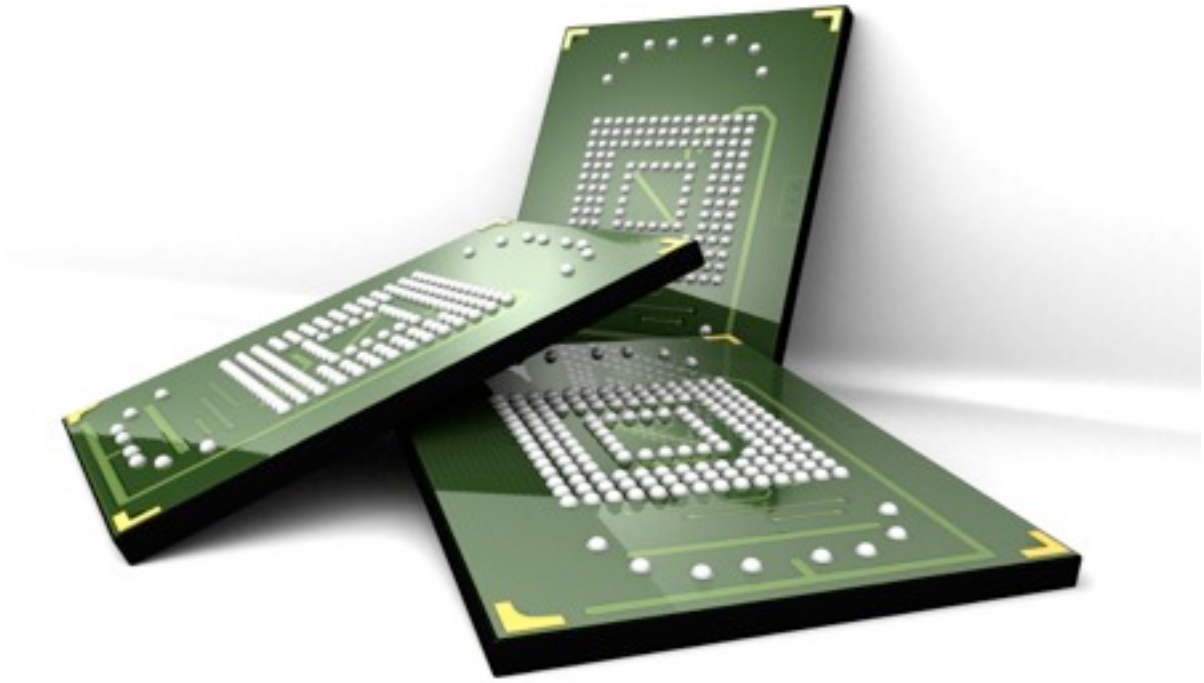
N A N D - X P L O R E

Hiding and Finding Data with NAND Flash Error Codes

INITIAL FINDINGS AND DEVELOPMENT PLAN



A DELIVERABLE PRODUCT OF:

MONKWORKS, LLC

JOSH "M0NK" THOMAS

m0nk.omg.pwnies@gmail.com • @m0nk_dot • monkworks.llc@gmail.com

Table of Contents

Overview of Research

The NAND-Xplore project was funded by the DARPA Cyber Fast Track initiative to investigate proof of concept capabilities of NAND Flash hardware to:

1) Hide files and programs from end users, operating systems and forensics software.
2) Detect when files and programs have been hidden from end users, operating systems and forensics software.

Review of the project and goals

(The following section is an excerpt from the initial proposal, included herein for context)

Over the past few years, consumer electronics hardware has quickly embraced solid-state memory as the primary means of data storage. These devices are ubiquitous across our culture; from smart phones to laptops to USB memory sticks to GPS navigation devices. On a daily basis we carry multiple NAND based devices in our pockets without considering the security implications, taking for granted the security solely because they reside on our person.

While some of the specific devices have received rigorous attention from the information security community, it is arguable if the entire class of NAND based devices has been run through the exploitation gauntlet. The smart phone based solutions, most notably the Android and iOS platforms; have received heavy attention of course; but only from an operating system and above viewpoint.

The NAND-Xplore project is an attempt to explore the lower levels of these systems, starting at the "bare metal" well below the operating environment. The project will attempt to expose weaknesses in the actual NAND hardware and implementation architectures and showcase the vulnerable underpinnings across the spectrum of NAND based platforms. The project will culminate in the delivery of two "proof of concept" tools:

- NAND-Hide
- NAND-Find

The NAND-Hide tool will inject files onto physical NAND devices that cannot be viewed by either the operating system or any typical forensics software. The tool will accomplish this task by subverting control of the NAND hardware directly and marking sections of memory as bad and unreadable. While this sounds simple on paper, the task itself is non-trivial. This tool will allow an attacker or developer of malicious software to remain resident on a device across reboots in a highly concealed manner.

The NAND-Find tool will attempt to analyze a platform at a low level and detect hidden files residing on the NAND itself. While the proposer is unaware of any existing malicious software using these techniques, the NAND-Find tool will expose any files hidden on a device using these or similar techniques.

Initial Selection of Devices

During the initial research performed when crafting the NAND-Xplore proposal, 3 target devices were selected for potential development of the Proof of Concept (POC) tools described. The devices were selected as exemplar devices to cover the 3 general and broad categories of modern devices that utilize disparate types of NAND hardware.

- The LG Nexus 4 smartphone was selected as an exemplar smart phone.
- The Samsung Google Chromebook was selected as an exemplar Chromebook and as a representative example of a typical laptop or ultrabook device.
- The Microsoft Surface RT device was selected as an exemplar tablet.

Microsoft Surface RT

Samsung Google Chromebook

LG Nexus 4

Additional Chromebook Devices Explored

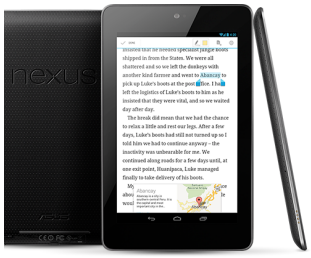
In addition to the Samsung Google Chromebook, the researcher performed an initial triage on the original Google branded Chromebook prototype, the CR-48. Details of this analysis will be contained within the Chromebook findings below.

The CR-48 Google Chromebook

The researcher did not explore the other Chromebooks on the market but can make general assumptions based on online research into the platforms. The Acer brand Chromebooks utilize a spinning platter hard drive for data storage and are thus irrelevant to this research. The older model Samsung devices are expected to behave in a highly similar manner to the XE303 model as they are simply prior iterations of the device.

Additional Android Devices Explored

In addition to the LG Nexus 4, the researcher performed various depths of exploration on a handful of other Android based devices that were available from previous projects. As with the Chromebooks, the results of these devices will be catalogued in later sections. The devices analyzed included the following:



Asus Nexus 7

Asus Transformer Prime

HTC Droid Incredible

HTC Nexus Google One

Kindle Fire

Motorola Droid

Motorola Droid X

Motorola RAZR

Motorola Xoom

Oppo Finder x907

Samsung Epic Touch 4G

Samsung Galaxy Nexus

Samsung Google Nexus S

Sony Xperia Arc S

Sony Xperia ION

Sony Xperia Play

Sony Tablet P

Sony Tablet S

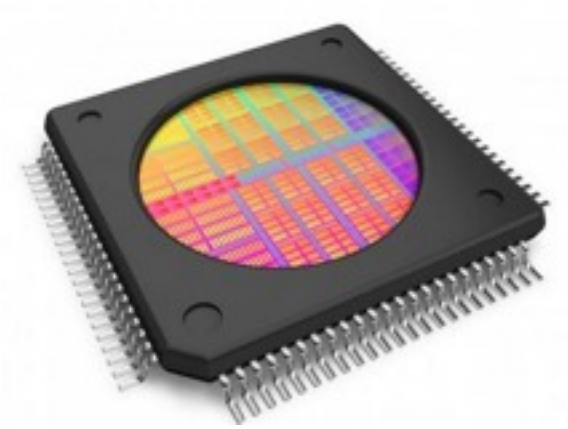
Sony Xperia X10a

Sony Xperia X10 Mini Pro

Toshiba Thrive

A Deeper Understanding of How NAND Functions

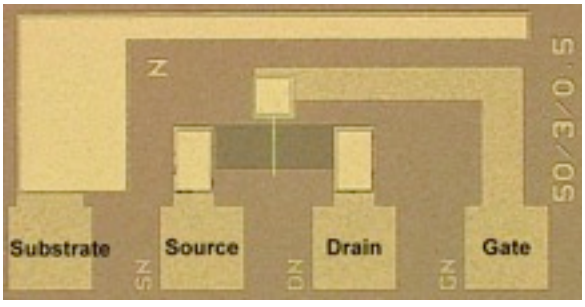
This section is intended to provide a quick, 5 minute primer on how NAND Flash works at a physical layer and not an in depth analysis. More information can be found on any of the manufacturers' websites or the JEDEC/ONFI standards committees websites.



Sample NAND Prototype chip with visible blocks and pages

Hardware functionality of actual NAND Flash

In the most basic sense, NAND devices store individual bits of data in a multidimensional array of floating-gate transistors. The floating-gate transistors allow each cell to trap individual electrons, thus keeping or removing a charge. It is this charge that corresponds to a single 0 or 1 for the device. The multiplexed transistor design, coupled with the concept of Fowler- Nordheim tunnel injection and release, allows this grid of floating gates to access cells at a single bit level. In layman's terms, consider the NAND flash to behave as a highly dense, addressable LED array.



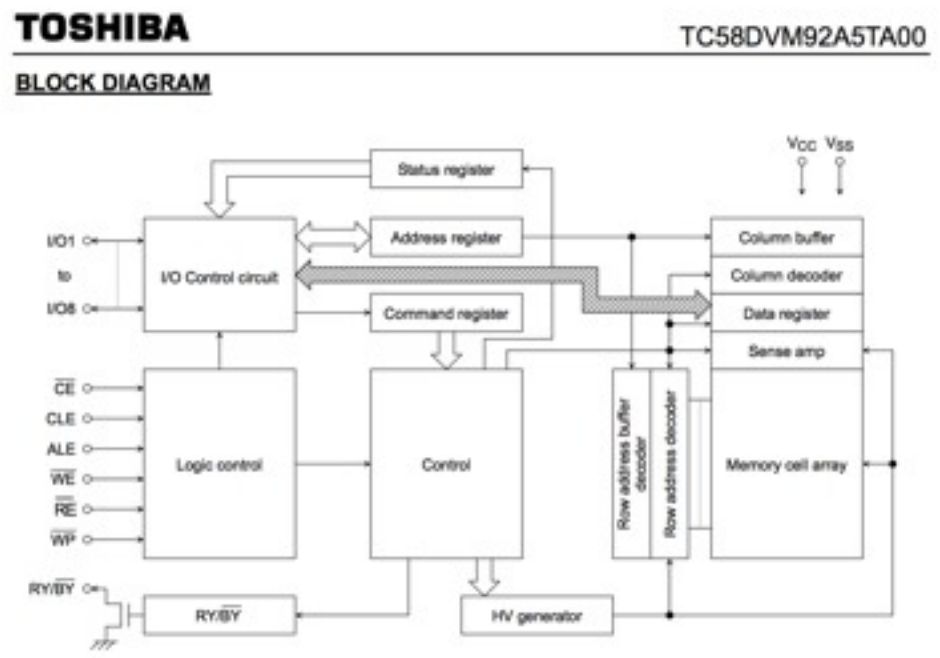
A Simple NAND Circuit

Each individual flash cell is contained in a collection designated as a page. Pages on NAND devices are typically collections of 512, 2048 or 4096 bytes. In turn, each page is collected into a construct known as a block. NAND blocks typically follow an exponential based size paradigm and can range from 16 KB to 512 KB.

While the grid architecture of NAND flash allows for addressing at the single bit level, such accuracy comes with a hard set of limitations.

- All bits on the device default to and are initially set to a 1.The shift from a 1 to a 0 is a simple electronic pulse to open the gate and dump the stored electron. Sadly, shifting the other direction (from a 0 to a 1) is non trivial and cannot be preformed at the bit level, only at the block level. As such, shifting a stored byte of 1111 1111 to 1010 1010 is trivial but the reverse would entail erasing and entire block of 512 KB.
- The physical floating-gate transistors are fragile and slowly wear down over time. Typical industry expectations are that each gate can survive around 100,000 state changes before becoming unreliable and unstable. Once a block has become unstable, the NAND controller has the ability to mark it "bad". This designation will ensure the block is removed from rotation and can no longer be read or accessed automatically.
- At the gates wear over time, charge leakage can occur. This leakage will corrupt neighboring cells and their stored information. Charge leakage can also occur with exceptionally high levels of repeated reading even without writing to a cell. This is mostly due to the power utilized across the grid to query a specific cell.

Given these limitations, NAND designers and manufacturers introduced automated leveling across the devices. This process attempts to distribute digital information across the hardware in an even manner, not allowing any single bit, page or block to be utilized more than another. The leveling software will also copy highly accessed information around the NAND to discourage charge leakage. If one has the correct tools, they can see this phenomenon by low- level analysis of a NAND. Typically, a forensics analyst can view multiple histories of a file because the NAND flash controller will elect to copy the entire file to a new block of NAND instead of modify the existing imprint. These older versions of the file stay resident until the block is reset and new data is written. This, as well as all other NAND interactions, is managed by the NAND controller hardware. This NAND controller is also a main culprit for why writing successive 0's and 1's repeatedly over an entire device is meaningless to the technology, typically because the NAND controller will simply disallow such wasteful access to the memory.



Toshiba NAND Reference Design with NAND Controller

The final applicable detail about NAND flash pertains to mass production yields, transistor size and quality control. Manufacturers are constantly pushing the size of this hardware to be well below a 100% reliable component threshold. As such, devices are known to contain and ship with bad and unusable sections. These sections, much like the blocks that have exhausted their maximum number of times data can be written, are marked as "bad" at the controller level using a collection of NAND flash based error codes. These blocks are simply considered unusable by the overall system and are removed from the addressable space of the memory by the NAND controller. The NAND controller supports this functionality by keeping an active map of the hardware detailing valid and error prone blocks.

Lastly, it should be noted that most but not all embedded NAND flash devices contain a hardware based NAND controller. Those devices that do not contain controlling hardware, such as smart cards, USB storage devices and the like, expect the controlling operating system to mark, flag, control and manipulate the hardware directly. As such, most modern operating systems have a basic understanding of NAND error and correction codes. For the devices that do contain hardware-based controllers, the operating system and hardware drivers preform read and write operations in a similar manner to their older magnetic platter counterparts.

Overview of the NAND Flash Standards

The 2 main standards bodies relevant to NAND are JEDEC and ONFI.

Development NAND Breakout with a standard TSOP connection

The JEDEC (Joint Electronic Device Engineering Council) committee is primarily concerned with ensuring the various vendors and manufacturers of NAND Flash hardware conform to certain chip package hardware standards. JEDEC is also concerned with ensuring general interoperability between manufacturers and NAND designs. JEDEC provides this services for numerous types of hardware and is far from a NAND specific committee.

The ONFI (Open NAND Flash Interface) group is a governing body for NAND Flash specific interface standards. The group intends to dictate how NAND will interface with other hardware and (to some extent) other software in the wild.

In general, most NAND devices connect to other hardware with either a TSOP (Thin, Small outline package) or BGA (Ball Grid Array) connection. The referenced standards dictate the footprint and layout of the hardware In typical situations, embedded NAND is delivered on a 169 ball BGA package.

Standard Types of NAND to Board connections

How NAND Devices are Utilized by the Linux Kernel Derivatives

** This section will make some simplifications to explanations about NAND and other data storage hardware from a Linux perspective. Some of the fine grain details will be ignored with the intent of simplifying the information for consumption. The intent is solely to remove or gloss over information that could confuse the arguments with tangential details.*

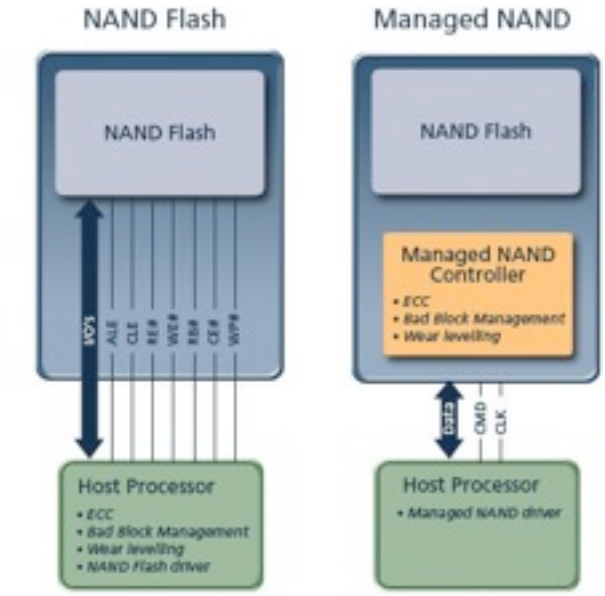
Raw NAND vs. FTL Technologies

NAND Flash can come in a variety of configurations when manufactured. In specific relation to this research we can categorize them as such:

- Raw NAND
- NAND + FTL (Managed NAND)

Raw NAND Flash is a slab of NAND storage in its most basic form and all management of the hardware and storage interactions are performed in software outside of the NAND. The Linux kernel utilizes the MTD (Memory Technology Device) subsystem to interact with these devices. This grouping contains only bare NAND and other MTD devices. To add to the confusion, some Raw NAND devices to have embedded ECC (error correction) and simple block management. The differentiation in this instance is the linux kernel is treated as the master controller with the embedded processing simply supporting.

NAND + FTL devices contain an on package NAND controller that manages the slab of NAND flash internal to the chip. This controller will manage bad blocks, wear leveling and data access internally and provide a FTL (Flash Transition Layer) interface to outside software such as the linux kernel. The FTL presents the NAND hardware as a standard block device externally. Though there are significant differences in implementation, this broad grouping contains MMC, eMMC, SD and SSD devices.



Raw NAND vs. Managed NAND (FTL)

While NAND + FTL has a large market share of embedded devices, it presents a handful of problems for the NAND-Hide and NAND-Find POC implementations. Namely, the internals of FTL implementations are both vendor specific and considered vendor secret IP. Given the intent of this research is to showcase inherent flaws in physical NAND devices, the FTL based devices will be considered out of scope for the initial POC. The raw NAND POC that will be provided during this research would also theoretically work on all FTL devices, it would just need to be customized to interact with the closed source embedded solution.

As stated, this research could be extended to MMC/eMMC FTL based NAND devices (and the researcher would be highly interested in doing so), but will be initially limited to raw NAND for the sake of the POC tools and respect for vendor trade secrets.

The non-complete list major manufacturers and vendors of relevance in the NAND space are Micron, Samsung, Toshiba and SanDisk.

The MTD Subsystem - Working without a controller

The MTD subsystem is the Linux kernel equivalent of an embedded NAND controller. The system controls how the raw NAND is accessed, how it is erased and how error corrections and block management are performed. The MTD project contains a full suite of memory storage interfaces and the working details will be captured later in this document. The project is a fully mature, open source project located at http://www.linux-mtd.infradead.org. For this research, we are specifically interested in the

bad block management interfaces provided by the MTD subsystem and to a lesser extent the basic ECC functionality.

Overview of Android / Linux Commands Used

For all devices analyzed (except the Microsoft Surface RT Tablet) a general collection of linux commands and tools can be utilized to explore a device in respect to NAND utilization. While the commands are general in nature, the following sample output will be from the Sony Xperia Arc S (Model LT18a) device running the stock Xperia Android build (4.1.B.0.587) and _the 2.6.32.9-perf (BuildUser@BuildHost #1) kernel (unless otherwise noted).

The first thing useful for the research is to determine what partitions are actually on the NAND during runtime. This task can be accomplished in a variety of ways, and each variant will expose different information.

Initially, it is worthwhile to simply view the files and system setup directly. To do this, one can simply log into the device using ADB (Android Debug Bridge) and view the configuration files. While not necessary for these initial commands, upgrading privileges to the root user and installing the BusyBox toolchain always makes working on a native device easier and as such it is recommended.

To quickly sample what partitions are resident and utilized, we can view the standard partition table:

```
$ cat /proc/partitions
major minor  #blocks  name

  31        0     409600 mtdblock0
  31        1       6144 mtdblock1
  31        2     103936 mtdblock2
  31        3     430080 mtdblock3
 179        0    7778304 mmcblk0
 179        1    7777280 mmcblk0p1
```

Code Block 1 - The Partition Table

As we can see from the output, the Arc S mounts 4 MTD based block partitions and 2 MMC based ones.

We can gain more insight into the NAND layout by observing the MTD mapping directly:

```
$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 19000000 00020000 "system"
mtd1: 00600000 00020000 "appslog"
mtd2: 06580000 00020000 "cache"
mtd3: 1a400000 00020000 "userdata"
```

Code Block 2 - The MTD Table

To observe brevity and not fill this document with copies of small files, the other files with interesting data relevant to the NAND usage on Android are:

```
/init.rc

/proc/devices
/proc/diskstats
/proc/filesystems
/proc/iomem
/proc/ioports
/proc/mtd
/proc/partitions
/proc/slabinfo
/proc/yaffs
```

Code Block 3 - Listing Files of Interest for the NAND

Reading through the full listings will give the reader a deeper understanding of how the NAND hardware is being utilized by Android during runtime.

For introspection of the boot process and how the kernel and NAND interact during initialization, we can examine the boot sequence directly. To execute this, simply run the following command with the phone plugged into USB but turned off. When the phone is powered on, the dmesg logs will flow to the terminal.

```
$ adb wait-for-device && adb shell dmesg
```

Code Block 4 - Grabbing the Boot Information

As the boot process itself is intricate and detailed, I have attempted to point out only the relevant logs from kernel / NAND interactions. Of main interest for the NAND-Hide POC tool is the partitioning and mapping from atag of the mtd devices.

```
$ adb wait-for-device && adb shell dmesg
...
<4>[    0.000000] Machine: mogami
<6>[    0.000000] Partition (from atag) system -- Offset:2f4 Size:c80
<6>[    0.000000] Partition (from atag) appslog -- Offset:f74 Size:30
<6>[    0.000000] Partition (from atag) cache -- Offset:fa4 Size:32c
<6>[    0.000000] Partition (from atag) userdata -- Offset:12d0 Size:d20
...
<6>[    2.897521] Trying to unpack rootfs image as initramfs...
<6>[    2.933807] Freeing initrd memory: 992K
<6>[    2.934570] smd probe
<6>[    2.934600] smd_core_init()
<6>[    2.934631] smd_core_init() done
<6>[    2.934631] smd_alloc_loopback_channel: 'local_loopback' cid=100
<6>[    2.934783] last_amsslog: Log not present
<6>[    2.934997] smd_alloc_channel() 'DS' cid=0
<6>[    2.935058] smd_alloc_channel() 'DIAG' cid=1
<6>[    2.935089] smd_alloc_channel() 'RPCCALL' cid=2
<6>[    2.935150] smd_alloc_channel() 'DATA1' cid=7
<6>[    2.935180] smd_alloc_channel() 'DATA2' cid=8
<6>[    2.935241] smd_alloc_channel() 'DATA3' cid=9
<6>[    2.935272] smd_alloc_channel() 'DATA4' cid=10
<6>[    2.935333] smd_alloc_channel() 'DATA5' cid=11
<6>[    2.935363] smd_alloc_channel() 'DATA6' cid=12
```

```
<6>[    2.935394] smd_alloc_channel() 'DATA7' cid=13
<6>[    2.935455] smd_alloc_channel() 'DATA8' cid=14
<6>[    2.935485] smd_alloc_channel() 'DATA9' cid=15
<6>[    2.935546] smd_alloc_channel() 'DATA11' cid=17
<6>[    2.935577] smd_alloc_channel() 'DATA12' cid=18
<6>[    2.935638] smd_alloc_channel() 'BRG_1' cid=28
<6>[    2.935668] smd_alloc_channel() 'DAL00' cid=38
<6>[    2.935729] smd_alloc_channel() 'BRG_0' cid=39
<6>[    2.935760] smd_alloc_channel() 'DATA5_CNTL' cid=40
<6>[    2.935821] smd_alloc_channel() 'DATA6_CNTL' cid=41
<6>[    2.935852] smd_alloc_channel() 'DATA7_CNTL' cid=42
<6>[    2.935913] smd_alloc_channel() 'DATA8_CNTL' cid=43
<6>[    2.935943] smd_alloc_channel() 'DATA9_CNTL' cid=44
<6>[    2.936004] smd_alloc_channel() 'DATA12_CNTL' cid=45
<3>[    2.936096] Notify: smsm init
<6>[    2.937377] SMD Packet Port Driver Initialized.
<6>[    2.937683] SMD: ch 2 0 -> 1
...
<6>[    2.969268] msm_fb_probe:  phy_Addr = 0x2ef9000 virt = 0xd0800000
<6>[    2.969573] MDP HW Base phy_Address = 0xa3f00000 virt = 0xd0100000
...
<6>[    4.431549] msm_nand_probe: phys addr 0xa0200000
<6>[    4.436096] msm_nand_probe: Dual Nand Ctrl in ping-pong mode
<6>[    4.441711] msm_nand_probe: dmac 0x7
<6>[    4.445281] msm_nand_probe: allocated dma buffer at ffc7c000, dma_addr 4f086000
<6>[    4.453033] status: c00020
<6>[    4.455261] nandid: 55d1b32c maker 2c device b3
<6>[    4.459777] ONFI probe : Found an ONFI compliant device MT29F8G16ADBDAH4    ,
<6>[    4.466888] Found a supported NAND device
<6>[    4.470855] NAND Id  : 0x55d1b32c
<6>[    4.474182] Buswidth : 16 Bits
<6>[    4.477294] Density  : 1024 MByte
<6>[    4.480590] Pagesize : 2048 Bytes
<6>[    4.483886] Erasesize: 131072 Bytes
<6>[    4.487365] Oobsize  : 64 Bytes
<6>[    4.490478] 2nd_bbm  : 0
<6>[    4.493011] CFG0 Init  : 0xa85408c0
<6>[    4.496551] CFG1 Init  : 0x0012745a
<6>[    4.500122] ECCBUFCFG  : 0x00000203
<5>[    4.503692] Creating 4 MTD partitions on "msm_nand":
<5>[    4.508636] 0x000005e80000-0x00001ee80000 : "system"
<5>[    4.707489] 0x00001ee80000-0x00001f480000 : "appslog"
<5>[    4.710754] 0x00001f480000-0x000025a00000 : "cache"
<5>[    4.760131] 0x000025a00000-0x00003fe00000 : "userdata"
...
<3>[    6.102142] mmc0: No card detect facilities available
<6>[    6.106658] mmc0: Qualcomm MSM SDCC at 0x00000000a3000000 irq 96,0 dma 8
<6>[    6.113098] mmc0: 8 bit data mode disabled
<6>[    6.117187] mmc0: 4 bit data mode enabled
<6>[    6.121185] mmc0: polling status mode disabled
<6>[    6.125610] mmc0: MMC clock 144000 -> 49152000 Hz, PCLK 96000000 Hz
<6>[    6.131866] mmc0: Slot eject status = 0
<6>[    6.135681] mmc0: Power save feature enable = 1
<6>[    6.140197] mmc0: DM non-cached buffer at ffc80000, dma_addr 0x4f119000
<6>[    6.146789] mmc0: DM cmd busaddr 0x4f119000, cmdptr busaddr 0x4f119300
<6>[    6.214355] mmc1: Qualcomm MSM SDCC at 0x00000000a3100000 irq 100,523 dma 8
<6>[    6.214477] mmc1: 8 bit data mode disabled
<6>[    6.214569] mmc1: 4 bit data mode enabled
<6>[    6.214630] mmc1: polling status mode disabled
<6>[    6.214721] mmc1: MMC clock 144000 -> 49152000 Hz, PCLK 96000000 Hz
<6>[    6.214813] mmc1: Slot eject status = 0
<6>[    6.214874] mmc1: Power save feature enable = 1
<6>[    6.214965] mmc1: DM non-cached buffer at ffc81000, dma_addr 0x4f11b000
```

```
<6>[    6.215087] mmc1: DM cmd busaddr 0x4f11b000, cmdptr busaddr 0x4f11b300
...
<6>[    6.637329] yaffs: dev is 32505858 name is "mtdblock2" rw
<6>[    6.637451] yaffs: passed flags ""
<7>[    6.637512] yaffs: yaffs: Attempting MTD mount of 31.2,"mtdblock2"
<7>[    6.639678] yaffs: yaffs_read_super: is_checkpointed 1
<6>[    6.694458] yaffs: dev is 32505856 name is "mtdblock0" rw
<6>[    6.694549] yaffs: passed flags ""
<7>[    6.694610] yaffs: yaffs: Attempting MTD mount of 31.0,"mtdblock0"
<6>[    6.734680] bq27520 0-0055: IT Enable confirmed to be set.
<7>[    6.820953] yaffs: yaffs_read_super: is_checkpointed 1
<3>[    6.855377] init: cannot open '/initlogo.rle'
<6>[    6.858215] yaffs: dev is 32505858 name is "mtdblock2" rw
<6>[    6.858306] yaffs: passed flags ""
<7>[    6.858367] yaffs: yaffs: Attempting MTD mount of 31.2,"mtdblock2"
<7>[    6.860443] yaffs: yaffs_read_super: is_checkpointed 1
<4>[    7.012176] mmc1: host does not support reading read-only switch. assuming
write-enable.
<6>[    7.012908] mmc1: new high speed SDHC card at address 0002
<6>[    7.013153] sdio_al mmc1:0002: Probing..
<6>[    7.013305] mmcblk0: mmc1:0002 00000 7.41 GiB
<6>[    7.013488]  mmcblk0: p1
<6>[    7.768249] bq27520 0-0055: bq27520_battery_info_setting() type=3 temp=270
status=0
<6>[    7.984466] yaffs: dev is 32505856 name is "mtdblock0" rw
<6>[    7.984558] yaffs: passed flags ""
<7>[    7.984619] yaffs: yaffs: Attempting MTD mount of 31.0,"mtdblock0"
<7>[    8.111175] yaffs: yaffs_read_super: is_checkpointed 1
<6>[    8.124114] yaffs: dev is 32505859 name is "mtdblock3" rw
<6>[    8.124206] yaffs: passed flags ""
<7>[    8.124267] yaffs: yaffs: Attempting MTD mount of 31.3,"mtdblock3"
<7>[    8.161956] yaffs: yaffs_read_super: is_checkpointed 0
<6>[    8.162200] yaffs: dev is 32505858 name is "mtdblock2" rw
<6>[    8.162322] yaffs: passed flags ""
<7>[    8.162384] yaffs: yaffs: Attempting MTD mount of 31.2,"mtdblock2"
<7>[    8.164459] yaffs: yaffs_read_super: is_checkpointed 1
<6>[    8.175048] yaffs: dev is 32505857 name is "mtdblock1" rw
<6>[    8.175140] yaffs: passed flags ""
<7>[    8.175201] yaffs: yaffs: Attempting MTD mount of 31.1,"mtdblock1"
<7>[    8.191528] yaffs: yaffs_read_super: is_checkpointed 0
...
```

Code Block 5 - dmesg on bootup

Analysis of Android Devices

Overall Hardware Analysis of Android Based Devices

With a myriad of hardware manufacturers and iterations of devices the Android platform is ripe with variation on NAND utilization. As vendors and device manufacturers tend to reuse both code and hardware designs when possible, this document will break the analysis down by overall manufacturer instead of actual device. While not exhaustive, this document will attempt to catalogue the major manufacturers of handsets and how they utilize NAND Flash at a high level. If a finding is device specific, it will be directly referenced as such. Also, as it would be highly cost and time prohibitive to test

all Android devices, each vendor section will list the specific devices analyzed for the device manufacturer. Devices not specifically listed were not analyzed and no assumptions should be made about them.

Lastly, Android devices have a somewhat awkward naming scheme that should be explained. Each device has a public facing name for marketing and sales purposes, such as the *Samsung Galaxy Nexus* (SGN). Internally to the AOSP each device also has a code name, so the SGN can be referred to as Tuna or Toro. To make matters more confusing, many vendors actually produce multiple variants of the hardware to accommodate different cellular service providers (AT&T, T-Mobile, Verizon, Sprint and so on), and with each hardware variant a new code name is given. So in the end, the name SGN can actually refer to either the Toro (Verizon), ToroPlus (Sprint) and Maguro (AT&T) hardware. In this document, I will attempt to reference both the public name, the code name and the model number where such information would be beneficial. As an example, the SGN variants will be referred to as either:

- The Samsung Galaxy Nexus if the information applies to all the devices in the family
- The SGN Toro/SGH-i525 if the information only applies to the Verizon CDMA variant of the hardware
- The SGN ToroPlus/SPH-L700 if the information only applied to the Sprint CDMA Variant of the hardware
- The SGN Maguro/GT-i9250 if the information only applies to the AT&T / T-Mobile GSM variant of the hardware

The Samsung Android Devices

The Samsung devices examined did not utilize a MTD partition for the main process space, nor did the compiled kernel appear to log many interactions with the MTD linux sub-system. This is probably because Samsung utilizes a proprietary and closed source scheme named BML that wraps and covers the lower level MTD system into MMC access or for boot protections. The filesystem for most Samsung devices does show traces of MTD usage but it does not seem to be utilized heavily.

Devices analyzed:

- The Samsung Galaxy Nexus: (all variants)
- Samsung Nexus S
- The Samsung Epic Touch 4G

Additional queries to the community revealed that the Samsung Galaxy S III also mostly bypasses the use of MTD partitions and it is expected all Samsung devices do.

Samsung Android devices tend to use Samsung branded NAND.

The SGN Toro/SGH-i525 device does contain a single small MTD partition accessible through the standard MTD system:

```
$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00001000 "w25q80"
```

Code Block 6 - Samsung SGN MTD partition

```
$cat /proc/partitions
major minor  #blocks  name

  31        0       1024 mtdblock0
 179        0   15388672 mmcblk0
 179        1        128 mmcblk0p1
 179        2       3584 mmcblk0p2
 179        3      20480 mmcblk0p3
 179        4       8192 mmcblk0p4
 179        5       4096 mmcblk0p5
 179        6       4096 mmcblk0p6
 179        7       8192 mmcblk0p7
 259        0      12224 mmcblk0p8
 259        1      16384 mmcblk0p9
 259        2     669696 mmcblk0p10
 259        3     442368 mmcblk0p11
 259        4   14198767 mmcblk0p12
 259        5         64 mmcblk0p13
 179       16       2048 mmcblk0boot1
 179        8       2048 mmcblk0boot0
```

Code Block 7 - Samsung SGN Partition table

```
$cat /proc/devices
Character devices:
  1 mem
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
 10 misc
 13 input
 21 sg
 29 fb
 89 i2c
 90 mtd
108 ppp
116 alsa
122 tf_driver
122 tf_ctrl
128 ptm
136 pts
166 ttyACM
180 usb
188 ttyUSB
```

```
189 usb_device
216 rfcomm
247 roccat
248 tiler
249 ttyGS
250 pvrsrvkm
251 ttyO
252 rpmsg_omx
253 ttyFIQ
254 rtc

Block devices:
  1 ramdisk
259 blkext
  7 loop
  8 sd
 31 mtdblock
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
254 device-mapper
```

Code Block 8 - The SGN Devices on a stock device

As we can see, the MTD device ”w25q80” is mapped to both a character device and a simple block device. As stated, other Samsung devices behave in a similar manner.

The LG Android Devices

The Newest iteration of the Google blessed AOSP Nexus line is the LG Nexus 4. The phone carries the model number E960 and the code name “mako”. The device contains the THGBM5G6A2JBAIR, a Toshiba branded NAND. The stock device, nor any of the after-market ROMS utilize any MTD partitions directly.

No other LG devices were available for analysis, but it can be assumed that if LG devices are harnessing Toshiba NAND across the board then they are all managed / NAND + FTL.

The Motorola Android Devices

Of the Motorola devices examined, roughly half harnessed MTD partitions for the main process space, the other half used only MMC/eMMC. The utilization of MTD was most significant across the Droid family (and lacking in the Xoom).

Devices Analyzed:

- Motorola Droid
    - Motorola Droid X
    - Motorola RAZR / RAZR Maxx
    - Motorola Xoom

The HTC Android Devices

Older HTC devices utilized MTD based access almost exclusively, but the company has been moving away from raw NAND into managed solutions on their newer devices.

Devices Analyzed:

- HTC Droid Incredible
- HTC Google Nexus One

The Sony Android Devices

The Sony Xperia offerings (referred to as Xperia 11 and 12 devices in the community) utilize MTD based NAND access for over half of the mapped storage on the device. While MTD is oddly lacking from the Tablet space, the smart phone offerings are ripe with potential for the NAND-Xplore POC tools. An added bonus (to the researcher at least) is the naming convention of Xperia and Xplore sort of match.

Devices Analyzed:

- Sony Xperia Arc S
- Sony Xperia Ion
- Sony Xperia Play

- Sony Tablet P
- Sony Tablet S
- Sony Xperia x10a
- Sony Xperia x10 Mini Pro

The Xperia Arc S will be selected as the development device of choice for the NAND-Xplore project, and will be detailed in that section below.

The Asus Android Devices

Asus devices showed mixed utilization of MTD access. The Nexus 7 harnessed the subsystem for a boot partition but not for the main system storage. MTD was absent from the Transformer Prime device analyzed.

Devices Analyzed:

- Asus Nexus 7
- Asus Transformer Prime

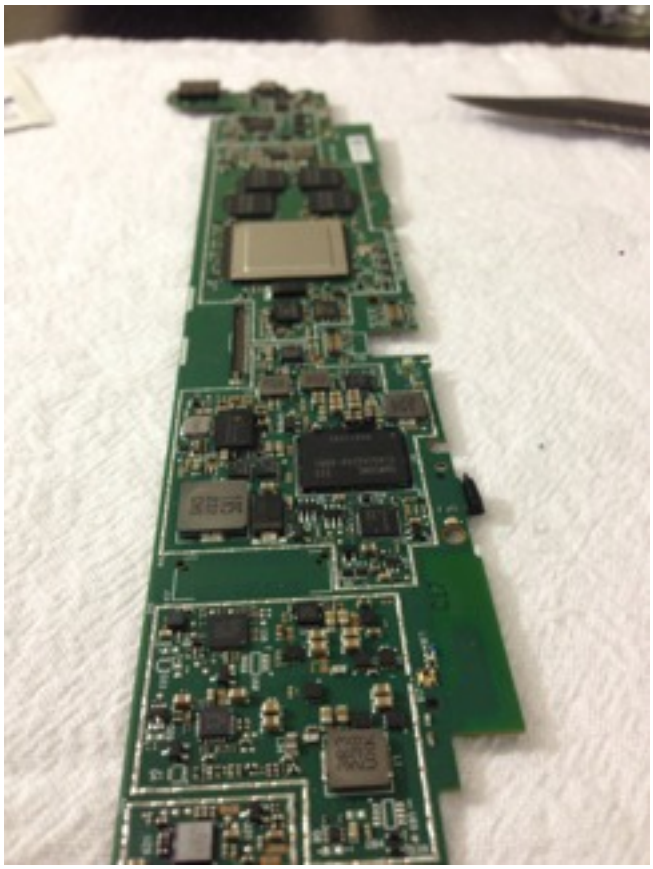
Analysis of Microsoft Surface RT Tablet

Overview analysis of the Surface RT platform

While expected, the Microsoft Surface platform proved to be difficult to develop for. The analysis did prove interesting, but the toolchains simply don't exist to explore this device with the same rigor as the linux based devices.

The Surface uses a Samsung branded KLMBG4GE4A Flash chip, part of the MoviNAND line. While documentation is sparse due to mass production & trade secrets, the same chip is used in the Samsung Galaxy Note "phablet" device. Sadly, the side channel introspection stops there as the Galaxy Note follows the Samsung path and uses BML.

During teardown, the researcher did note the Surface motherboard shipped with a handful of debug pinouts around the central processor and memory. These pads were not explored with JTAG or ICE due to time.



Main Board from the Microsoft Surface

Samsung NAND utilized in the Surface

While not relevant to this research, the most interesting find during the Microsoft Surface teardown was this awkward wire. It appears to jump a connection across the entire device for some unknown reason.

The odd connecting wire in the MS Surface

Analysis of Google Chromebook Devices

Overview analysis of the Chromebook / Chromium project

Similar to the Android AOSP project, at the heart of the Chromium project lies the Linux kernel. This base framework provides the same MTD subsystem as the sister Android project. In reality, at least from the perspective of this research, the 2 projects can be considered identical.

Chromebook hardware analysis

Unlike the Android project, there exist very few Chromebook devices. This research analyzed the NAND Flash based Chromebook devices for MTD based utilization.

The Samsung "Daisy/Snow" device is built around a SanDisk SDIN7DU2-16G NAND Flash chip in a standard 169 pin BGA package. This BGA chip can act in both managed and unmanaged (raw) mode and SanDisk provides a simplified TSOP package for raw access. The researcher utilized this chip when exploring the Segger NAND Evaluation harness.

The "Daisy/Snow" device utilizes a single MTD mapping during machine boot for the loader. Once the kernel is loaded, that MTD access is removed and the device remapped as MMC.

```
 # LoadKernel() debug data (not in print-all)
vdat_timers            = LFS=424218,653214 LF=1441289,1632796 LK=1891610,4402591 #
Timer values from VbSharedData
wpsw_boot              = 1                              # Firmware write protect
hardware switch position at boot
wpsw_cur               = 1                              # Firmware write protect
hardware switch current position
+ rootdev -s
/dev/mmcblk0p5
+ ls -aCF /root
./  ../  .force_update_firmware
+ ls -aCF /mnt/stateful_partition
./   .developer_mode  .tpm_status  encrypted/     encrypted.key  lost+found/
../  .tpm_owned       dev_image/   encrypted.block  home/      unencrypted/
+ cgpt show /dev/mmcblk0
       start        size    part  contents
           0           1          PMBR (Boot GUID: 2805DB23-3877-D14B-
BBC7-2EF643847E5B)
           1           1          Pri GPT header
           2          32          Pri GPT table
      282624    22073344       1  Label: "STATE"
                                   Type: Linux data
                                   UUID: 7BD0DDB5-1FCA-774D-A399-881672E0C572
       20480       32768       2  Label: "KERN-A"
                                   Type: ChromeOS kernel
                                   UUID: EE1C9957-CFCB-564F-81A1-49D451F908D4
                                   Attr: priority=1 tries=0 successful=1
    26550272     4194304       3  Label: "ROOT-A"
                                   Type: ChromeOS rootfs
                                   UUID: A54FDA70-F2AE-CA45-A2C6-31BB9949ED34
       53248       32768       4  Label: "KERN-B"
                                   Type: ChromeOS kernel
                                   UUID: 359C0A60-A786-364A-9EAC-523185AA12EC
                                   Attr: priority=2 tries=0 successful=1
    22355968     4194304       5  Label: "ROOT-B"
                                   Type: ChromeOS rootfs
                                   UUID: FBB77538-1F45-6E46-B9DE-135732A2BE7E
       16448           1       6  Label: "KERN-C"
                                   Type: ChromeOS kernel
                                   UUID: A511C323-31CC-D440-8B37-22FC7C1C5A38
                                   Attr: priority=0 tries=15 successful=0
       16449           1       7  Label: "ROOT-C"
                                   Type: ChromeOS rootfs
                                   UUID: A2BF3F99-D166-1247-8C6A-BEFD1F8702B0
       86016       32768       8  Label: "OEM"
                                   Type: Linux data
                                   UUID: 961B6939-26E7-884A-AA39-1844996DCBC2
       16450           1       9  Label: "reserved"
                                   Type: ChromeOS reserved
                                   UUID: F7DA8CBC-F93E-7942-BA15-A9A0B8C27CA6
       16451           1      10  Label: "reserved"
```

```
                                Type: ChromeOS reserved
                                UUID: 32A033CE-5A0B-3F44-8D87-2A87BAEDE64E
         64       16384      11  Label: "RWFW"
                                Type: ChromeOS firmware
                                UUID: 4119D401-D509-054E-896D-3B1672FEE9F2
     249856       32768      12  Label: "EFI-SYSTEM"
                                Type: EFI System Partition
                                UUID: 2805DB23-3877-D14B-BBC7-2EF643847E5B
   30777311          32          Sec GPT table
   30777343           1          Sec GPT header
+ flashrom -V -p internal:bus=spi --wp-status
flashrom v0.9.4  : 571cb5a : Oct 08 2012 10:16:03 UTC on Linux 3.4.0 (armv7l), built
with libpci 3.1.9, GCC 4.6.x-google 20120301 (prerelease), little endian
Acquiring lock (timeout=30 sec)...
Lock acquired.
Initializing internal programmer
Initializing linux_spi programmer
```

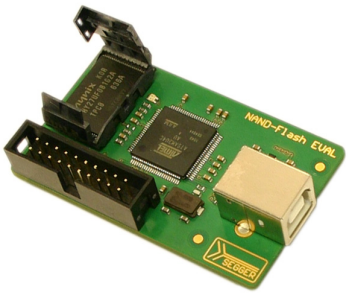
Code Block 9 - Excerpt from the very noisy Samsung Daisy boot process

Devices Analyzed:

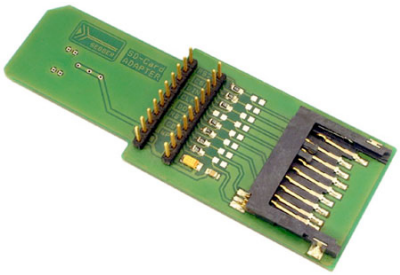
- Samsung Google Chromebook 11.6 (Code named Daisy and Snow depending on)
- Google CR-48 Chromebook Prototype (Code named Mario)

Building a NAND Hardware Test Harness

In addition to the introspection of shipped commercial devices, the MonkWorks researcher built a simple test harness to explore NAND devices at a hardware level without the overhead of a platform. The harness utilizes standard tools from the SEGGER company.



SEGGER NAND Flash Eval Board

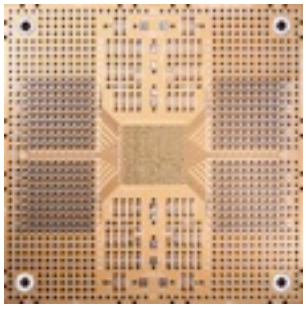


SEGGER SD Card Adapter

While most of the devices analyzed to date harness a Ball Grid Array (BGA) connection to the main processor board on the device, the majority of NAND manufacturers make a Thin Small Outline Package (TSOP) variant that can be loaded into the SEGGER evaluation board and accessed directly. This

provides the researcher direct access to the chip at the lowest levels, including any on-board / embedded controllers.

If needed, a more elaborate test harness has been envisioned but not yet built. This harness will utilize a SchmartBoard to directly probe the BGA chipsets for the various NAND producers. The only benefit of this approach is the ability to de-solder NAND chips from a Smartphone and analyze them directly.



SchmartBoard for BGA Packages

It should be noted that while obtaining TSOP variants of NAND Flash is trivial (Mouser.com and DigiKey.com have a large selection), obtaining BGA variants in small batches is difficult. Most manufacturers do not allow small purchases of these chips, and typically do not even share the datasheets to the public once the devices are selected for mass production and sold to device manufacturers.

Conclusions and Path Forward

Overall Conclusions

Attacking the MTD subsystem and partitions will be the most direct path to a successful POC for the NAND-Xplore tools. While it is believed that, with considerable time, a general attack and defend POC on all NAND through the MCC system could be produced, the researcher would suggest it be considered out of scope for this project. The added hurdles for manipulation of this MMC/eMMC system make the project harder and more abstract, but they do nothing to mitigate the attack surface overall. As such, the researcher considers a valid collection of POC tools for the MTD subsystem to fulfill the initial scope of the NAND-Xplore proposal.

Final Device Selection

Given the utilization of MTD partitions and openness of the platform, the Sony Xperia Arc S and the Sony Xperia Play have been selected for the initial revision of the NAND-Xplore tools. The HTC Droid Incredible will also be developed against as a secondary target for the POC toolchain.

```
$cat /proc/devices

Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 21 sg
 29 fb
 81 video4linux
 86 ch
 89 i2c
 90 mtd
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
235 msm_vpe
236 gemini
237 msm_camera
238 cypress_touchscreen
239 ttyGS
240 uio
241 ttyHS
242 ttyMSM
243 kgsl
244 msm_rotator
245 dia
246 msm_vidc_enc
247 msm_vidc_dec
248 msm_vidc_reg
249 adsp
250 msm_audio_lpa
251 oncrpc
252 smdpkt
253 smd
254 rtc

Block devices:
  1 ramdisk
259 blkext
  7 loop
  8 sd
 31 mtdblock
```

```
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
254 device-mapper
```

Code Block 10 - The Sony Arc S /proc/devices file

```
$cat /proc/partitions
major minor  #blocks  name

  31        0     409600 mtdblock0
  31        1       6144 mtdblock1
  31        2     103936 mtdblock2
  31        3     430080 mtdblock3
 179        0    7778304 mmcblk0
 179        1    7777280 mmcblk0p1
```

Code Block 11 - The Sony Xperia Arc S partitions file

```
$cat /proc/mtd
dev:    size   erasesize  name
mtd0: 19000000 00020000 "system"
mtd1: 00600000 00020000 "appslog"
mtd2: 06580000 00020000 "cache"
mtd3: 1a400000 00020000 "userdata"
```

Code Block 12 - The Sony Xperia Arc S MTD file

The researcher will also attempt to build a custom ROM/Kernel package for the Nexus 4 that bypasses the managed functionality of the NAND and utilizes the MTD sub-system for the POC NAND_Xplore tools. This task will be considered secondary to the main thrust of MTD POC.

NAND-Hide Path to Functional POC

The current plan for the *NAND-Hide* POC tool involves injecting a kernel module that handles hiding files or simply to use a one time python script to directly call and manipulate the MTD BBT (Bad Block Table) system. This is nice because the specification itself shows the software/RAM based BBT is actually

written back down to the hardware device so we have persistence even if the chip is desoldered and moved to an external extraction platform.

NAND-Find Path to Functional POC

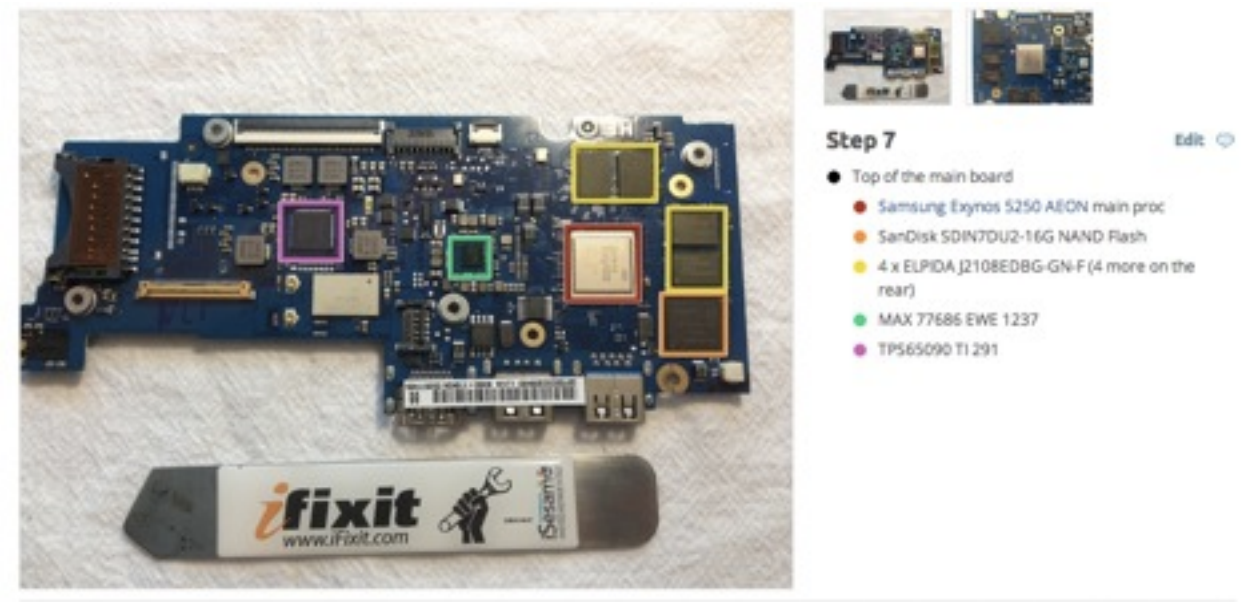
Using a collection of kprobes, we should be able to monitor the kernel / mtd sub-system and flag/alert any odd activity along this path. This can be done dynamically during runtime as a background process and it is expected the tool itself could be shared over the google play app store.

Addendum 1: The Samsung Google Chromebook Teardown

During the initial analysis of the device, the MonkWorks researcher realized that iFixit.com did not have a Teardown / Introspection page for 11.6" Samsung Chromebook (model number XE303C12-A01US). As part of the intent of this research is to inform the public, the researcher diverted an hour and created one.

The teardown is located here: http://www.ifixit.com/Teardown/Samsung+Chromebook+11.6+Teardown/12225/1

While not directly relevant to this project as a whole, the teardown did illuminate the NAND Flash hardware in the device:



iFixit.com teardown showing the SanDisk embedded NAND chip

Addendum 2: Notes on this Document

Most of the images in this document were downloaded from the internet without regards to copyright. MonkWorks does not own nor make claim to any of the images herein, unless they were obviously taken by the researcher in specific relation to this project.

This document is meant to be both directly informative about the overall NAND-Xplore project and a general place for notes on my methodology. As such, this document contains spurious and somewhat tangential information that was discovered during the research portion of the project.

Addendum 3: Planned Addendum for final version of this document

I intend to extend this document post delivery to include a rudimentary set of walkthroughs and tutorials on the Android and Chromium projects. As such, this document will eventually contain addendum tutorials for:

- Setting up a development environment to build Android and Chromium OS project
- Building the Android AOSP project for a specific device
- Building the Android open Source kernel for a specific device
- Building a custom ROM for a specific device (CyanogenMod)
- Building the Chromium open source project for a specific device

Links to Various Products Used or Referenced

A) Android Official Development Tools: http://developer.android.com/index.html

B) Android Open Source Project and Source: http://source.android.com/

C) Chromium Project and Source: http://www.chromium.org/

D) JEDEC - Global Standards for the Microelectronics Industry: http://www.jedec.org

E) LG Nexus 4 Teardown: http://www.ifixit.com/Teardown/Nexus+4+Teardown/11781/1

F) Monk's Android Tools on GitHub: https://github.com/monk-dot/ugly-pwnies

G) MTD - Memory Technology Devices Linux Sub System: http://www.linux-mtd.infradead.org/

H) ONFI - Open NAND Flash Interface group: http://www.onfi.org/

I) Samsung Google Chromebook iFixIt Teardown: http://www.ifixit.com/Teardown/Samsung+Chromebook+11.6+Teardown/12225/1

J) SchmartBoard for BGA Packages: http://www.schmartboard.com/index.asp?page=products_bga&id=110

K) SEGGER NAND Flash Eval Board and SD Card Adapter: http://www.segger.com/emfile_test_debug_hardware.html

L) Toshiba NAND for the LG Nexus 4: http://www.semicon.toshiba.co.jp/eng/product/memory/selection/nand/mlc/emmc/index.html

M) Toshiba - Sample Datasheet NAND images were borrowed from: http://www.toshiba.com/taec/components/Datasheet/TC58DVM92A5TA00.pdf

N) YAFFS Filesystem: http://www.yaffs.net/