPT related events "MPT.ROOT_INSPECTION_EVENT" and "MPT.OS_ROOT_INSPECTION_EVENT".

This data utilizes the custom methods *doOsInspection* and *doInspection*, which in turn use the Android Package Manager API to iterate installed packages on the device. android.content.Context->getPackageManager()->getInstalledPackages(0); It uses MptItem class. The corresponding MltMonitorService method is insertRootingHistory.

The URI for t303 is CONTENT_ROOTING_HISTORY_URI. t303 is related to the MptOnRootInspector class (and its subclasses). It corresponds to a message->*what* value of 0x11, and trigger interface $RootInspectorTrigger which has two functions, *onRootingHistoryInspector* and *onPreloadInspector*. The latter *onPreloadInspector* seems to deal with preloaded apps and deals exclusively with MptBasicLogProvider.

The research specimen for this case has only 2 rows.

F001	ROWID
F002	getSampleTime()
F003	List of packages separated by \n (ASCII 0xa)

4.4 t304 - App History

This data represents installations, updates, and removals of apps on the device. It uses MptItem. Trigger function is onAppInfoChanged, write functions include insertAppAccumulateLog, updateAppAccumulateLog, and updateRemovedAppAccumulateLog. The URI for t304 is CONTENT_APP_HISTORY_UIR (apparent typo on URI).

F001	ROWID
F002	getSampleTime()
F003	pkg_name
F004	Original install time.

	Stores the original value of F002 so that when it is upgraded or deleted, F002 can be modified.
F005	App removal time. This value is null for any app that has not been removed.
F006	App version

4.5 t305 - App Accumulation

Keeps track of installed applications (packages), launch counts and usage.

Provided by the com.android.internal.app.IUsageStats service by invoking the *getAllPkgUsageStats* method, which returns com.android.internal.os.PkgUsageStats, an object containing this information. This is a difficult piece of code to interpret, but I believe the way this works is that a hashmap is generated using existing data to decide whether to insert or update the data in t305.

This data is associated with MptOnAppUsageWatcher (and its subclasses) and MptItem. MptOnAppUsageWatcher's trigger interface defines two trigger functions: onAppUsageInfoInsert and onAppUsageInfoUpdate... The URI for t305 is CONTENT_APP_ACCUMULATION_URI.

F001	ROWID
F002	getSampleTime()
F003	Package Name
F004	Application Usage Time
F005	Application Launch Count

4.6 t306 - Call Accumulation

More work to be done in understanding the purpose of this table. In light of the available time, and the lack of rows in the research specimen, that work is pushed toward the future.

Involved methods: onUpdateIncommingCall, onUpdateOutgoingCall, onTelephonyInfoChanged, onDataConnectionStateChanged.

Also involves MptOnTelephonyWatcher, MptOnLogDumper.

Associated with message->*what* property of 0x3,

The URI for t306 is CONTENT_CALL_ACCUMULATION_URI. In the research specimen this table held only 1 row, which had an F002 of 0x14 (20).

F001	ROWID
F002	getSampleTime()
F003	eventType 0x20 = onDataConnectionStateChanged?
F004	Integer - unknown
F005	Long - unknown

4.7 t307 - Acc App Usage

This table uses the logcat process with the parameters: "-v time CallNotifier:D AndroidRuntime:E ActivityManager:V DEBUG:I Process:I *:S"

This data is associated with MptOnLogReceiver (and its subclasses) and uses MptItem. The trigger interface is called MptOnLogReceiver$ProcessInfoTrigger which requires one method be implemented called onProcessInfoChanged.

onProcessInfoChanged takes 4 parameters (string, byte, int, int) and handily defines plaintext descriptions of them in the .locals list, which is how I populated the list of fields.

The insert function, insertProcessInfoLog, is invoked by DbFlushManager when the message->*what* property equals 0xF...The URI for t307 is CONTENT_ACC_APP_USAGE_URI. This database table had 15000 rows in our dataset.

F001	ROWID
F002	getSampleTime()
F003	Package name
F004	eventType 0xb = "activity" 0xd = "broadcast" oxe = "service" 0xf = content 0x10 = "added" 0x13 = parse first 4 chars as int, 0x14 = "has died." || SIG: 9 Reaffirmed via definitions in MptOnLogReceiver: MPT_APP_USAGE_END = 0x14; MPT_APP_USAGE_OTHERS = 0x13; MPT_APP_USAGE_START = 0x5a; MPT_APP_USAGE_START_ACTIVITY = 0xb;

	MPT_APP_USAGE_START_ADDED_APP = 0x10;
	MPT_APP_USAGE_START_BROADCAST = 0xd;
	MPT_APP_USAGE_START_PROVIDER = 0xf;
	MPT_APP_USAGE_START_SERVICE = 0xe;

F005	pid (Process ID)
F006	uid (User ID)

4.8 t308 - Acc Battery Info

This data is associated with MptOnBatteryWatcher, which defines two trigger interfaces - *BatteryStateTrigger* and *BatteryInfoTrigger*, and uses MptItem. *BatteryStateTrigger* defines the *onBatteryStateChanged* method while BatteryInfoTrigger defines *onBatteryInfoChanged*. These trigger functions use a dbTransactionType of 0x0 and 0x1, respectively. DbFlushManager utilizes *insertBatteryLog* for both of these triggers. The URI for t308 is CONTENT_ACC_BATTERY_INFO_URI.

MptOnBatteryWatcher's two subclasses (*$1 and *$2) are Broadcast Receivers which will be explained in detail in subsections 4.8.1 and 4.8.2.

Our dataset had 24000 rows in this table making it one of the largest.

F001	ROWID
F002	getSampleTime()
F003	eventType 0xb (11) = BATTERY_CHANGED 0xc (12) = BATTERY_LOW 0xd (13) = BATTERY_OKAY
F004	Packed battery information including: battery level, voltage, temperature, health, status, plug type, and presence.

4.8.1 onBatteryInfoChanged

MptOnBatteryWatcher's first receiver, $1, is associated with onBatteryInfoChanged, and is fairly complex. It listens for (android.intent.action) BATTERY_CHANGED events. The data contained within this F004 includes: level, voltage, temperature, health, status, plugType, and present. It uses a function called getPackedBatteryInfo which employs bitmasks to place all of these values of data into a single numeric value. Noteably, Receiver#1 also appears to hardcode "f003=11" by placing it into the ContentResolver. The key to being able to decode F004 is understanding the getPackedBatteryInfo method.

getPackedBatteryInfo takes 6 integers as arguments and arranges them using a combination of masking and bitshifting. These operations are highly confusing in smali, but Bytecode Viewer does a good job of spelling it out in one line:

```
   return (0x7 & i) << 60 | (0xF & paramInt6) << 56
| (0xF & paramInt5) << 52 | (0xF & paramInt4) <<
48 | (0xFFFF & paramInt3) << 32 | (0xFFFF &
paramInt2) << 16 | 0xFFFF & paramInt1;
```

The nearby function *getUnpackedBatteryLevel* gives us a clue to how to decode a value:

```
   return (int)(0xFFFF & paramLong);
```

In our encoding example, *paramInt1* represents the battery level. In our decoding example, the reverse is applied. One important note is that because bitshifting occurs last during encoding, it needs to occur first during decoding, which we can achieve using parenthesis. Therefore, the correct way to decode *paramInt3* for example would be:

```
   return (int)(0xFFFF & (paramLong >> 32));
```

Notice that the bitwise operator has also swapped directions.

There is another alternative contained within MptOnBatteryWatcher though and those are the PACKED_BATTERY_*_MASK fields, which are themselves bitmasks which can be applied to database values without any bitshifting. They are as follows:

PACKED_BATTERY_LEVEL_MASK = 0xffff
PACKED_BATTERY_VOLTAGE_MASK = 0xffff0000
PACKED_BATTERY_TEMPERATURE_MASK = 0xffff00000000
PACKED_BATTERY_HEALTH_MASK = 0xf 000000000000
PACKED_BATTERY_SATUS_MASK = 0xf0000000000000
PACKED_BATTERY_PLUG_TYPE_MASK = 0xf00000000000000
PACKED_BATTERY_PRESENT_MASK = 0x7000000000000000

Therefore, for an example database F004 value of 1166996576538132534, we can bitmask said value with any of the above masks to find that value.

4.8.2 onBatteryStateChanged

Receiver #2, associated with onBatteryStateChanged, is straightforward. It listens for android.intent.action.BATTERY_LOW and BATTERY_OKAY events and logs them using 0xc (12) and 0xd (13) respectively.

4.9 t309 - Acc Signal Strength

Related to MptOnTelephonyWatcher and its subclasses. Uses *MptItem*. Implements MptOnTelephonyWatcher$ SignalStrengthTrigger which defines the function onSignalStrengthChanged(byte,bool,int,int,int,int,int). This trigger function is invoked exclusively by the *MptOnGetSignalStrengths(android.telephony.SignalStrength)* method found in MptOnTelephonyWatcher.

MptOnGetSignalStrengths has separate logic in place to deal with GSM, GSM/LTE, CDMA or EVDO. In the case of non-LTE GSM, the *eventType* is set to 23. eventType maps to F003. For LTE, the value is 24. In our working dataset, T309 had 6000 rows, 4390 of which had an f003 of 23, and 1610 of which had an f003 of 24.

The API documentation shows us the following function definitions[30]:

```
getGsmSignalStrength()

     Get the GSM Signal Strength, valid values
are (0-31, 99) as defined in TS 27.007 8.5.

getGsmBitErrorRate()

     Get the GSM bit error rate (0-7, 99) as
defined in TS 27.007 8.5.
```

TS 27.007 8.5 is a reference to the 3GPP (3rd Generation Partnership Project) Technical Specification which is a publically available document titled 'AT command set for User Equipment (UE)'. [35] We can use the values in this reference to map the possible integer values to dBm ranges.

Something to be aware of is that dBm ranges are always negative because signal gets weaker as it moves away from its source. This could explain the manipulation seen in F004 where a constant value of 0x72 is subtracted from twice the value returned by getLteSignalStrength().

This table is written to by insertSignalStrengthLog for messages with a ->*what* property of 0x5.

..The URI for t309 is CONTENT_ACC_SIGNAL_STRENGTH_URI.

F001	ROWID
F002	getSampleTime()
F003	eventType 21 = CDMA or EVDO 22 = CDMA or EVDO 23 = Non-LTE GSM 24 = LTE/GSM
F004	Signal Strength (For F003=23) The result of *getGsmSignalStrength()* (For F003=24) The result of (*getLteSignalStrength()* * 2 - 0x72).
F005	Error Rate / RSRP (For F003=23) The result of *getGsmBitErrorRate()* (For F003=24) *The result of getLteRsrp()* aka Reference Signal Received Power.
F006	LTE RSRQ (For F003=23) Always set to -1. (For F003=24) The result of *getLteRsrq()*, aka Reference Signal Received Quality.
F007	LTE RSSNR (For F003=23) Always set to -1. (For F003=24) The result of *getLteRssnr()* aka Reference Signal to Noise Ratio
F008	LTE CQI (For F003=23) Always set to -1. (For F003=24) The result of *getLteCqi()* aka Channel Quality Indicator.

4.10 t310 - Acc Telephony Info

The URI for t310 is CONTENT_ACC_TELEPHONY_INFO_URI. The sole reference to this URI is in the function *insertTelephonyInfoLog* (MltMonitorService), which takes an *MptItem* as a parameter.

insertTelephonyInfoLog writes simultaneously to MptCommonLog (t004) and MptMainLog (t310).

The insert function is invoked in correspondence with a message->what property value of 0x3. The function that causes this is a trigger interface, MptOnTelephonyWatcher$TelephonyInfoTrigger. The method name is onTelephonyInfoChanged and it takes two bytes as parameters. These bytes become f003 and f004.

In the research specimen, this table contained 6000 rows.

F001	ROWID
F002	getSampleTime()
F003	getEventType() In research specimen the range of distinct values included 31, 32, and 33. (0x1F, 0x20, 0x21) I believe these are the constants defined in the constructor of MptOnTelephonyWatcher including: 0x1f - MPT_EVENT_TELEPHONY_SERVICE_STATE 0x20 - MPT_EVENT_TELEPHONY_DATA_CONN_STATE 0x21 - MPT_EVENT_TELEPHONY_CALL_STATE 0x22 - MPT_EVENT_TELEPHONY_DATA_ACTIVITY_STATE Note that 0x22 doesn't appear in this table, I believe it instead ends up in t327, Acc Data Activity (see 4.27)
F004	getIntArg0() In research specimen the range of distinct values included -1, 0, 1, 2, and 3. We can look to the definitions in the constructor of MptOnTelephonyWatcher to see the possible symbolic representations. If F003 = 0x1f: SERVICE_STATE_UNKNOWN: -0x1 SERVICE_STATE_IN_SERVICE: 0x0 SERVICE_STATE_OUT_OF_SERVICE: 0x1 SERVICE_STATE_EMERGENCY_ONLY: 0x2 SERVICE_STATE_POWER_OFF: 0x3 If F003 = 0x20: DATA_CONN_STATE_UNKNOWN: -0x1 DATA_CONN_STATE_DISCONNECTED: 0x0 DATA_CONN_STATE_CONNECTING: 0x1 DATA_CONN_STATE_CONNECTED: 0x2 DATA_CONN_STATE_SUSPENDED: 0x3 If F003 = 0x21: CALL_STATE_UNKNOWN: -0x1 CALL_STATE_IDLE: 0x0 CALL_STATE_INCOMING_CALL: 0x1 CALL_STATE_OFFHOOK: 0x2 If F003 = 0x22: DATA_ACTIVITY_NONE: 0x0 DATA_ACTIVITY_IN: 0x1 DATA_ACTIVITY_OUT: 0x2 DATA_ACTIVITY_INOUT: 0x3 DATA_ACTIVITY_DORMANT: 0x4

There were only 9 distinct pairs of F003/F004 value pairs:

F003	F004
31	0
31	1
31	3
32	-1
32	0
32	2
33	0
33	1
33	2

4.11 t311 - Acc CDMA Cell Info

This table contains CDMA related event logs. It is related to *MptOnTelephonyWatcher* and implements the *CDMACellInfoTrigger*. This trigger interface requires one method be implemented, *onCDMACellInfoTrigger*. It uses *MptItem* as a container to send data to the database. It is called exclusively by the function *onCellLocationChanged* (*MptOnTelephonyWatcher$1*) which parses information from the API class *android.telephony.cdma.CdmaCellLocation* which derives from *android.telephony.CellLocation*.

The trigger function uses a message->*what* value of 0x6 which maps to *insertCDMACellInfoLog*.

A note on latitude and longitude measurements from the Android API:

Latitude is a decimal number as specified in 3GPP2 C.S0005-A v6.0. (http://www.3gpp2.org/public_html/specs/C.S0005-A_v6.0.pdf) It is represented in units of 0.25 seconds and *ranges from -1296000 to 1296000, both values inclusive (corresponding to a range of -90 to +90 degrees). Integer.MAX_VALUE is considered invalid value.* [40]

Longitude is a decimal number as specified in 3GPP2 C.S0005-A v6.0. (http://www.3gpp2.org/public_html/specs/C.S0005-A_v6.0.pdf) It is represented in units of 0.25 seconds and ranges from -2592000 to 2592000, both values inclusive (corresponding to a range of -180 to +180 degrees). Integer.MAX_VALUE is considered invalid value. [40]

The URI for t311 is CONTENT_ACC_CDMA_CELL_INFO_URI.

This database table was empty in our dataset, which is not surprising as it is a GSM phone.

F001	ROWID
F002	getSampleTime()
F003	getBaseStationId() *cdma base station identification number, -1 if unknown* [40]
F004	getBaseStationLatitude() *cdma base station latitude in units of 0.25 seconds, Integer.MAX_VALUE if unknown* [40]
F005	getBaseStationLongitude() *cdma base station longitude in units of 0.25 seconds, Integer.MAX_VALUE if unknown* [40]
F006	getNetworkId() *cdma network identification number, -1 if unknown* [40]
F007	getSystemId() *cdma system identification number, -1 if unknown* [40]

4.12 t312 - Acc GSM Cell Info

This table contains GSM related event logs. It is related to MptOnTelphonyWatcher and implements the GSMCellInfoTrigger. This trigger interface requires one method be implemented, *onGSMCellInfoTrigger*. It uses *MptItem* as a container to send data to the database. It is called exclusively by the function *onCellLocationChanged* (*MptOnTelephonyWatcher$1*) which parses information from the API class *android.telephony.gsm.GsmCellLocation* which derives from android.telephony.CellLocation.

GSM Cell ID (CID) is a generally unique number used to identify each base transceiver station (BTS) or sector of a BTS within a location area code (LAC) if not within a GSM network. [50]

There are publically available databases which index cell tower information based on four parameters: MCC (Mobile Country Code), MNC (Mobile Network Code), LAC (Location Area Code), and CellID (CID). These four parameters combine form the LAI (Location Area Identity). When all four parameters are known, a GPS coordinate for the base station (cell tower) can be ascertained by performing a lookup on one of the many databases available. The wikipedia page for Cell ID [50] provides a list of these databases, some of which are free and some of which are commercial and require money to access.

Because t312 contains only the LAC and CID parameters, one must also reference t301, specifically F005 which provides the MCC MNC component. In later versions of MPT, it appears that the MCC and MNC are also included as separate columns in this table.

The trigger function uses a message->*what* value of 0x2 which maps to *insertGSMCellInfoLog*. The URI for t312 is CONTENT_ACC_GSM_CELL_INFO_URI.

In the research specimen, this table contained 4250 rows.

F001	ROWID
F002	getSampleTime()
F003	getCid() *gsm cell id, -1 if unknown, 0xffff max legal value* [45]
F004	getLac() *gsm location area code, -1 if unknown, 0xffff max legal value* [45]
F005	getPsc() *On a UMTS (Universal Mobile Telecommunications Service, 3G) network, returns the primary scrambling code of the serving cell. -1 if unknown or GSM*

4.13 t313 - Acc Connectivity Info

This table is written to by *insertUmsInfoLog*, which uses a *ContentValues* object and *insertConnectivityInfoLog* which uses *MptItem*.

insertConnectivityInfoLog corresponds to a message->*what* of 0x12. The only time this appears to be set is in the function *onGpsInfoChanged* (*MltMonitorService*), which is an implementation of the trigger interface MptOnGpsWatcher$GpsEventTrigger. These triggers occur in MptOnGpsWatcher$1 (an android.location.GpsStatus.Listener [70]) and MptOnGpsWatcher$2 (a BroadcastReceiver [75]).

The URI for t313 is CONTENT_ACC_CONNECTIVITY_INFO_URI. In the research specimen, this table had 3000 rows.

4.13.1 MptOnGpsWatcher$1

GpsStatus.Listener [70] is an interface that is 'used for receiving notifications when GPS has changed'. It requires one method be implemented, *onGpsStatusChanged*, which is described as being called to report changes in the GPS status. A list of event type constants, and their causation is provided:

GPS_EVENT_STARTED (0x1) - *Event sent when the GPS system has started.*

GPS_EVENT_STOPPED (0x2) - *Event sent when the GPS system has stopped.*

GPS_EVENT_FIRST_FIX (0x3) - *Event sent when the GPS system has received its first fix since starting.*

GPS_EVENT_SATELLITE_STATUS (0x4) - *Event sent periodically to report GPS satellite status.*

The logic for *onGpsStatusChanged* is convoluted, even when viewing with Bytecode Viewer. There are invocations to both *onGpsInfoChanged* and *onGpsInfoSaved* however, which are both members of the *MptOnGpsWatcher$GpsEventTrigger* interface. The former seems to be related to GPS status changes while the latter seems to be related to actual GPS information (satellite information, elevation, azimuth etc). My interpretation of the first part of the code is that in the case of event type 0x4, an option called GPS_FIRST_FIX_CONTINUE (MptOnGpsWatcher) is evaluated to decide whether to continue with insertion or not. In all other cases, the value 0x2a is set which I believe maps to column F003 of this table. The second half of the function deals with obtaining satellite info from the system service "location" and likely correlates to t319 (see section 4.19).

Rows inserted on behalf of MptOnGpsWatcher$1 always have an F003 of 0x2a (42). F004 is simply parameter 1 of the function invocation, which is described as mState but represents the constant value from the list above.

4.13.2 MptOnGpsWatcher$2

This is a Broadcast Receiver [75] which listens for android.location.PROVIDERS_CHANGED. This intent is defined in the API reference documentation for android.location.LocationManager [80] as PROVIDERS_CHANGED_ACTION, and is described as "*Broadcast intent action when the configured location providers change.*" This is vague, but the explanation for LocationManager itself is perhaps more helpful: "*This class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location, or to fire an application-specified Intent when the device enters the proximity of a given geographical location.*" [80]

As with $1, the logic here is complex. Events here are sent to onGpsInfoChanged with the constant pairs 0x29 (41) and 0x1, or 0x2a (42) and 0x5. These values are destined for F003 and F004, respectively.

F001	ROWID
F002	getSampleTime()
F003	
F004	In sample data, always between 1-5.

4.14 t314 - Acc Power Info

This table stores information about reboot and shutdown device events. It is related to MptOnPowerWatcher and its subclasses.

Written to by insertPowerInfoLog (MltMonitorService). Implements MptOnPowerWatcher$PowerEventTrigger interface, which defines the onPowerInfoChanged method. This trigger function is invoked by MptOnPowerWatcher$1, a BroadcastReceiver [75]. $1 listens for android.intent.action.ACTION_SHUTDOWN and android.intent.action.REBOOT.

ACTION_SHUTDOWN is defined in the developer reference under android.content.Intent:

Broadcast Action: Device is shutting down. This is broadcast when device is being shutdown (completely turned off, not sleeping). Once the broadcast is complete, the final shutdown will proceed and all unsaved data will be lost. Apps will not normally need to handle this, since the foreground activity will be paused as well. [85]

ACTION_SHUTDOWN correlates to an eventType of 0x34 (52) which is stored in F003.

ACTION_REBOOT is also defined in the developer reference under android.content.Intent:

Broadcast Action: Have the device reboot. This is only for use by system code. [85]

ACTION_REBOOT correlates to an eventType of 0x36, which is stored in F003.

The values 0x33 and 0x35 are sent to this table when MLT initializes itself (specifically when MltMonitorService onCreate gets called). The difference being related to the existence of the file /mpt/started . If it exists, 0x35 is used, if not, 0x33 is used.

The URI for t314 is *CONTENT_ACC_POWER_INFO_URI* and the research specimen contains 300 rows.

F001	ROWID
F002	getSampleTime()
F003	eventType 0x34 (52) = ACTION_SHUTDOWN [85] 0x36 (54) = ACTION_REBOOT [85]
F004	
F005	

4.15 t315 - Acc Screen Info

This table is related to MptOnScreenWatcher and its subclasses. The trigger interface, MptOnScreenWatcher$ScreenEventTrigger has one method, o*nScreenInfoChanged.* It uses MptItem and an message->*what* property value of 0x8.

onScreenInfoChanged is fired by MptOnScreenWatcher$1, a *BroadcastReceiver* [75] which listens for

The *insertScreenInfoLog* function is responsible for writing to this table.

..The URI for t315 is CONTENT_ACC_SCREEN_INFO.

F001	ROWID
F002	getSampleTime()
F003	eventType 0x3d = MPT_EVENT_SCREEN_ON 0x3e = MPT_EVENT_SCREEN_OFF
F004	
F005	

4.16 t316 - Acc Resource Info

The URI for t316 is CONTENT_ACC_RESOURCE_INFO_URI. This URI is invoked in the function *insertResourceInfo*, which uses *MptItem* to populate a *ContentValues* dictionary and then performs inserts into MptCommonLog (t010) and MptMainLog (t316). The insert function is called in response to a message->*what* value of 0x1B (27).

This table is associated with MptOnResourceInfoWatcher (and its subclasses) and implements the trigger interface MptOnResourceInfoWatcher$ResourceInfoChecker, which has one method, *onResourceChecked* taking two integers as parameters. The parameters are provided with names in the method definition as "cpuUsage" and "availRam". MptOnResourceInfoWatcher$1 is a TimerTask (java.util.TimerTask) which represents a task that can be scheduled for one-time or repeated execution by a Timer. [95] The *run* command invokes the custom functions *getCpuUsage* and *getAvailRam* which are members of the parent class.

getCpuUsage reads from the file /proc/stat, tokenizing/splitting the result and then parsing it for certain values. Once obtained, mathematic operations appear to be performed, possibly to ascertain a CPU use percentage. Unfortunately the logic is quite complex and further work is required here to ascertain how exactly this function operates. Bytecode Viewer also fails to decode this function.

getAvailRam on the other hand, uses the Android API class android.app.ActivityManager$MemoryInfo, specifically the *availMem* property, a long integer, which is described on the API documentation as "The available memory on the system". [100] Note that it appears the value of availMem is divided by 0x400 (1024) suggesting a probable conversion of bytes to kilobytes.

In our research specimen this table contained 12000 rows. It appears that this timer is setup to run every 60 seconds (java.util.Timer->schedule with a v4 of 0xea60 aka 60000ms). I further corroborated this by ordering some of the sample data chronologically and then calculating the difference between F002s. I found that the value was very close to 60000, sometimes +/- 5 to 15 ms, which to me would be a reasonable variance considering the asynchronous nature of this application.

F001	ROWID
F002	getSampleTime()
F003	getCpuUsage() Custom function that appears to ascertain a percentage of CPU usage at that moment using /proc/stat In the research specimen, this value always falls between 0 and 100.
F004	getAvailRam() android.app.ActivityManager.MemoryInfo's availMem property divided by 1024. In the research specimen, this column's value ranged between 186696 and 1401212. I suspect this would represent a minimum of 186MB and a maximum of 1.4 GB.

4.17 t317 - Acc Bluetooth Info

The URI for t317 is CONTENT_ACC_BLUETOOTH_INFO_URI. This URI is invoked in the function *insertBluetoothInfoLog* which uses a *ContentValues* array. It corresponds to a message->*what* property of 0x1A (26) which is sent by the *onBluetoothInfoChanged* method, which it implements due to the MptOnBluetoothWatcher$BluetoothEventTrigger trigger interface.

This table corresponds with MptOnBluetoothWatcher (and its subclasses). It assigns values to event types as follows:

MPT_EVENT_BT_ADAPTER_CHANGED - 0xb
MPT_EVENT_BT_HEADSET_CHANGED - 0xc
MPT_EVENT_BT_A2DP_CHANGED - 0xd
MPT_EVENT_BT_PBAP_CHANGED - 0xe
MPT_EVENT_BT_DEVICE_CHANGED - 0xf
MPT_EVENT_BT_DEVICE_BOND_CHANGED - 0x10
BT_STATE_UNKNOWN - (-0x1)
BT_STATE_OFF - 0xa
BT_STATE_TURNING_ON - 0xb
BT_STATE_ON - 0xc
BT_STATE_TURNING_OFF - 0xd
BT_BOND_STATE_NONE - 0xa
BT_BOND_STATE_BONDING - 0xb
BT_BOND_STATE_BONDED - 0xc
BT_STATE_ERROR - (-0x1)
BT_STATE_DISCONNECTED - 0x0
BT_STATE_CONNECTING - 0x1
BT_STATE_CONNECTED - 0x2
BT_STATE_DISCONNECTING - 0x3
BT_STATE_PLAYING - 0x4
BT_STATE_ACL_CONNECTED - 0x5
BT_STATE_ACL_DISCONNECTED - 0x6
BT_STATE_ACL_DISCONNET_REQUESTED - 0x7

The $1 subclass is a BroadcastReceiver which registers the following intent filters (each of which having the android.bluetooth prefix abbreviated as a.b.):
a.b.adapter.action.STATE_CHANGED
a.b.device.action.BOND_STATE_CHANGED
a.b.device.action.ACL_DISCONNECTED
a.b.a2dp.profile.action.CONNECTION_STATE_CHANGED
a.b.headset.profile.action.CONNECTION_STATE_CHANGED
a.b.pbap.intent.PBAP_STATE.

The logic in the BroadcastReceiver is convoluted. It appears to assess the event type and assign a value using several packed-switch statements and conditional logic. However, ultimately the trigger function is called with 3 parameters (byte,byte,string).

Research data specimen contained 2454 rows.

F001	ROWID
F002	getSampleTime()
F003	eventType Values were always 11, 12, 13, 15, 16 in research data.
F004	Values were always 0,1,2,5,6,10,11,12,13 in research data. Parameters as follows: a.d.b.a.STATE_CHANGED: 0xb, -0x1 a.d.b.a.BOND_STATE_CHANGED: 0x10, 0xa - Note: this is the only outcome where a possible value for F005 is obtained. a.b.d.a.ACL_CONNECTED => 0xf, 0x5 a.b.d.a.ACL_DISCONNECTED => 0xf, 0x6 All others => 0xf, 0x7
F005	MAC address of Bluetooth hardware for a.b.d.action.BOND_STATE_CHANGED events only -- a concatenation of *localBluetoothDevice.getAddress()*, a newline character ('\n') and

	android.bluetooth.device.extra.REASON (int). In the research data this REASON code appears to always be 0.

4.18 t318 - Acc Wifi Property

The URI for t318 is CONTENT_ACC_WIFI_PROPERTY_URI. It is written to by the insertWifiLog method using an MptItem, and writes simultaneously to MptCommonLog (t012) and MptMainLog (t318). It corresponds to a message->*what* property value of 0x10 (16), and the trigger interface MptOnWifiWatcher$WiFiEventTrigger.

The variable names for *onWifiChanged* include eventType (byte), mState (integer), and str (string).

MptOnWifiWatcher assigns the following constants in its constructor:
WIFI_EVENT_UNKNOWN => -0x1
WIFI_STATE_UNKNOWN => -0x1
WIFI_STATE_DISABLING => 0x0
WIFI_STATE_DISABLED => 0x1
WIFI_STATE_ENABLING => 0x2
WIFI_STATE_ENABLED => 0x3
WIFI_STATE_FAILED => 0x4
MPT_EVENT_WIFI_STATE_CHANGED => 0x1f
MPT_EVENT_NETWORK_STATE_CHANGED => 0x20

The $1 subclass is a BroadcastReceiver which is setup by the parent class to listen for (android.net.wifi = a.n.w):
a.n.w.WIFI_STATE_CHANGED
a.n.w.STATE_CHANGE
a.n.w.supplicant.CONNECTION_CHANGE
a.n.w.supplicant.STATE_CHANGE

The custom function makeWifiInfo assembles a string by concatening several strings obtained via Android API calls with '0xa' as a separator (which represents a Linefeed in ASCII). The calls used include (each with the android.net.wifi.WifiInfo prefix abbreviated as a.n.w.wi):
a.n.w.wi.getBSSID(), a.n.w.wi.getIpAddress(), a.n.w.wi.getLinkSpeed(), a.n.w.wi.getRssi(), a.n.w.wi.getSupplicantState().

The research specimen contained 3000 rows

F001	ROWID
F002	getSampleTime()

F003	eventType (byte)
	In research data, this column was always either 31 (0x1f) or 32 (0x20):
	MPT_EVENT_WIFI_STATE_CHANGED => 0x1f
	MPT_EVENT_NETWORK_STATE_CHANGED => 0x20

F004	mState (int)
	In research data, this column was one of the following values: 0,1,2,3. The specifications are from the constants defined:
	WIFI_STATE_UNKNOWN => -0x1
	WIFI_STATE_DISABLING => 0x0
	WIFI_STATE_DISABLED => 0x1
	WIFI_STATE_ENABLING => 0x2
	WIFI_STATE_ENABLED => 0x3
	WIFI_STATE_FAILED => 0x4

F005	str (string) In research data, whenever F003=31, this value was blank. The majority of the time, when F003=32, this field has a value. Occasionally with the pairing of F003=32,F004=2 this value was blank.

4.19 t319 - Acc GPS Sate

The URI for t319 is CONTENT_ACC_GPS_SATE_URI. Other than its value assignment (MptMainLogProvider), I am unable to find any code which writes to this URI. However, there are 69 rows in t319 in the research specimen.

4.20 t320 - Exception Blobs

This table contents blobs (binary large objects) that are collected when an exception has occurred on the device. Exceptions are essentially errors. Reviewing exception details can assist with diagnosis and debugging as they are supposed to contain specifics about the errors that occur.

While the purpose of this table is fairly clear, the logic is quite complex and, given time constraints will not be researched fully. However it should be noted that entire SQLite databases have been found in F004 and F005.

The URI for t320 is CONTENT_EXCEPTION_BLOBS_URI.

4.21 t321 - Exception PCSync

The URI for t321 is CONTENT_EXCEPTION_PCSYNC_URI. There are no other references to this URI in the MPT source. The schema and triggers are created in MptMainLogProvider.

The research specimen for this table has no records, so I conclude that either that the corresponding code does not exist as of this version, or that it is intended to be written to from another application.

The circumstances are the identical for Exception Web, Exception SMS, and Exception MMS, and so the fields for these tables cannot effectively be reversed for this version of MPT.

4.22 - t322 - Exception Web

The URI for t322 is CONTENT_EXCEPTION_WEB_URI. See 4.21 for further information.

4.23 - t323 - Exception SMS

The URI for t323 is CONTENT_EXCEPTION_SMS_URI. See 4.21 for further information.

4.24 - t324 - Exception MMS

The URI for t324 is CONTENT_EXCEPTION_MMS_URI. See 4.21 for further information.

4.25 - t325 Acc Recent Activity

The URI for t325, CONTENT_ACC_RECENT_ACTIVITY_URI, is called exclusively by insertRecentActivityLog (MltMonitorService) and uses a *ContentValues* dictionary.

This data type is related to *RecentActivityWatcher* (and its subclasses), and the trigger interface *RecentActivityWatcher$RecentActivityTrigger*, which requires that the *onRecentActivityChanged* method. It corresponds with a handler->*what* property value of 0x14.

RecentActivityWatcher has one subclass, $1, which is a *BroadcastReceiver* [75]. It uses an intent filter of the type 'android.intent.action.TIME_TICK'. In response to broadcasts with that intent, it uses *android.content.Context.getSystemService("activity")*, which it casts as an android.app.ActivityManager and then invokes the ->*getRecentTasks* method passing the hard coded values 0x8 and 0x2 as parameters. The API documentation states these parameters represent *maxNum* and *flags*, respectively. As for *flags*, 0x2 corresponds to the flag RECENT_IGNORE_UNAVAILABLE, which provides a list that does not contain any recent tasks that currently not available to the user. Therefore, this statement is a request to ActivityManager for up to 8, available recent tasks.

$1 then creates a new Intent, adds action android.intent.action.MAIN and category android.intent.category.HOME, and then invokes resolveActivityInfo using the PackageManager which it resolved from the provided android.content.Context.

The reversed code from here becomes somewhat complex, but my interpretation is that it iterates each of the 8 tasks looking specifically for items that can be cast as ActivityManager.RecentTaskInfo. Over the course of this loop, successful casting will result in the appending of the application's package name and activity name (app name + activity) with a '/' in between them and a '\n' (newline) at the end. This constructed string will then be passed to onRecentActivityChanged which gets placed in the database.

4.26 - t326 Acc External Media

t326 logs system events relating to external media (SD cards, possibly USB storage via OTG).

The URI for t326 is CONTENT_ACC_EXTERNAL_MEDIA_URI. The only use of this URI is in the function *insertExternalMediaStateLog*, which uses a *ContentValues* object. It corresponds with a message->*what* property value of 0x17 (23).

This datatype involves ExternalMediaWatcher and its subclasses. The trigger interface

ExternalMediaStateTrigger requires one method, onExternalMediaStateChanged.

ExternalMediaWatcher$1 is a BroadcastReceiver that uses an intent filter for the following events, setting a different *v2* depending on the event.

v2 ultimately gets used in the triggering of *onExternalMediaStateChanged* as parameter 1. There are two custom functions, *getDfInfo()* and *getMountInfo()*, the return value of which get assigned to variables v1(F004) and v3 (F005) respectively.

F001	ROWID
F002	getSampleTime()
F003	eventType MEDIA_BAD_REMOVAL - 0x1 MEDIA_NOFS - 0x2 MEDIA_UNMOUNTABLE - 0x3 MEDIA_SCANNER_STARTED - 0x4 MEDIA_SCANNER_FINISHED - 0x5
F004	The value of getDfInfo(). df is a linux command which displays disk space information for file systems in use. getDfInfo() parses this
F005	The value of getMountInfo(). getMountInfo() uses the mount info to list active partitions. It searches specifically for lines matching /mnt/sdcard and /mnt/sdcard/_ExternalSD

4.27 - t327 Acc Data Activity

The URI for t327 is CONTENT_ACC_DATA_ACTIVITY_URI. This URI is referenced by insertDataActivityInfoLog, which uses MptItem and correlates with a message->*what* property value of 0x4 (4).

This is associated with the MptOnTelephonyWatcher trigger interface which requires two methods be implemented: *onDataActivityInfoChanged(byte, byte)* and *onTelephonyInfoChanged(byte, byte)*. This particular table is pertinent to the *onDataActivityInfoChanged* method. It uses MptItem with a message->*what* property of 0x4.

MptOnTelephonyWatcher$1 is an android.telephony.PhoneStateListener.

4.28 - t328 Acc GPS Satellite

The URI for t328 is CONTENT_ACC_GPS_SATELLITE_URI. Once defined, this URI is not referenced anywhere else in the MPT code. The research specimen for this table has no records, so I conclude that either the corresponding does not exist as of this version, or that it is intended to be written to from another application. The 9 fields in this table cannot effectively be reversed.

ACKNOWLEDGMENTS

Thanks to Or Begam of Cellebrite's research team for sharing his expertise reverse engineering best practices and tools, as well as providing additional samples of numerous other versions of the MLT APK.

REFERENCES

[1] apktool github repository (https://ibotpeaches.github.io/Apktool/)

[3] JesusFreke's smali github repository (https://github.com/JesusFreke/smali/)

[5] Switch statement, Wikipedia (https://en.wikipedia.org/wiki/Switch_statement)

[10] android.database.sqlite.SQLiteDatabase reference (https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html)

[15] android.telephony.TelephonyManager reference (https://developer.android.com/reference/android/telephony/TelephonyManager.html)

[20] android.os.Build reference (https://developer.android.com/reference/android/os/Build.html)

[25] android.util.DisplayMetrics (https://developer.android.com/reference/android/util/DisplayMetrics.html)

[30] android.telephony.SignalStrength (https://developer.android.com/reference/android/telephony/SignalStrength.html)

[35] 3GPP Technical Specification - AT command set for User Equipment (UE) - 27.007 V8.5.0

(https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1515)

[40] android.telephony.cdma.CdmaCellLocation (https://developer.android.com/reference/android/telephony/cdma/CdmaCellLocation.html)

[45] android.telephony.gsm.GsmCellLocation (https://developer.android.com/reference/android/telephony/gsm/GsmCellLocation.html)

[50] Cell ID Wikipedia (https://en.wikipedia.org/wiki/Cell_ID)

[55] Notepad++ (https://notepad-plus-plus.org/)

[60] smali Syntax Highlighting (http://androidcracking.blogspot.ca/2011/02/smali-syntax-highlighting-for-notepad.html)

[65] Bytecode Viewer github repo (https://github.com/Konloch/bytecode-viewer/releases)

[70] android.location.GpsStatus.Listener (https://developer.android.com/reference/android/location/GpsStatus.Listener.html)

[75] android.content.BroadcastReceiver (https://developer.android.com/reference/android/content/BroadcastReceiver.html)

[80] android.location.LocationManager (https://developer.android.com/reference/android/location/LocationManager.html)

[85] android.content.Intent (https://developer.android.com/reference/android/content/Intent.html)

[90] android.app.ActivityManager (https://developer.android.com/reference/android/app/ActivityManager.html)

[95] java.util.TImerTask (Java Platform SE Documentation) (https://docs.oracle.com/javase/7/docs/api/java/util/TimerTask.html)

[100] android.app.ActivityManager.MemoryInfo (https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo.html)

[105] android.content.ContentProvider (https://developer.android.com/reference/android/content/ContentProvider.html)

[110] android.app.Service (https://developer.android.com/reference/android/app/Service.html)