Deeper than Ever Before

Exploring, Subverting, Breaking and Pivoting with NAND Flash Memory...

A Story Of Research By: Josh"m0nk" Thomas / @m0nk_dot

./whoami

- Partner and Chief Breaking Officer @ Atredis Partners (www.atredis.com)
  - Hardware reverse engineer and security researcher
  - Mobile / smartphone / embedded systems geek
  - Mesh networking enthusiast
  - ex software developer with a history of AI, crypto and rootkits
  - Enjoys the fuzzy boundaries between kernel drivers and hardware
  - Meme aficionado
- josh@atredis.com / @m0nk_dot on twitter

whois atredis.com

- Focused and targeted security firm
- Specializing in advanced hardware and software assessments
  - Mobile and embedded systems
  - Societal infrastructure
  - Black boxes
  - Advanced malware and rootkit analysis
- Handcrafted artisanal and deep bespoke research

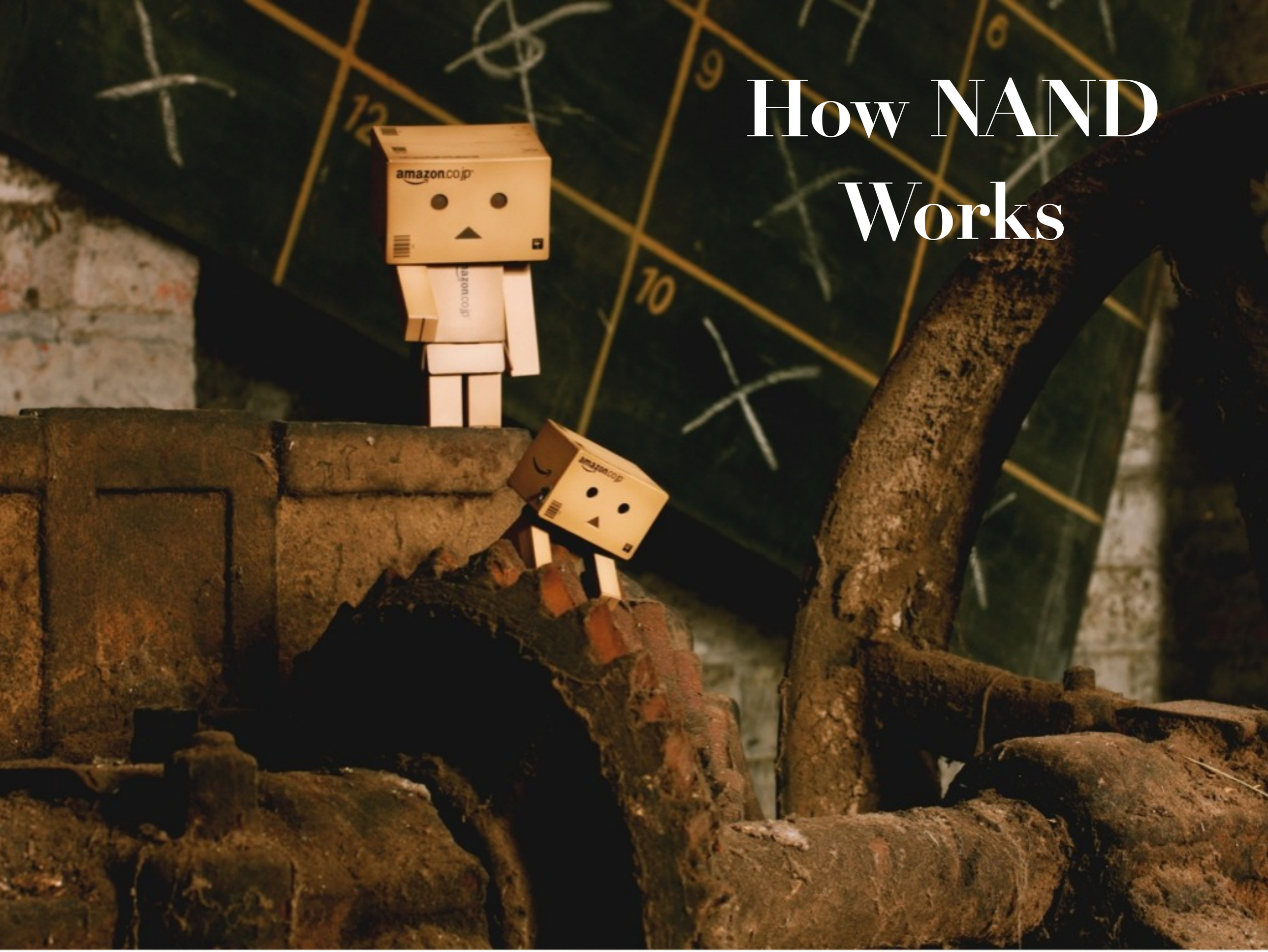
story arc

- Introduction and defensive postures of NAND flash
- How NAND works from a hardware and software perspective
- Project NandX: hiding and destroying
  - Selecting an attack surface
  - Exploring the project and the source code
  - Defensive postures and forensics in NAND flash: revisited
  - Moving away from hiding
- Project Burner: Full control of the platform
- Thank you / Q&A

NAND & Defense

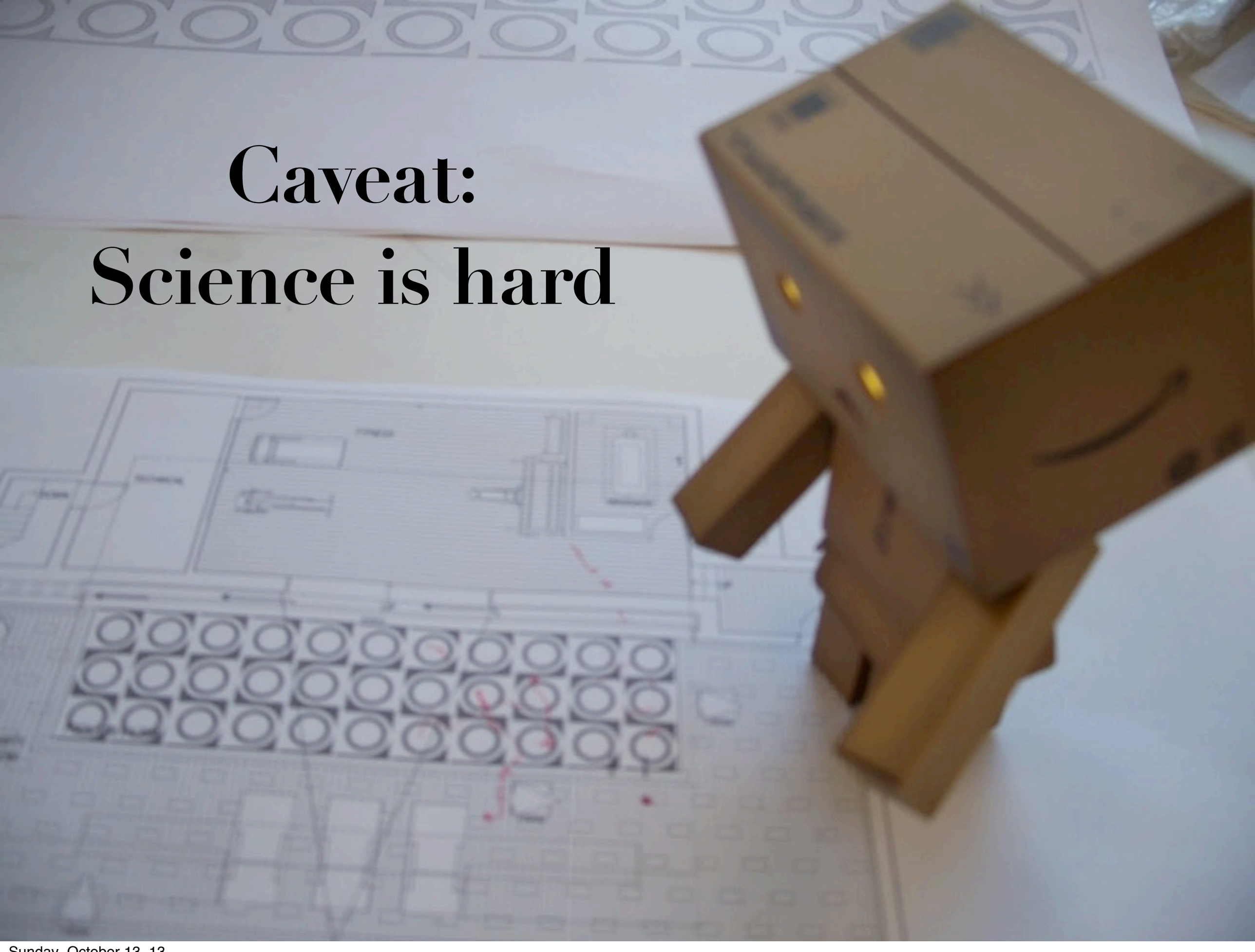
NAND Flash defensive postures

- What we talk about today is a functional aspect of the NAND flash hardware itself
- This is NOT a bug or flaw, it is simply HOW the hardware functions
- It is the reuse of an elegant controlled failure mechanism
- The only protections for this hardware attack surface are re-engineering and advanced forensic tools
- We need better tools



How NAND Works

Caveat:
Science is hard



How does NAND work: Hardware Overview

- Buckets - Might not be the technical term
- Pages - Typically 512, 2048 or 4096 bytes
- Blocks - Typically 16kb - 512kb
- Initially set to 1 (0xFF)
- Shifting to 0 is easy
- Shifting to 1 is hard

How does NAND work:

It's a trap

- It is easy to store electrons
- It is highly difficult to catch single electrons in the wild

How does NAND work:
Crafting Data

- To write data to NAND we simply start removing stored electrons
- In essence, we carve our data out of the 0xFF blob
- When we save a file, we determine if the diff can be carved... or if a new version must be generated
- This makes forensics hard

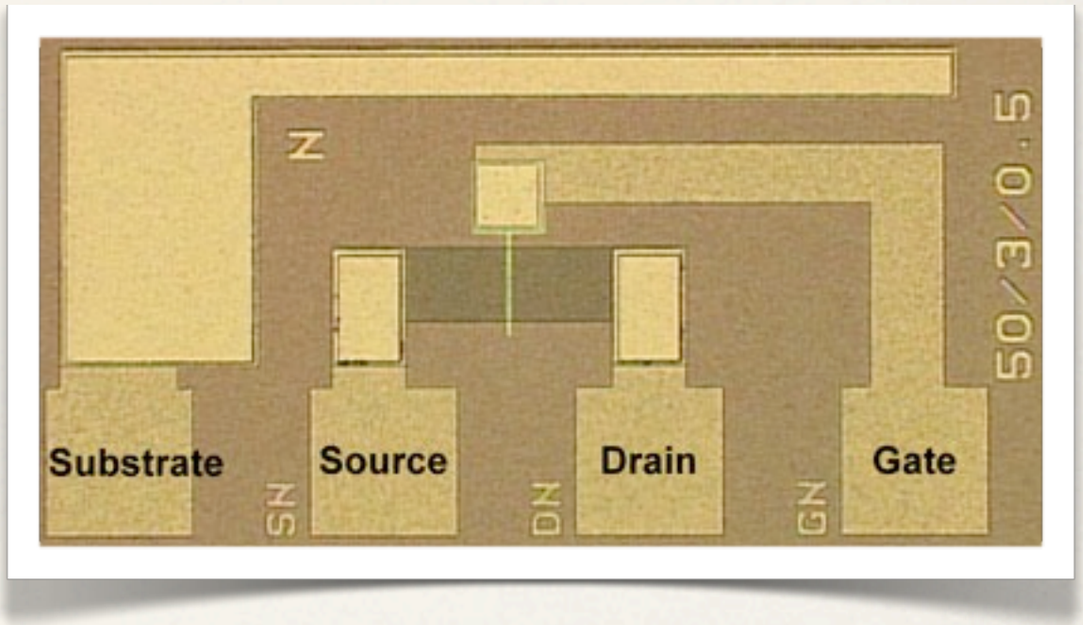
How does NAND work: Elegant & Controlled Failure

- Gates are hard to build and somewhat fragile...

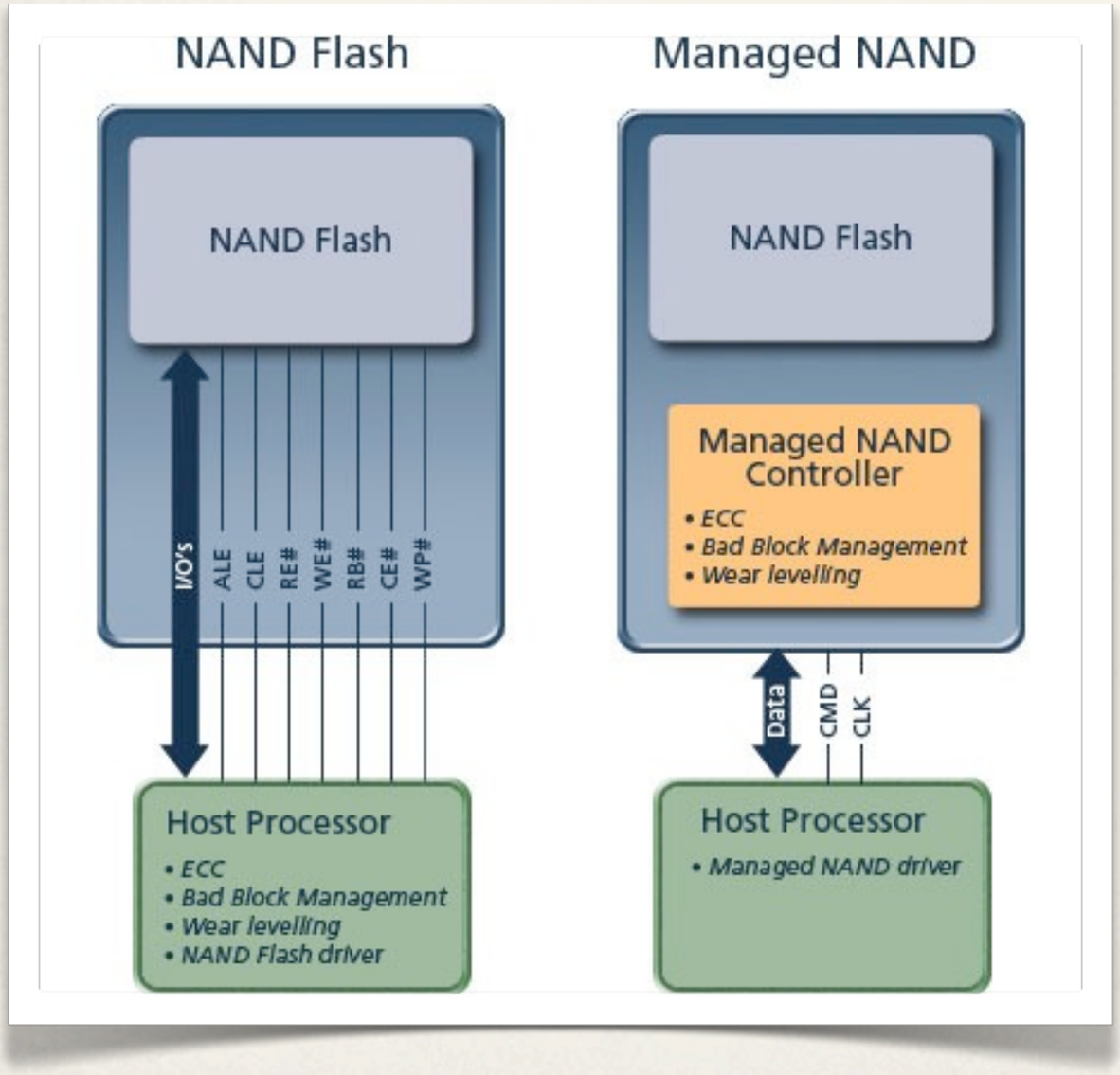
- Things break normally after ~10 - 100k writes

- Because they wear out, we do wear leveling to disperse the headache across the full surface

- Wear leveling leaves residue and makes forensics hard

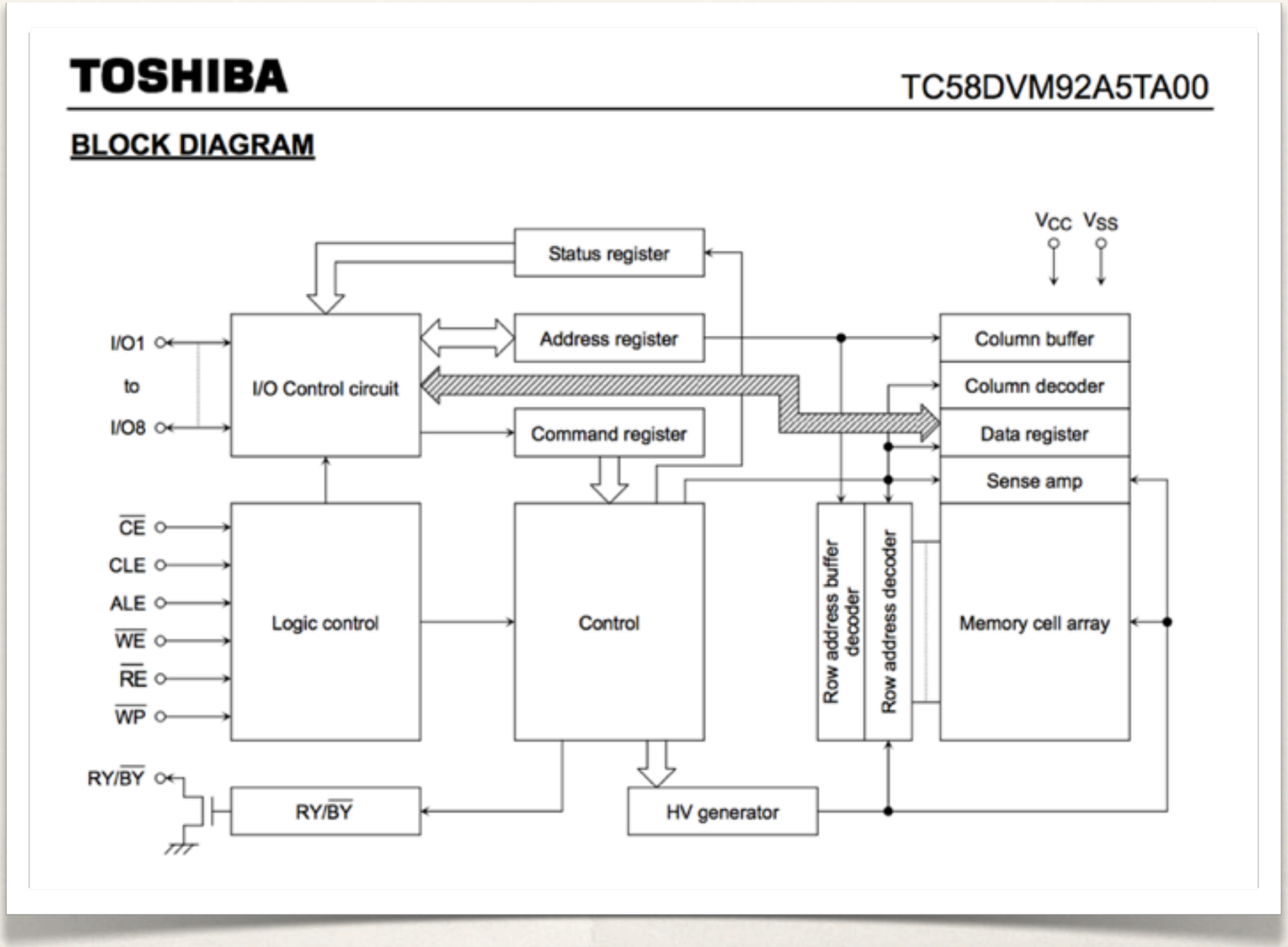


How does NAND work: Types of NAND flash



- Raw NAND flash
  - All controls are software controls
    - Hardware is simply raw, dumb storage
- Managed NAND flash
  - Controls are embedded in hardware
  - Embedded turing controllers

How does NAND work: Managed NAND flash

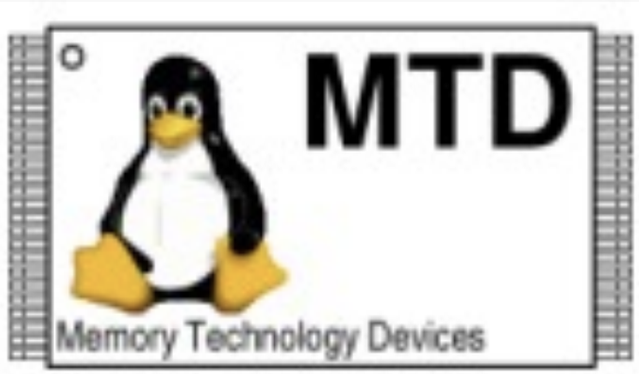


How does NAND work: Software & the Android Kernel

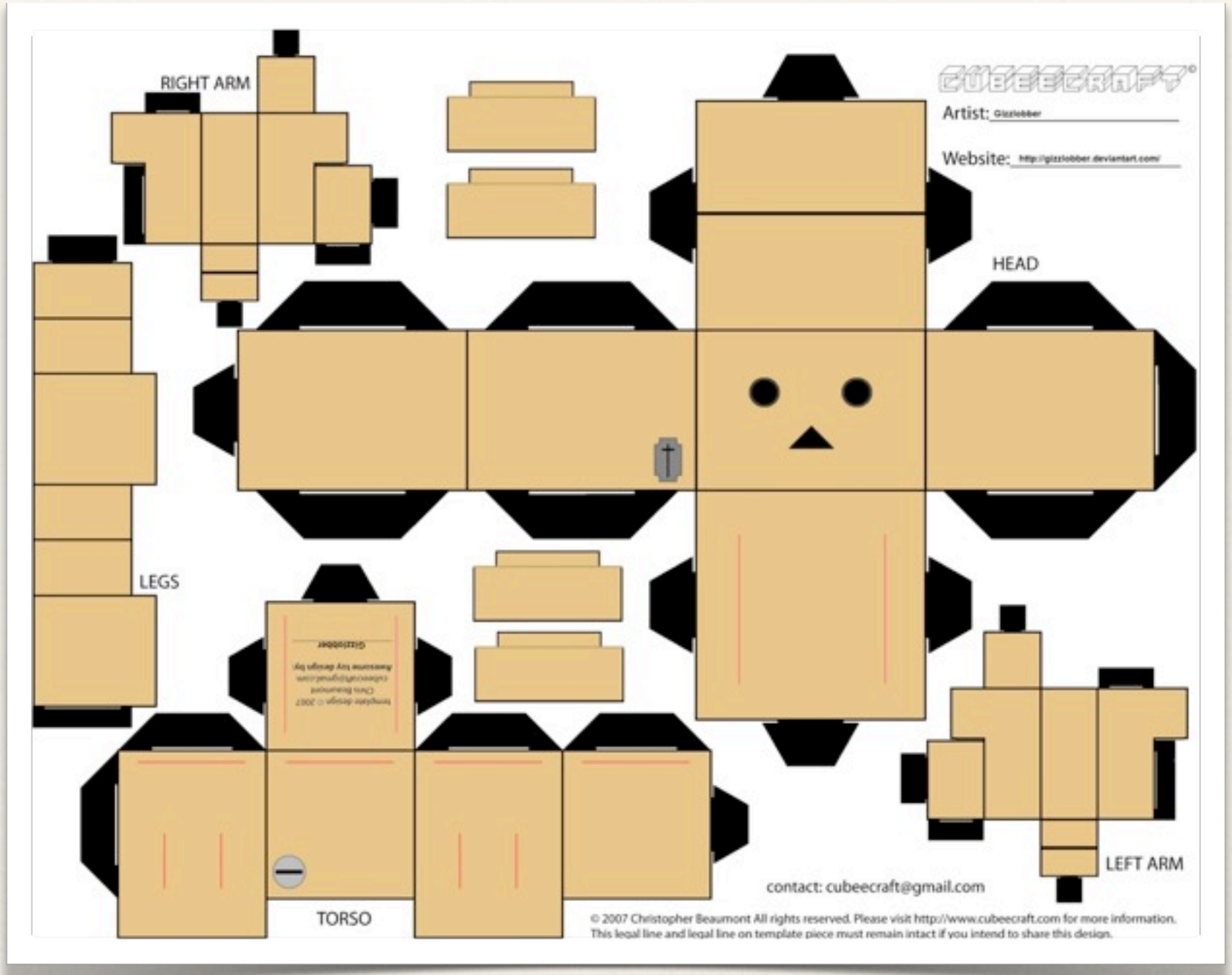
- RAW NAND = Complex Driver
- MMC/eMMC = Simple Driver
- Proprietary (closed) wear leveling algorithms are normally embedded
- Regardless of the driver, the hardware must interact with the system and expose failure mechanisms

How does NAND work: The MTD Meta Driver

- MSM / MTD Subsystem
- Kind of a meta-driver
- Used heavily for boot partitions on Android
- In certain phones it is used to manage all NAND



How does NAND work: Building the system



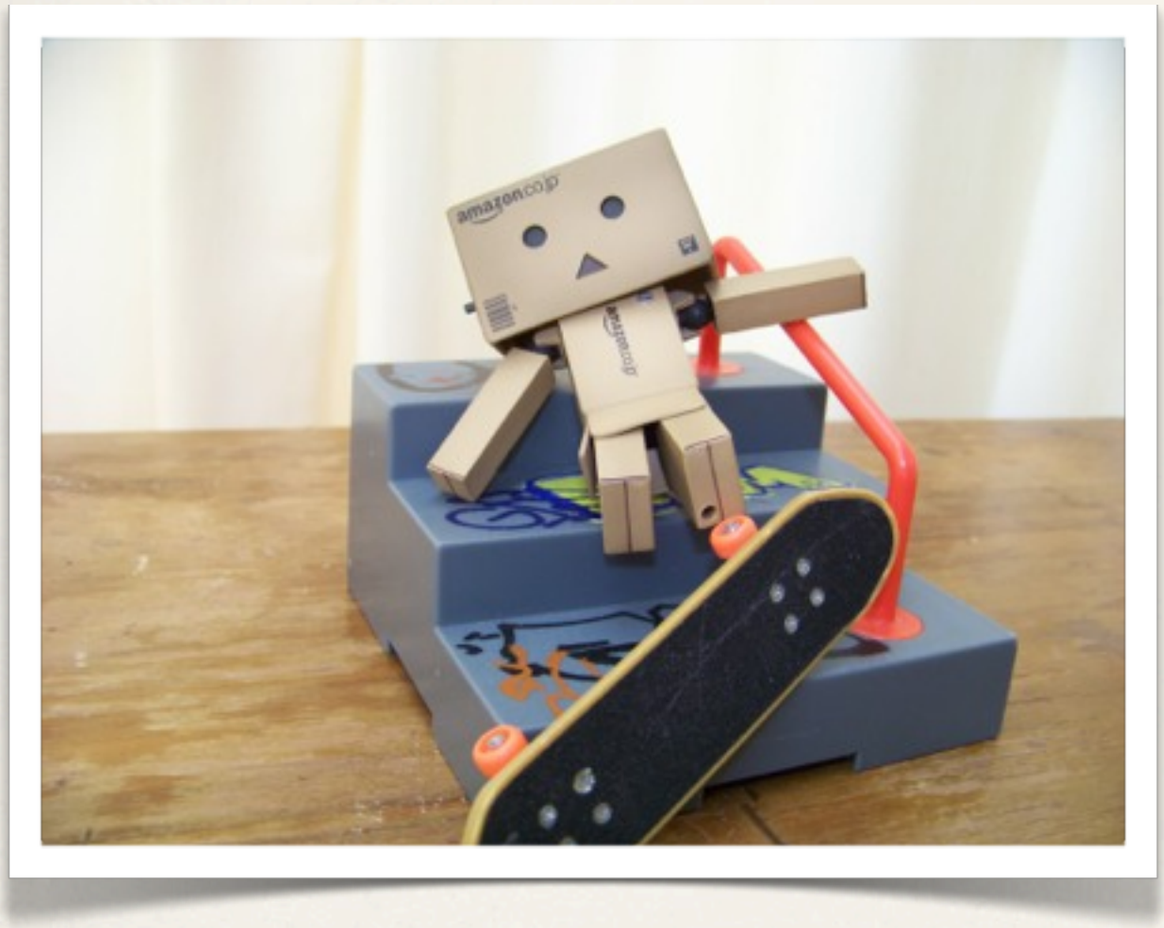
How does NAND work: Revisiting controlled failure

- Given the HW fails, NAND typically does a lazy erase with a single flag
- The controller has a threshold for corrupted read / writes
- There are ECC / Forward ECC implementations to help with failure

How does NAND work: Failure is inevitable

- When failure occurs, the NAND block is marked bad in the OOB / BBT
- Block disappears from the addressable system
- No current tools can read it
- No current tools can mark the block good again

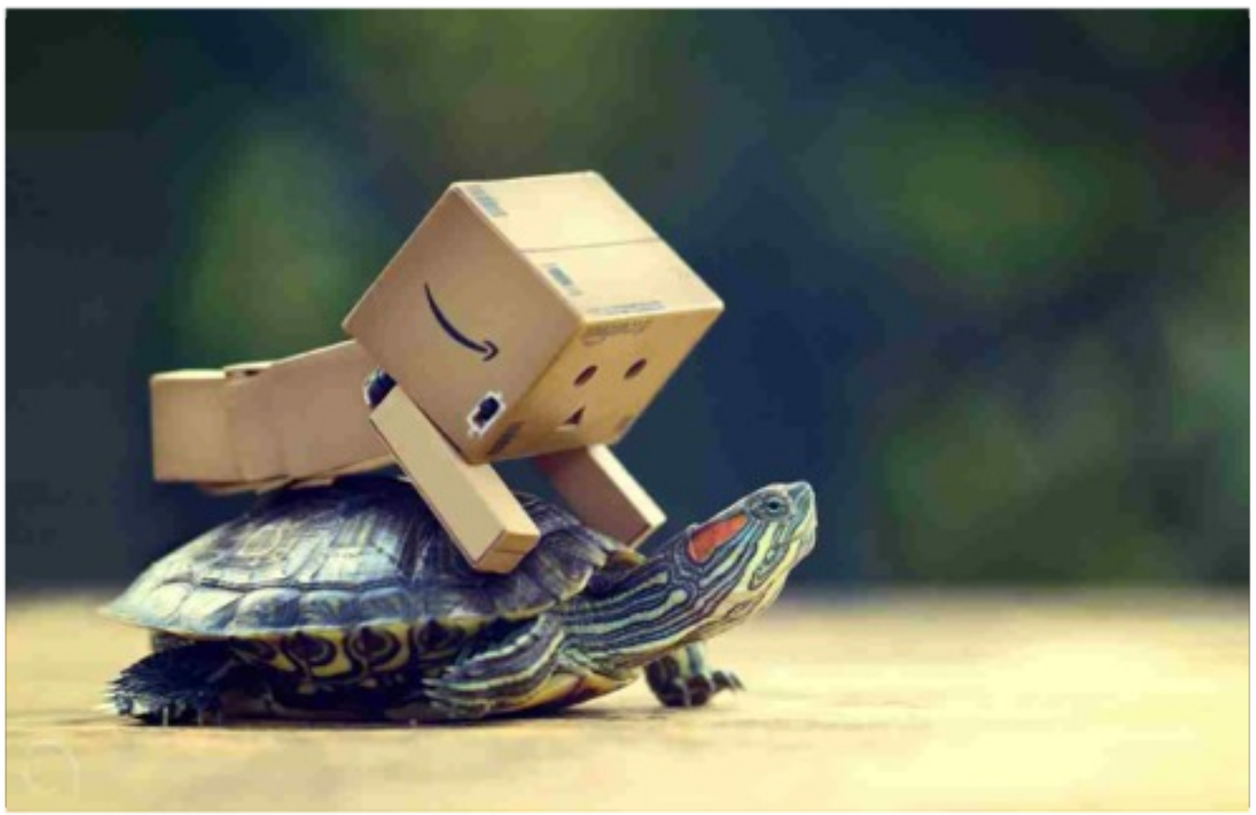
How does NAND work: When things go wrong



- Some systems fully manage the BBT in kernel memory (this is written back to disk as the “master” during reboot)
- Some systems use dual-page OOB markers for BBT & ECC (Sony!)
- Some systems use 1st or last block for the entire BBT & ECC (think of it as address -10)

How does NAND work: Attack surface exposure

- YAFFS and other File Systems
- MTD at the Driver Level
- Android / Linux Kernel
- Flash transition layers
- Reverse the embedded controller hardware
- Poor user-land coding



Project NandX: Hiding

Project NandX

- Goals:
  - Arbitrarily mark a functional block of NAND flash as bad (thus removing it from the addressable system)
  - Arbitrarily read & write from my controlled bad block
  - Ensure no other tools can read / write, including the Android kernel and the DD tool
- Expectations:
  - A lot of hardware reverse engineering and embedded controller firmware assessment

Project NandX: Reality of the first stage

- ~ 30 phones purchased / tested / returned

- Looking for raw NAND
- Open Source kernel

Project NandX:
Exploring all the phones


amazon.co.jp
ebay tm
Sorry
I'm not
perfect

Project NandX: The Sony Xperia Arc S!

```
$ 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
```

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

Project NandX: The MTD subsystem

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

```
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
```

Project NandX:
Modifying the API

- Almost everything I do is simply calling the API in the wrong order
  - The 1 exception is the OOB write of the BBT
- Steps:
  - Pick a block and wipe it
  - Cover the entire block in 0xDEADBEEF
  - Mark the block as bad (0x00 out the OOB in the case of Sony)
  - Read and write back arbitrarily
- Watch the reboot from collision!

Project NandX:

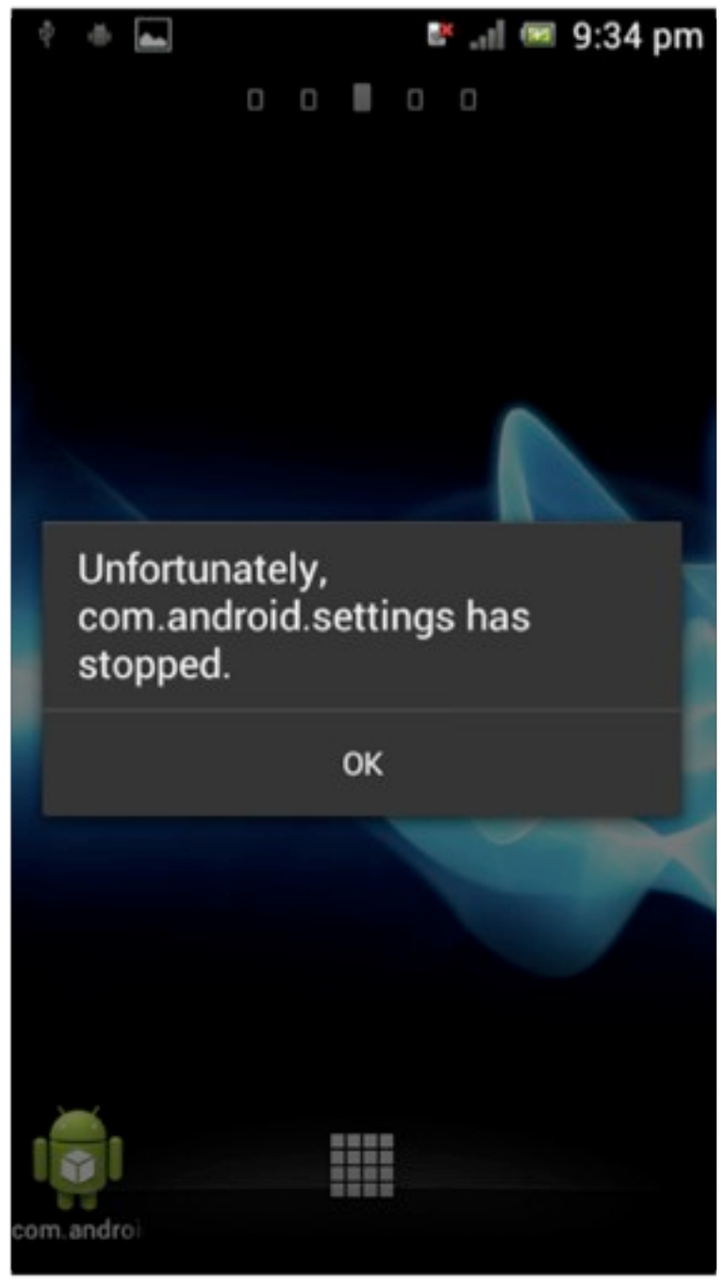
Demo time!!!

http://youtu.be/AE_oUkKKaBY

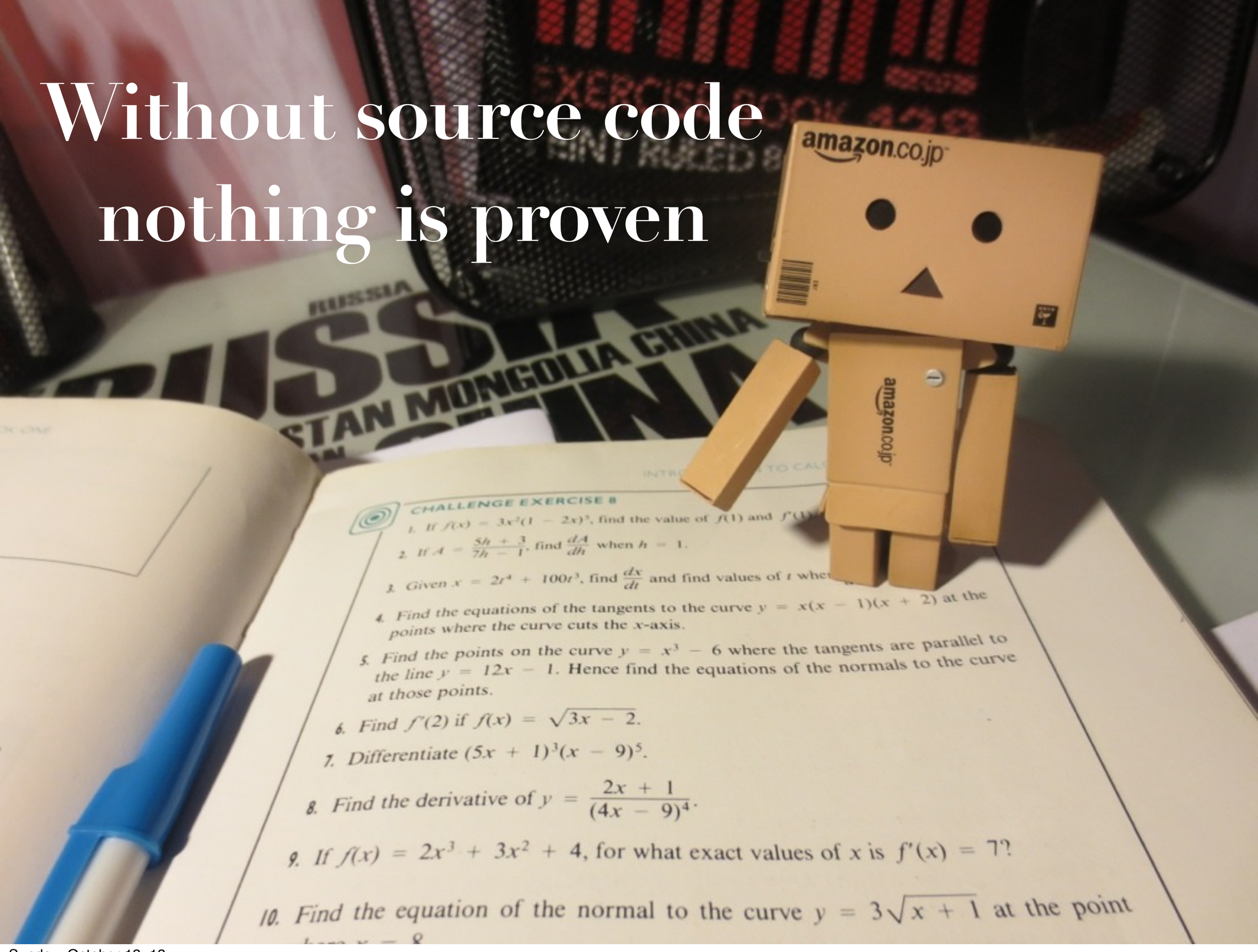
Project NandX:

A disappearing act

- Block 37 is gone
- Covered in 0xDEADBEEF
- It originally held Settings.app (com.android.settings) from Android
- Reboot from double free on hardware



Without source code nothing is proven



Project NandX:

Main source

```
7530 ▶  /*... */
7539    static void nandx_file_injector(int blockLocation, void *bufferToWrite)
7540 ▼  {
7541 ▶    /*... */
7554
7555      //TODO: Grab and check return values here!!!!
7556
7557 ▶    /*... */
7564
7565      int err = 0;
7566
7567      //Moves all data out of the target block (no, it really doesn't)
7568      nandx_move_data_from_block( blockLocation );
7569
7570      //Erases the targeted block
7571      nandx_erase_block( blockLocation );
7572
7573      //Injects our buffer directly into the block
7574      nandx_buffer_write_to_block( blockLocation, bufferToWrite );
7575
7576      //Marks the target block as bad
7577      err = nandx_mark_bad_framework( blockLocation );
7578 ▼    if( !err ){
7579        printk(PRINT_PREF "First attempt at marking %d bad failed, going manual\n",
...  blockLocation);
7580        err = nandx_mark_bad_manual( blockLocation );
7581 ⌙    }
7582
7583 ⌙  }
```

Project NandX: 1st stage marking bad blocks

```
7138 ▶  /*⋯ */
7147    static int nandx_mark_bad_framework(int blockLocation)
7148 ▼  {
7149 ▶    /*⋯ */
7168      int ret;
7169      loff_t addr = blockLocation * mtd->erasesize;
7170
7171      printk(PRINT_PREF "Marking the block %d as BAD\n", blockLocation);
7172
7173      ret = mtd->block_markbad(mtd, addr);
7174      if (ret)
7175        printk(PRINT_PREF "Success - block %d has been marked bad\n", blockLocation);
7176      else
7177        printk(PRINT_PREF "Failure - Why U no mark block %d as bad?\n", blockLocation);
7178
7179      return ret;
7180
7181 ⌙  }
```

Project NandX:
2nd stage marking bad blocks

```
7183 ▶  /*... */
7193    static int nandx_mark_bad_manual(int blockLocation)
7194 ▼  {
7195 ▶    /*... */
7219
7220      int ret;
7221      loff_t ofs = blockLocation * mtd->erasesize;
7222
7223      // THIS CALL IS THE ENTIRE MAGIC OF NANDX-HIDE
7224      ret = msm_nand_block_markbad(mtd, ofs);
7225
7226      if(ret)
7227        printk(PRINT_PREF "We call into the driver and make %d go away.\n", blockLocation);
7228      else
7229        printk(PRINT_PREF "Odd.. even a RAW write on the OOB doesn't kill block: %d\n",
   ...  blockLocation);
7230      return ret;
7231 ▶  }
```

Project NandX:

Overview logs

```
<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] ================================================
```

Project NandX:

Detailed logs

```
<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:
0000000000000000000000000000000000000000000000000000000000000000
<2>[  109.246276] 00020:
0000000000000000000000000000000000000000000000000000000000000000
...
...
<2>[  109.247344]
0000000000000000
<2>[  109.247375]
0000000000000000
<2>[  109.247406]
deadbeefdeadbeefd
```

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

Project NandX: Final hiding thoughts

- Once a block is marked bad, no tool currently on the market can reclaim it
- Factory reset does not reclaim the space
- Flashing a new ROM does not reclaim the space
- DD does not see the block to copy it
- 0xDEADBEEF is hidden and fully persistent

Project NandX: Turning a tool into a weapon

Project NandX:
Why stop at hiding?

- Thought 1:
  - Kill data you desire in place, wait for IT to wipe and trash the drive, physical exfiltrate the data
- Thought 2:
  - Marking the block bad means it disappears from the system permanently...

Project NandX:

The final weapon

- Remotely side-load the kernel module to remove 1 block at a time until the physical NAND flash chip has no valid blocks left
- Can not be fixed, must be replaced
- Almost all embedded devices use NAND, not just phones
- SCADA

Project NandX: Continued reading

- When given the opportunity, I try to open source all research
- Not always possible given vendor trade secrets & intellectual property
- Respecting others, this project was built 100% on open source software so I could release the research
- Everything is here (including a very long white paper):
  - https://github.com/monk-dot/NandX

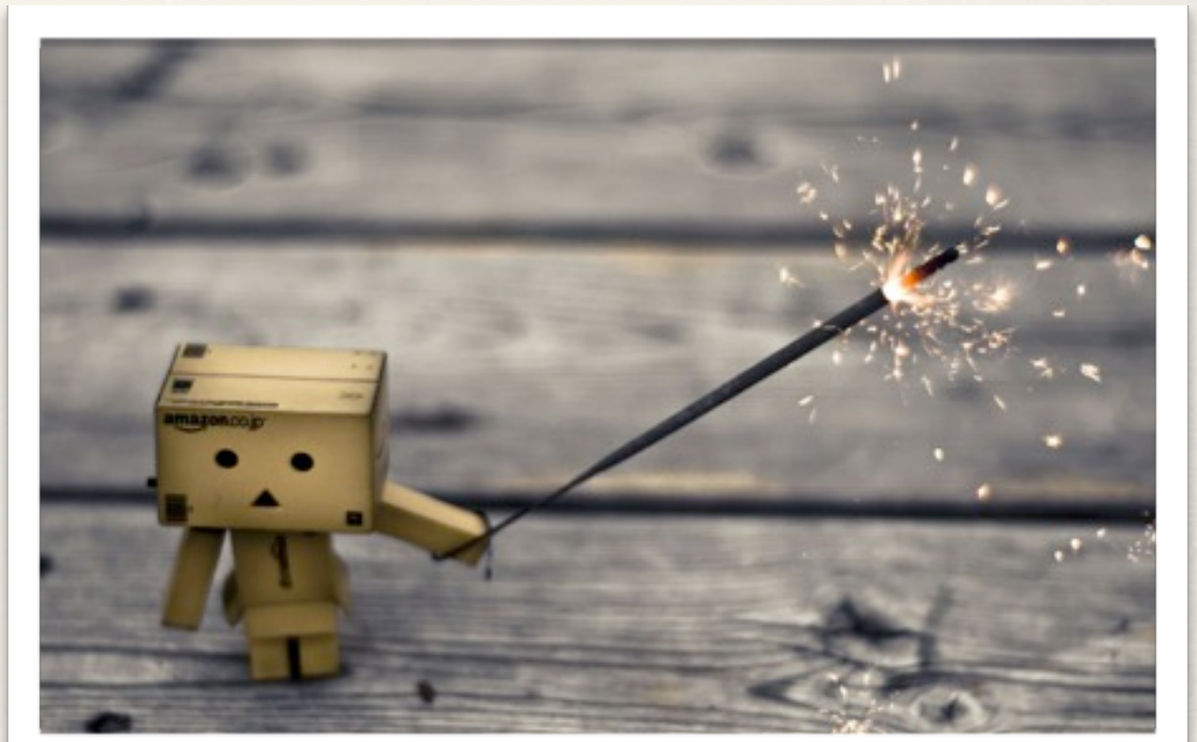
Project Burner: Beyond Hiding

Project Burner:
High level overview

- Hypothesis:
  - Given full kernel control of an Android phone, I can control the power and voltage from the internal battery to physically manipulate internal hardware. I can control the process and target individual components to the point of full kinetic destruction using unexpected voltages.
- End result:
  - Hypothesis is correct.

Project Burner: Android Power Hardware

- The Battery stores raw power to distribute
- The USB stack also pushes power into the system
- The PMIC hardware is utilized to distribute voltage across the traces
- Kernel manages the PMIC settings directly
- Not all traces are protected by capacitors ad resistors



Project Burner: Android Power Regulation

- Well documented in:
  - <kernel_source>/Documentation/power/regulator/overview.txt
- Aside from drivers, less than 10 *.c and *.h files control the voltage flow throughout the PCB
- Of particular interest is the *_regulator.c file for whatever PCB you are exploring

Project Burner:

The NAND target

- The target platform for Project Burner was the Sony Xperia Z (yuga)

- Based on the Qualcomm Snapdragon reference platform

- The NAND controller is tied to the SDCard controller

- Qualcomm MSM 7X00A SDCC controls both traces on the PCB

Project Burner:
Upping NAND Voltage

project kernel/sony/apq8064/

diff --git a/arch/arm/mach-msm/board-sony_yuga-regulator.c
b/arch/arm/mach-msm/board-sony_yuga-regulator.c

-- RPM_LDO(L5, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L5, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),

-- RPM_LDO(L6, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L6, 0, 1, 0, 5900000, 5900000, NULL, 0, 0),

Project Burner:
Upping NAND Voltage

- With higher voltages:
  - Every NAND read corrupts the data in transit
  - Every NAND write corrupts the NAND hardware
- PMIC values are stored so rebooting the device essentially corrupts all of NAND as the bootloader tries to load the kernel

Project Burner:
Dropping NAND Voltage

project kernel/sony/apq8064/

diff --git a/arch/arm/mach-msm/board-sony_yuga-regulator.c
b/arch/arm/mach-msm/board-sony_yuga-regulator.c

-- RPM_LDO(L5, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L5, 0, 1, 0, 1250000, 1250000, NULL, 0, 0),

-- RPM_LDO(L6, 0, 1, 0, 2950000, 2950000, NULL, 0, 0),
++ RPM_LDO(L6, 0, 1, 0, 1250000, 1250000, NULL, 0, 0),

Project Burner: Dropping NAND Voltage

- With lower voltages:
  - Most NAND reads corrupt the data in transit
  - Every NAND write attempt fails at the hardware level
- This technique essentially generates a phone frozen in time, no writes can occur and the device fails to successfully boot

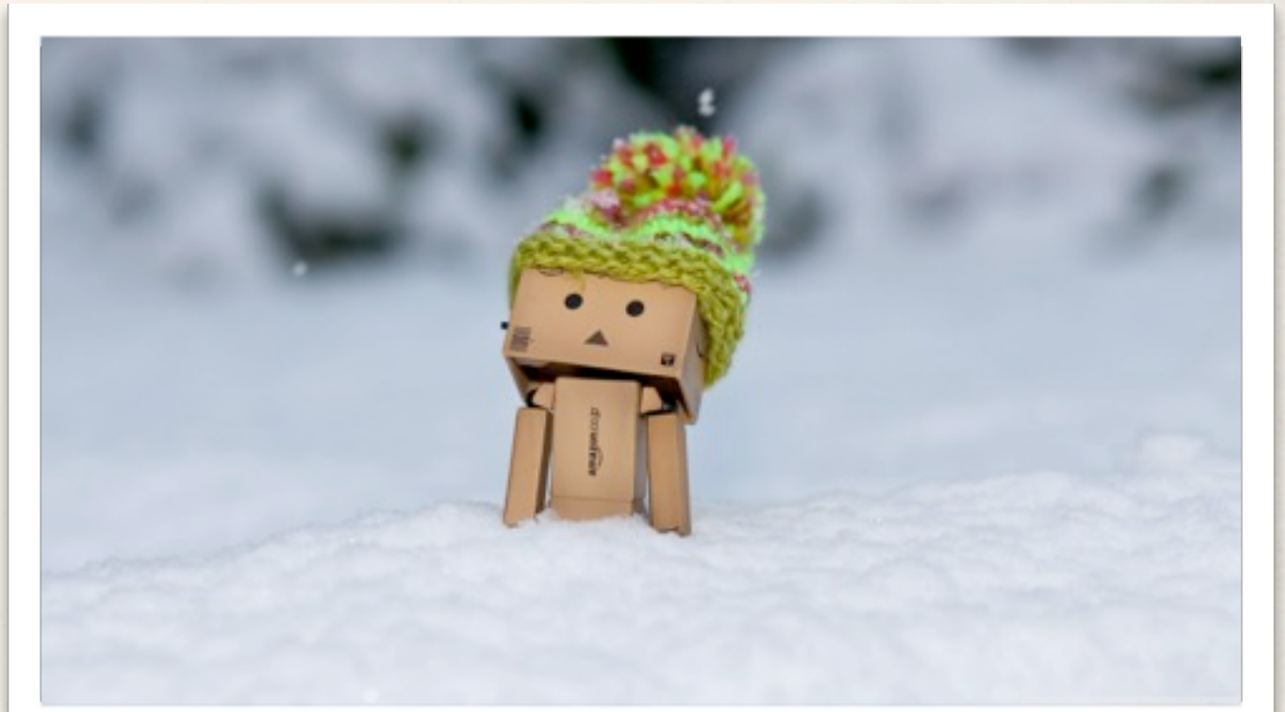
Project Burner: Thermal regulation

Project Burner:
Thermal regulation

- When playing with voltages out of expected ranges, one needs to pay attention to the thermal checking daemon: thermald

- This daemon should not be shut down, it simply needs to be nullified

- A topic for another talk... soon



Final Thoughts

An aside about malware and rootkits

- We tend to assume all malware is boring and driven by capitalist intents

- This mentality changes drastically when we shift our focus to highly advanced and expensive tools

- I urge you to explore and extend advanced research in the field, there are not enough of us pushing the boundaries in an open manner

- I open source in an attempt to nullify the use of my research as a weapon

Questions?


amazon.co.jp
amazon.co.jp
Thank You - @m0nk_dot