I.MX MEMORY MADNESS

HOW TO DUMP, PARSE, AND ANALYZE I.MX FLASH MEMORY CHIPS

Damien Cauquil | HITB Amsterdam 2019 (🎂🎉)
WHO AM I?

🔍 Head of R&D @ Econocom Digital.Security

🌟 Senior security researcher

💻 Hardware hacker (or at least pretending)
AGENDA

- Firmware extraction 101
- Meet the i.MX architecture
- i.MX flash memory layout
- imx-nand-tools FTW
- Best practices
FIRMWARE EXTRACTION 101
WHY DO WE WANT TO EXTRACT A DEVICE'S FIRMWARE?

- Contains filesystems, applications, binary files
- May also contain VERY interesting data:
  encryption/decryption keys, certificates, passwords
WHY DO WE WANT TO EXTRACT A DEVICE'S FIRMWARE?

We need to understand everything about a device:

- How it has been designed
- How it (really) works
- Where and how every bit of data is stored
METHOD #1: CLIPPING & READING
METHOD #2: CHIP-OFF
PROFESSIONAL FLASH PROGRAMMER
NAND DUMP SIZE

$ ls -alh camera.bin
-rwx------ 1 virtualabs virtualabs 1,1G camera.bin

Dump file is greater than 1 GB!
PAGES, BYTES AND OOB

- Bytes are stored, erased, and modified in pages.
- NAND flash chips are not 100% reliable and errors when storing bits may occur.
- To avoid this, vendors usually provide more space to store Error Correction Codes (ECC) in spare-byte area (OOB).
PAGES, BYTES AND OOB

- OOB
- OOB
- OOB
- OOB
- OOB

- 5x OOB
# Pages, Bytes and OOB

## MT29F16G08ADB and MT29F16G16ADB

### Features

- **Open NAND Flash Interface (ONFI) 1.0-compliant**
- **Single-level cell (SLC) technology**
- **Organization**
  - Page size x8: 4320 bytes (4096 + 224 bytes)
  - Page size x16: 2160 words (2048 + 112 words)
  - Block size: 64 pages (256K + 14K bytes)
  - Plane size: 2 planes x 2048 blocks per plane
  - Device size: 8Gb: 4096 blocks
  - Device size: 16Gb: 8192 blocks
- **Asynchronous I/O performance**
  - t_RC/t_WC: 20ns (3.3V), 30ns (1.8V)

---

*Digital Security*
import sys

PAGE, OOB = 4096, 224
BLOCK = PAGE + OOB
orig_dump = open(sys.argv[1], 'rb').read()
out_dump = open(sys.argv[2], 'wb')
nblocks = int(len(orig_dump) / BLOCK)
for i in range(nblocks):
    out_dump.write(orig_dump[i*BLOCK:(i+1)*PAGE + OOB])
out_dump.close()
orig_dump.close()
$ binwalk ipcam.fw.bin

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
</table>
| 96188     | 0x177BC     | CRC32 polynomial table, [...]
| [... ]    |             |                                                       |
| 2490368   | 0x260000    | Squashfs filesystem, [...]
| 4456448   | 0x440000    | Squashfs filesystem, [...]
| 5505024   | 0x540000    | Squashfs filesystem, [...]
| 6684672   | 0x660000    | Squashfs filesystem, [...]
| 7208960   | 0x6E0000    | JFFS2 filesystem, little endian                       |
| 7643512   | 0x74A178    | JFFS2 filesystem, little endian                       |
EXTRACTING FILES FROM VARIOUS FILESYSTEMS

- **SquashFS**: compressed filesystem, one partition/image
- **YAFFS2**: Yet Another Flash FS
- **JFFS2**: Journalized Flash FS version 2, one partition/image
- **UBI**: Unsorted Block Image, multiple partitions with various FS
IT'S A DOCUMENTED PROCESS

PenTestPartners just published a blog entry:

AND WE STUMBLED UPON AN I.MX6 SYSTEM
HEX ANALYSIS REVEALED WEIRD BYTES

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D81:E2E0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E2F0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E300</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E310</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E320</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E330</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E340</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E350</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E360</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E370</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E380</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E390</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E3A0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E3B0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E3C0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E3D0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>3D81:E3E0</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>
```

Digital Security
A CRAPPY BYTE BEFORE UBI SIGNATURE
SAME 1-BYTE OFFSET IN BINWALK OUTPUT

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24891865</td>
<td>0x17BD1D9</td>
<td>YAFFS filesystem</td>
</tr>
<tr>
<td>25159701</td>
<td>0x17FE815</td>
<td>YAFFS filesystem</td>
</tr>
<tr>
<td>25436181</td>
<td>0x1842015</td>
<td>YAFFS filesystem</td>
</tr>
<tr>
<td>25712661</td>
<td>0x1885815</td>
<td>YAFFS filesystem</td>
</tr>
<tr>
<td>25727234</td>
<td>0x1889102</td>
<td>Unix path: /usr/share/brw/local/index.html</td>
</tr>
<tr>
<td>35389441</td>
<td>0x21C0001</td>
<td>UBI erase count header, version: 1, EC: 0x4, VID header offset: 0x1000</td>
</tr>
</tbody>
</table>

UBI header is not aligned on page size (0x1000)
THAT'S WEIRD 😞

- Quick investigation revealed anomalies

- Our dump seems OK, but we still cannot extract data from it

- It must be related to i.MX: maybe a custom storage mechanism
I.MX ARCHITECTURE AND MEMORY LAYOUT
I.MX ARCHITECTURE

- Integrated Multimedia Application processors
- Popular in automotive and home automation industries
- Provides a lot of features including:
  - Secure/non-secure RAM
  - SATA II support
  - Secure Boot ...
I.MX ARCHITECTURE
I.MX ARCHITECTURE

- Can boot on various storage devices:
  - NAND Flash
  - Parallel NOR Flash
  - SD card
  - MMC
  - SATA HDD

- It also embeds a boot ROM (Freescale Inc.)
GENERAL-PURPOSE MULTIMEDIA INTERFACE

- controls how data is read/stored on NAND flash chips
- supports multiple NAND flash chips
- uses BCH to perform error control and correction
NAND FLASH STRUCTURE

2KB Main area

512 main  parity  512 main  parity  512 main  parity  512 main  parity

Bad block information at column address 2048

64B spare

Metadata

Swap byte

Bad block information at 4th block's data area

(image extracted from i.MX28 reference manual)
HOW IS DATA STORED?

- Data is split in 512-byte chunks
- ECC bits are added at the end of each chunk
- Chunks are then grouped and stored in a page preceeded by one metadata block
- Bad block marker byte is swapped with first metadata byte!
WEIRD BYTE EXPLAINED!
FIRMWARE CONFIGURATION BLOCK (FCB)

- This structure contains all the required information about how data is stored.
- It must be present in the first 1MB.
- Second field of this structure contains "FCB " in ASCII.
FCB SIGNATURE IN HEXDUMP

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0000</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:0010</td>
<td>46 43 42 20</td>
</tr>
<tr>
<td>0000:0020</td>
<td>00 10 00 00</td>
</tr>
<tr>
<td>0000:0030</td>
<td>00 10 00 00</td>
</tr>
<tr>
<td>0000:0040</td>
<td>00 02 00 00</td>
</tr>
<tr>
<td>0000:0050</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:0060</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:0070</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:0080</td>
<td>36 00 00 00</td>
</tr>
<tr>
<td>0000:0090</td>
<td>00 10 00 00</td>
</tr>
<tr>
<td>0000:00A0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:00B0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:00C0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:00D0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:00E0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:00F0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0000:0100</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>
FIRMWARE CONFIGURATION BLOCK (FCB)

- NAND page data size
- Block N ECC type
- Block N size
- Block 0 ECC type
- Block 0 size
- Number of bytes in metadata of a page
- ...
FCB SIGNATURE IN HEXDUMP

Offset +0x3C: number of bytes of metadata block
1-BYTE OFFSET EXPLAINED!

| 30C3:4790 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:47A0 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:47B0 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:47C0 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:47D0 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:47E0 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:47F0 | FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF |
| 30C3:4800 | 00 55 42 49 23 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| 30C3:4810 | 02 00 00 10 00 00 00 20 00 07 42 E4 66 00 00 00 |
| 30C3:4820 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| 30C3:4830 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 15 87 C5 |
| 30C3:4840 | 78 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| 30C3:4850 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
| 30C3:4860 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |

UBI# .......................... Bäf .......................... Ä
x ..........................
DISCOVERED BAD BLOCK TABLE (DBBT)

- Provides custom NAND bad block management
- Its headers provide information about the number of bad blocks and impacted pages
(image extracted from i.MX28 reference manual)
ECC

- Provides a way to dynamically fix errors, if possible
- Uses **BCH** (Bose, Ray-Chaudhuri and Hocquenghem) error-correcting code
- Data bytes may be shifted by a number of bits due to BCH bits
SO, WHAT'S NEXT?
FROM NAND FLASH DUMP TO FILESYSTEMS
RECOVER AND REMAP ALL THE BYTES

- We first find an **FCB structure** and parse it to recover all the critical parameters.

- Then we **remove every metadata and ECC bits** according to this FCB.

- We use ECC bits to **fix errors and save each block in an output file**.
@ Page address

@ rectified page address
IMX NAND TOOLS

$ sudo pip install imx-nand-tools

FCB PARSING
CONVERTING IMAGE TO USEABLE DUMP

virtualabs@virtubox:~$  

digital.security
ANALYZING THIS NEW DUMP

virtualabs@virtubox:~$
UBI OVERVIEW

UBI
Logical erase blocks

Volume 1

LEB  LEB  LEB  LEB

Volume 2

LEB  LEB  LEB

MTD
Physical erase blocks

PEB  PEB  PEB  PEB  PEB  PEB  PEB  PEB  PEB  PEB

Free block

Free block
UBIREADER

- Provides a **set of tools** to parse, analyze and extract volumes and files from a UBI container
- **Open-source** and available on Github
- Written in **Python**
- Does not support **fastboot** mode
ACCESSING FILES STORED IN VARIOUS IMAGES

```
$ ubireader_extract_files -iw img-xx_vol-iio_0633_0.ubifs
[...]
$ ls ubifs-root/ -al
total 76
  drwxr-xr-x  19 virtualabs virtualabs 4096 mai    9 09:40 .
  drwxr-xr-x   3 virtualabs virtualabs 4096 mai    9 09:40..
  drwxr-xr-x   2 virtualabs virtualabs 4096 mai    9 09:40 bin
  drwxr-xr-x   2 virtualabs virtualabs 4096 mai    9 09:40 boot
  drwxr-xr-x   5 virtualabs virtualabs 4096 mai    9 09:40 Data
[...]
  drwxr-xr-x   2 virtualabs virtualabs 4096 mai    9 09:40 tmp
  drwxr-xr-x   7 virtualabs virtualabs 4096 mai    9 09:40 usr
  drwxr-xr-x   2 virtualabs virtualabs 4096 mai    9 09:40 var
```
THAT'S A WIN
SECURITY THROUGH OBSCURITY

(Image: XKCD #257)

digital.security
NOT SO OBSCURE AFTERALL

- **Reference manuals** describe how i.MX GPMI works and how data is read/stored on NAND flash memory.

- **Publicly available code on Github** provides a better understanding of critical structures and how things are implemented.
struct fcb_block {
    FCB_ROM_NAND_Timing_t m_NANDTiming;
    //\!< Optimum timing parameters for Tas, Tds, Tdh in nsec.
    uint32_t m_u32PageDataSize;  //\!< 2048 for 2K pages, 4096 for 4K pages.
    uint32_t m_u32TotalPageSize; //\!< 2112 for 2K pages, 4314 for 4K pages.
    uint32_t m_u32SectorsPerBlock; //\!< Number of 2K sections per block.
    uint32_t m_u32NumberOfNANDs; //\!< Total Number of NANDs - not used by ROM.
    uint32_t m_u32TotalInternalDie; //\!< Number of separate chips in this NAND.
    uint32_t m_u32CellType; //\!< MLC or SLC.
    uint32_t m_u32EccBlockNEccType; //\!< Type of ECC, can be one of BCH-0-20
    uint32_t m_u32EccBlock0Size; //\!< Number of bytes for Block0 - BCH
    uint32_t m_u32EccBlockNSize; //\!< Block size in bytes for all blocks other than Block0 - BCH
    uint32_t m_u32EccBlock0EccType; //\!< Ecc level for Block 0 - BCH
}
static inline uint32_t mx28_nand_get_ecc_strength(uint32_t page_data_size,
                                                   uint32_t page_oob_size)
{
    int ecc_strength;

    /*
     * Determine the ECC layout with the formula:
     * ECC bits per chunk = (total page spare data bits) /
     * (bits per ECC level) / (chunks per page)
     * where:
     * total page spare data bits =
     * (page oob size - meta data size) * (bits per byte)
     */
    ecc_strength = ((page_oob_size - MXS_NAND_METADATA_SIZE) * 8) /  
                   (MXS_NAND_BITS_PER_ECC_LEVEL * 
                    mx28_nand_ecc_chunk_cnt(page_data_size));
Y U NO ENCRYPT?

• i.MX systems support NAND flash encryption

• Most of the systems we have tested so far do not use encryption (what did you expect?)
KNOWN VARIANTS

- Some i.MX dumps we made seemed to use a different ECC mechanism

- Various GPMI drivers mention different versions of Freescale ROM and variants of FCB structure

- The current version of `imx-nand-tools` worked for all of our dumps but may fail with yours, so...
INSTALL, TEST, AND CONTRIBUTE!
CONCLUSION

- i.MX system uses a **custom NAND flash layout**
- This **layout is documented** in various documents and publicly available code
- **imx-nand-tools** provides a set of tools to handle this layout and convert dumps into useable images
- i.MX systems **should use NAND flash encryption feature** to avoid key/password/IP leaks
THANKS FOR ATTENDING, ANY QUESTION?

Contact

✉️ damien.cauquil@digital.security
🐦 @virtualabs
RELATED LINKS

- **PTP firmware extraction tips & tricks:**
- **IMX28 Reference manual:**
- **UBOOT NAND utility:**
  https://github.com/u-boot/u-boot/blob/master/tools/mxsboot.c
- **Freescale Linux driver:**