



Intel ME: Two Years Later

Igor Skochinsky
Hex-Rays

Breakpoint 2014
Melbourne

Outline

- Recap (from Breakpoint 2012)
- New discoveries
- Attacking the ME
- ME variations
- Dynamic Application Loader
- Tools/Demo
- Results
- Future work

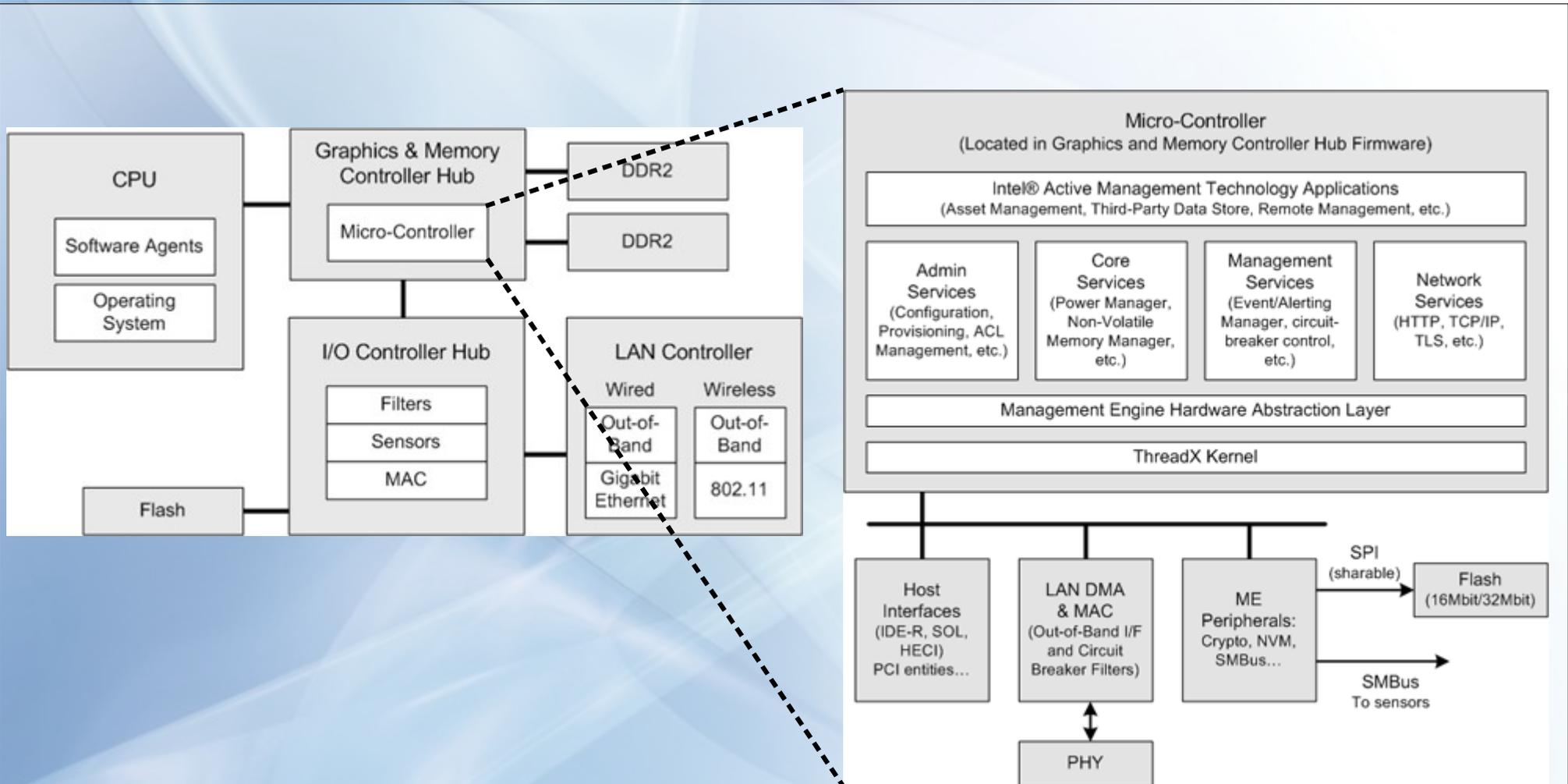
About myself

- Was interested in software reverse engineering for around 15 years
- Longtime IDA user
- Working for Hex-Rays since 2008
- Helping develop IDA and the decompiler (also doing technical support, trainings etc.)
- Have an interest in embedded hacking (e.g. Kindle, Sony Reader)
- Recently focusing on low-level PC research (BIOS, UEFI, ME)
- Moderator of [reddit.com/r/ReverseEngineering/](https://www.reddit.com/r/ReverseEngineering/) and [reverseengineering.stackexchange.com](https://www.stackexchange.com/questions/tagged/reverse-engineering)

ME: Recap

- Management Engine (or Manageability Engine) is a dedicated microcontroller on all recent Intel platforms
- In first versions it was included in the network card, later moved into the chipset (GMCH, then PCH, then MCH)
- Shares flash with the BIOS but is completely independent from the main CPU
- Can be active even when the system is hibernating or turned off (but connected to mains)
- Has a dedicated connection to the network interface; can intercept or send any data without the main CPU's knowledge

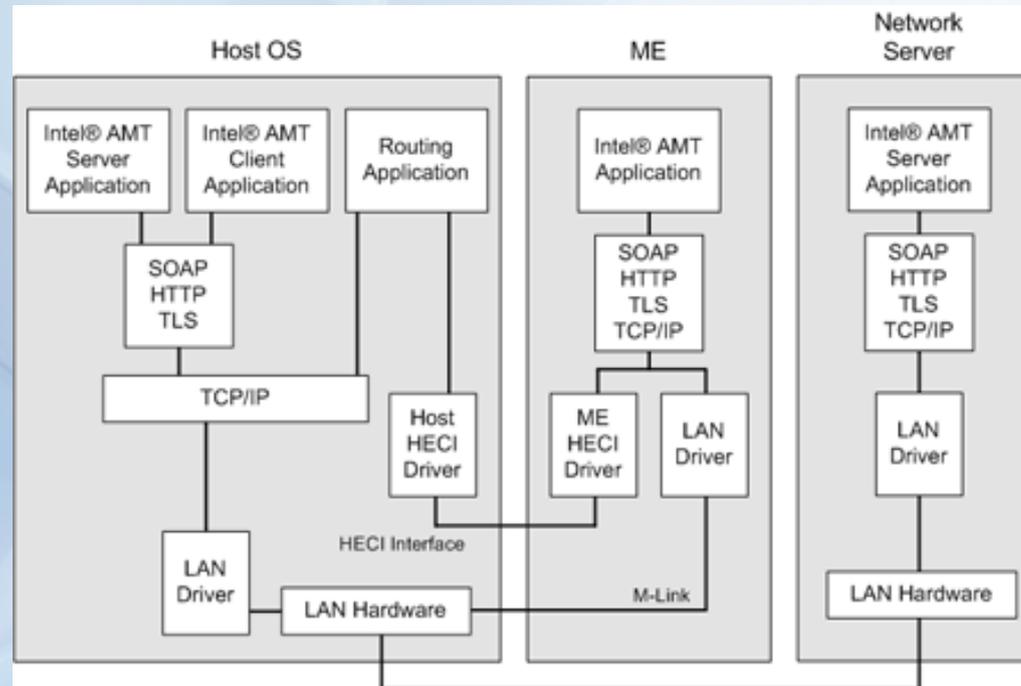
Recap: high-level overview



Credit: Intel 2009

Recap: communication

Communicating with the Host OS and network



- HECI (MEI): Host Embedded Controller Interface; communication using a PCI memory-mapped area
- Network protocol is SOAP based; can be plain HTTP or HTTPS

Recap: ME components

Some of the ME components/features

- Active Management Technology (AMT): remote configuration, administration, provisioning, repair, KVM
- System Defense: lowest-level firewall/packet filter with customizable rules
- IDE Redirection (IDE-R) and Serial-Over-LAN (SOL): boot from a remote CD/HDD image to fix non-bootable or infected OS, and control the PC console
- Identity Protection: embedded one-time password (OTP) token for two-factor authentication
- Protected Transaction Display: secure PIN entry not visible to the host software

Recap

Sources of information

- Intel's whitepapers and other publications (e.g. patents)
- Intel's official drivers and software
 - HECI/MEI driver, management services, utilities
 - AMT SDK, code samples
 - Linux drivers and supporting software; coreboot
- BIOS updates for boards on Intel chipsets
 - Even though ME firmware is usually not updateable using normal means, it's still very often included in the BIOS image
 - Sometimes separate ME firmware updates are available too

Recap

Sources of information

- Intel's ME Firmware kits are not supposed to be distributed to end users
- However, many vendors still put up the whole package instead of just the drivers, or forget to disable the

FTP listing

[PDF Intel® Management Engine System Tools User Guide](#)

<ftp://mx2.kristal.ru/.../System%20Tools%20User%20Guide.pdf>

File Format: PDF/Adobe Acrobat - [Quick View](#)

System Tools User Guide for. Intel® Management Flash Image Tool (FITC)
.....16. 3.1. System Requirements .



[Index of /Driver/Acer Aspire 4738/AutoRun/DRV/Intel Turbo Boost ...](#)

<110.138.195.161/Driver/.../AutoRun/.../Flash%20Image%20Tool/>

5 Jan 2012 – ... Aspire 4738/AutoRun/DRV/Intel Turbo Boost Manageability Engine Code/ MOD01D004C000N000L/Tools/System Tools/Flash Image Tool/ ...

[Gateway ZX4850 Intel iAMT Драйвер v.7.0.0.1144 для Windows 7 ...](#)

<driver.ru/?aid=1026521210333254de1090799368>

... iAMT_Intel_7.0.0.1144_W7x64/Tools/System Tools/Flash Image Tool/fitc.exe 157
2010-12-20 17:46 iAMT_Intel_7.0.0.1144_W7x64/Tools/System Tools/Flash ...

[ACER Veriton M290 Intel iAMT Драйвер v.7.0.0.1144 для Windows 7](#)

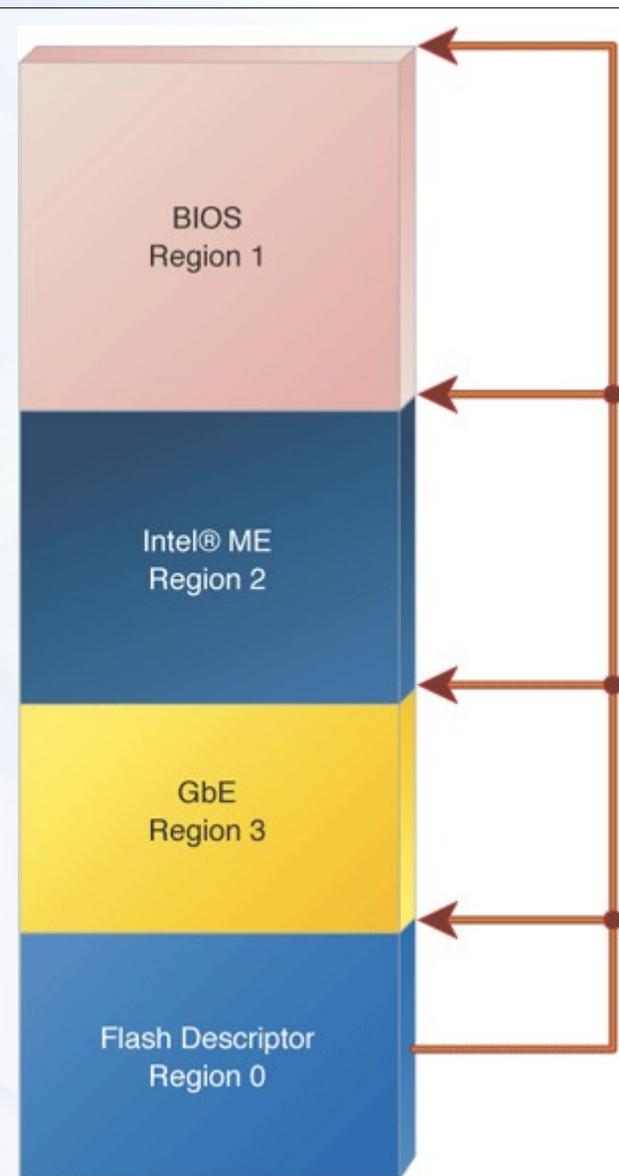
<driver.ru/?aid=10243816228895cec42e66ac5c8d>

... Tools/Flash Image Tool/fitc.exe 157 2011-02-22 11:42 iAMT_Intel_7.0.0.
1144_W7x86x64/Tools/System Tools/Flash Image Tool/fitc.ini 1481 2011-02-22 11:42

With a few picked keywords you can find the good stuff :)

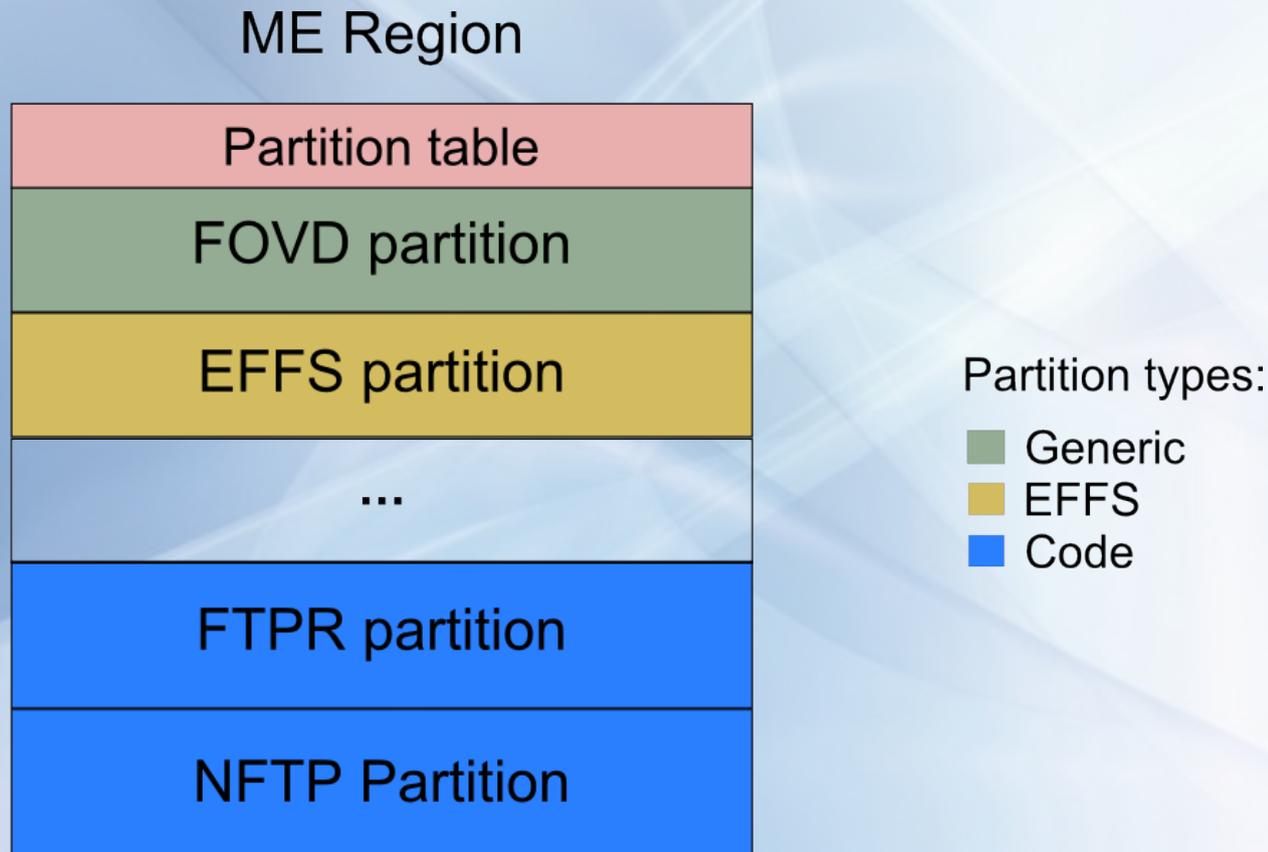
Recap: flash layout

- The SPI flash is shared between BIOS, ME and GbE
- For security, BIOS (and OS) should not have access to ME region
- The chipset enforces this using information in the Descriptor region
- The Descriptor region must be at the lowest address of the flash and contain addresses and sizes of other regions, as well as their mutual access permissions



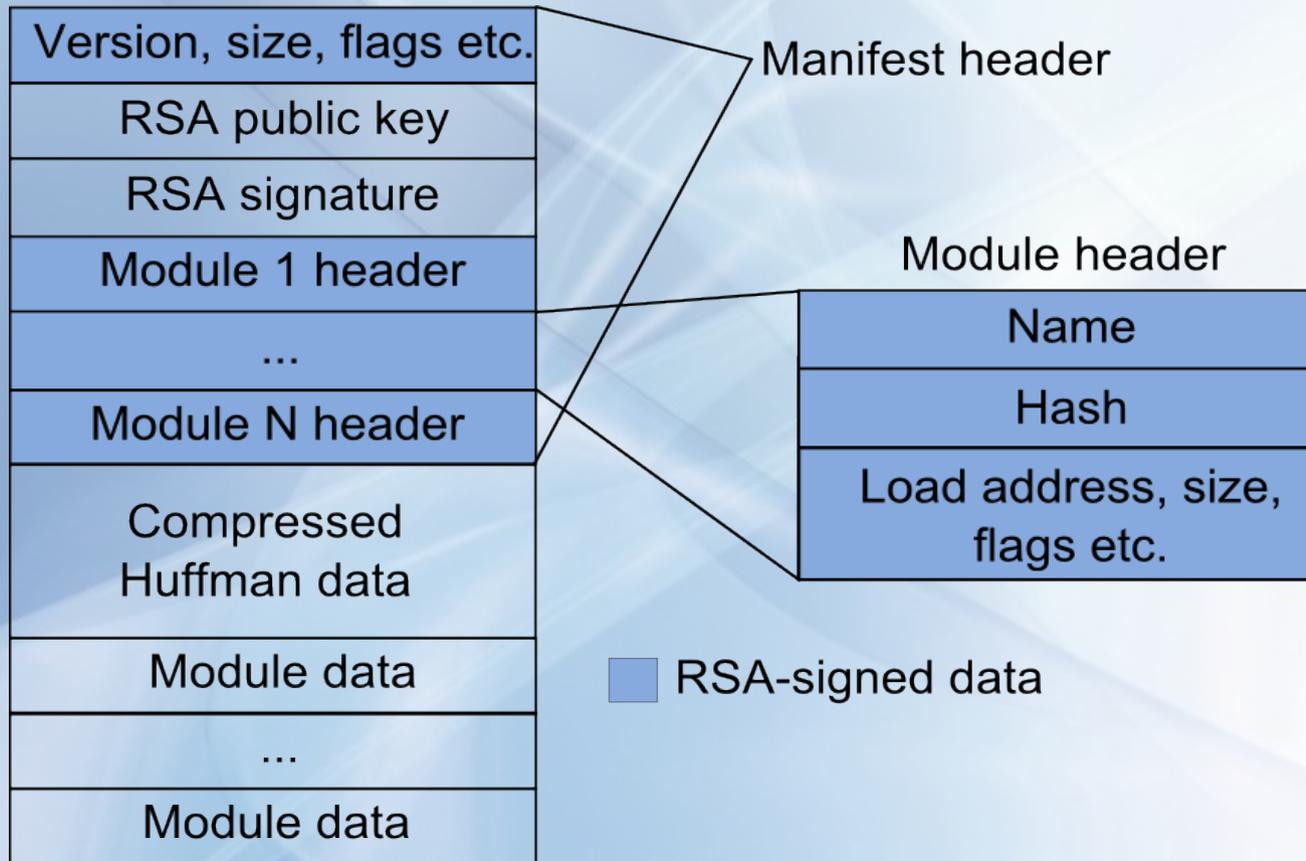
Recap: ME region layout

- ME region itself is not monolithic
- It consists of several partitions, and the table at the start describes them



Recap: ME code partition

- Code partitions have a header called "manifest"
- It contains versioning info, number of code modules, module header, and an RSA signature



Recap: ME code modules

Some common modules found in recent firmwares

Module name	Description
BUP	Bringup (hardware initialization/configuration)
KERNEL	Scheduler, low-level APIs for other modules
POLICY	Secondary init tasks, some high-level APIs
HOSTCOMM	Handles high-level protocols over HECI/MEI
CLS	Capability Licensing Service – enable/disable features depending on SKU, SKU upgrades
TDT	Theft Deterrence Technology (Intel Anti-Theft)
Pavp	Protected Audio-Video Path
JOM	Dynamic Application Loader (DAL) – used to implement Identity Protection Technology (IPT)
fTPM	Firmware TPM

Recap: ME core evolution

- It seems there have been three generations of the microcontroller core so far, and corresponding changes in firmware layout

	ME Gen 1	ME Gen 2	SEC/TXE
ME versions	1.x-5.x	6.x-10.x	1.x (Bay Trail)
Core	ARCTangent-A4	ARC 600(?)	SPARC
Instruction set	ARC (32-bit)	ARCompact (32/16)	SPARC V8(?)
Manifest tag	\$MAN	\$MN2	\$MN2
Module header tag	\$MOD	\$MME	\$MME
Code compression	None, LZMA	None, LZMA, Huffman	None, LZMA

- My investigations cover mostly Gen 2 firmware

Recap: Security

- ME includes numerous security features
- Code signing: all code that is supposed to be running on the ME is signed with RSA and is checked by the boot ROM

“During the design phase, a Firmware Signing Key (FWSK) public/private pair is generated at a secure Intel Location, using the Intel Code Signing System. The Private FWSK is stored securely and confidentially by Intel. Intel AMT ROM includes a SHA-1 Hash of the public key, based on RSA, 2048 bit modulus fixed. Each approved production firmware image is digitally signed by Intel with the private FWSK. The public FWSK and the digital signature are appended to the firmware image manifest.

At runtime, a secure boot sequence is accomplished by means of the boot ROM verifying that the public FWSK on Flash is valid, based on the hash value in ROM. The ROM validates the firmware image that corresponds to the manifest's digital signature through the use of the public FWSK, and if successful, the system continues to boot from Flash code.”

From "Architecture Guide: Intel® Active Management Technology", 2009

Recap: Unified Memory Architecture (UMA) region

- ME requires some DRAM to put unpacked code and runtime variables (MCU's own memory is too limited and slow)
- This memory is reserved by BIOS on ME's request and cannot be accessed by the host CPU once locked.

18:12	RV	0	<i>Reserved</i>
11	RWO	0	Enable for Intel® ME memory region
10	RWO	0	Lock for Intel ME memory region base/mask. This bit is only cleared upon a reset. MESEGMASK and MESEGBASE cannot be changed once this bit is set.
9:0	RV	0	<i>Reserved</i>

- A memory remapping attack was demonstrated by Invisible Things Lab in 2009, but it doesn't work on newer chipsets
- Cold boot attack might be possible, though...

Recap: results and issues (as of 2012)

- Figured out the basic layout of the firmware and the code modules
- Wrote some scripts to parse it
- Learned how to modify hidden BIOS settings
- Added ARC support to IDA
- Started disassembling different modules

Issues:

- Missing code – jumps to nowhere
- Some modules are huffman compressed – could not decompress
- UMA code (supposedly decompressed) is inaccessible

New discoveries

Intel FSP

- Intel Firmware Support Package; first release was in 2013
- Low-level initialization code from Intel for firmware writers
- Freely downloadable from Intel's site
- The package for HM76/QM77 included* ME firmware, tools and documentation

Intel® 7 Series Family- Intel® Management Engine Firmware 8.1

1.5MB Firmware Bring Up Guide

May 2013

| **Revision 1.0**

Intel Confidential

Documentation still contained
"confidential" markings :)

*Intel took it down and replaced with a
generic package, without the secret ME bits :(

<http://www.intel.com/content/www/us/en/intelligent-systems/intel-firmware-support-package/intel-fsp-overview>

ME: the missing code mystery

- To save flash space, various common routines are stored in the on-chip ROM and are not present in the on-flash firmware
- They are used in the firmware modules by jumping to hardcoded addresses
- This complicated reverse-engineering somewhat because a lot of code is missing
- I could guess what some of the functions do, but there were a lot of them
- However, one of the ME images I found contained a new partition I haven't seen before, named "**ROMB**"...

```
ld      r0,  =_sbss?  
ld      r2,  =_ebss?  
mov     r1,  0  
sub     r2,  r2,  r0  
bl      0x205139E4 # memset??  
(address 0x205139E4 is not  
present in the binary)
```

ME: ROM Bypass

- Apparently, the pre-release hardware allows to override the on-chip ROM and boot using code in flash instead
- This is used to work around bugs in early silicon

Binary input file	<p>Navigate to your Source Directory (as specified in Section 2.1) and switch to the Firmware subdirectory. Choose the ME FW binary image.</p> <p>Note: You may choose to build the ME Region only. To do so, Flash Image Descriptor Region Descriptor Map parameter Number of Flash components must be set to 0.</p> <p>Note: Loading an ME FW binary image that contains ME <u>ROM Bypass</u> unlocks the <u>ME Boot from Flash</u> parameter in Flash Image Descriptor Region PCH Straps PCH Strap 10.</p>
-------------------	---

<u>ME boot from Flash</u>	false (grayed out)	<p>false (default) = No ME Region binary loaded, or ME Region binary does not contain ME <u>ROM bypass</u> image</p> <p>Note: On B0 and later PCH stepping parts this setting should be set to 'false'</p>
---------------------------	-----------------------	--

ME: ROM Bypass

- If this option is on, the first instruction of the ME region is executed instead of the boot ROM
- It jumps to the code in ROMB partition

ROM Bypass Image



ME: ROM Bypass

- By looking at the code in the ROMB region, the inner workings of the boot ROM were discovered
- The boot ROM exposes for other modules:
 - common C functions (memcpy, memset, strcpy etc.)
 - ThreadX RTOS routines
 - Low-level hardware access APIs
- It does basic hardware init
- It verifies signature of the FTPR partition, loads the BUP module and jumps to it
- Unfortunately, BUP and KERNEL employ Huffman compression with unknown dictionary, so their code is not available for analysis :(

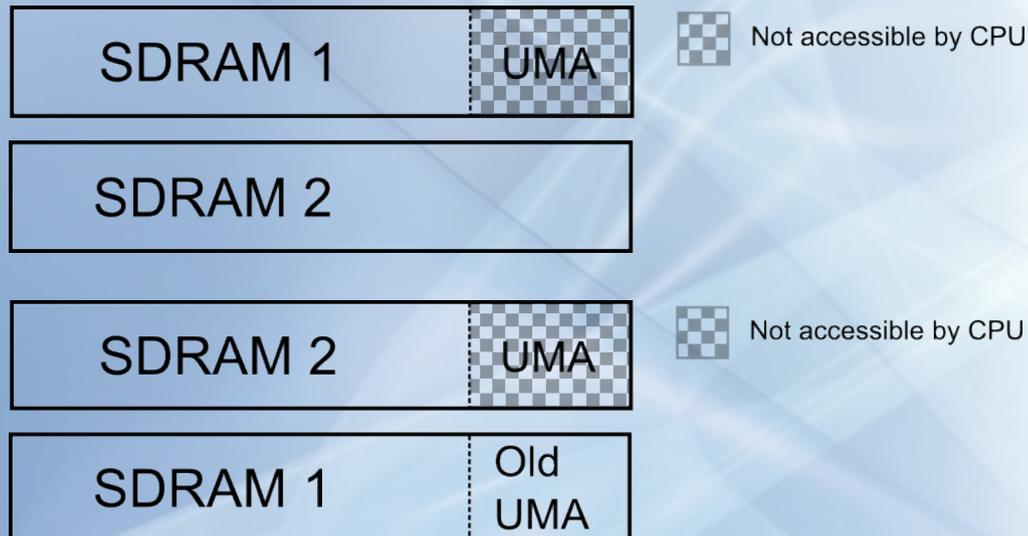
Attacking the ME

ME: attacking UMA

- I decided to try and dump the UMA region since it contains unpacked Huffman code and runtime data
- Idea #1: simply disable the code which sets the MESEG lock bit in the BIOS
- [some time spent reversing memory init routines...]
- Patched out the code which sets the lock bit
- Updated necessary checksums in the UEFI volume
- Reflashed the firmware and rebooted
- Result: bricked board
- Good thing I had a spare board and could restore the old firmware using hotswap flashing...

ME: attacking UMA

- Idea #2: cold boot attack
- Quickly swap the DRAM sticks so that UMA content remains in memory



First Boot: Let ME
unpack code into UMA

Second boot: after swapping,
Old UMA should be accessible

- Unfortunately, dumped memory contains only garbage...

ME: attacking UMA

- Bought lower-speed memory – did not help
- Bought professional grade freezing spray – did not help
- Eventually discovered that DDR3 used in my board can employ memory scrambling

“The memory controller incorporates a DDR3 Data Scrambling feature to minimize the impact of excessive di/dt on the platform DDR3 VRs due to successive 1s and 0s on the data bus. [...] As a result the memory controller uses a data scrambling feature to create pseudo-random patterns on the DDR3 data bus to reduce the impact of any excessive di/dt.”

(from Intel Corporation Desktop 3rd Generation Intel® Core™ Processor Family, Desktop Intel® Pentium® Processor Family, and Desktop Intel® Celeron® Processor Family Datasheet)

ME: attacking UMA

- Idea #3: use different UMA sizes across boots
- The required UMA size is a field in the \$FPT header
- The FPT is protected only by checksum – not signature – so it's easy to change

```
000000: 20 20 80 0F 40 00 00 10 | 00 00 00 00 00 00 00 00   Ъа@ ▶
000010: 24 46 50 54 13 00 00 00 | 20 10 30 DA 07 00 64 00   $FPT!! ▶0b• d
000020: 20 00 00 00 01 FC FF FF | 00 00 00 00 00 00 00 00   0ьяя
000030: 46 4F 56 44 4B 52 49 44 | 00 04 00 00 00 00 0C 00   FOVDKRID ◆ ♀
000040: 01 00 00 00 01 00 00 00 | 00 00 00 00 00 83 07 00   @ @ í•
000050: 4D 44 45 53 4D 44 40 44 | 00 10 00 00 00 00 00 00   MDESMDID ▶ ▶
000060: 00 00 00 00 00 00 00 00 | 00 00 83 23 00 00 00 00   @ @ í#
000070: 16 13 50 53 15 53 10 11 | 00 00 00 00 00 00 10 00   F00000TD
```

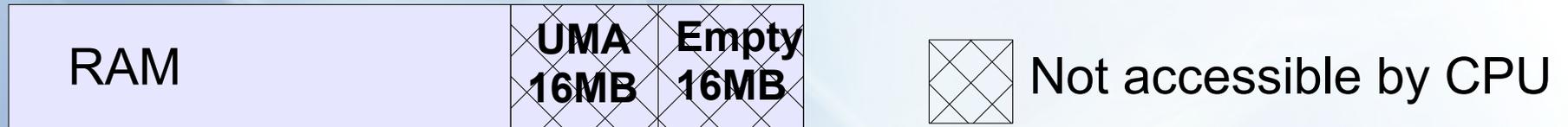
UMA Size (MB) (points to 20 00 00 00)

FPT header (points to 20 10 30 DA 07 00 64 00)

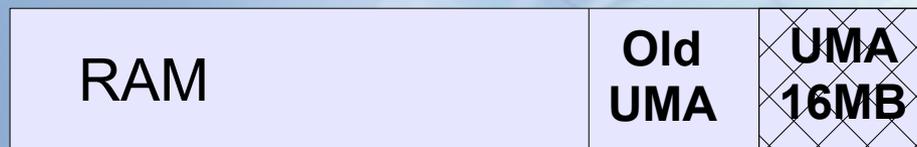
Checksum (points to DA)

ME: attacking UMA

- Flash FPT that requests 32MB, reboot. BIOS will reserve top 32MB but ME will use only half of the region



- Flash FPT that requests 16MB, reboot. BIOS will reserve top 16MB, so the previously used 16MB will be accessible again



- Unfortunately got garbage again :(It seems that memory is reinitialized with different scrambling seed between the boots.

ME: attacking UMA

- Idea #4: disable memory scrambling
- Scrambling can be turned off using a BIOS setting on some boards

Memory Scrambler

Values: Enabled, Disabled

Enables or disables Memory Scrambler support.

Scrambler Seed Generation

Values: Enabled, Disabled

Enables or disables the generation of a scrambler seed for security purposes. The memory scrambler scrambles the contents of memory in the DIMMs so that they cannot be removed and read. When enabled, a scrambler seed is not generated. When disabled, a scrambler seed is always generated.

- On my board the option is hidden but it's possible to change it by editing the UEFI variable "Setup" directly
- However, it did not help – the memory is still garbage
- Probably caused by aggressive memory training

ME: attacking UMA

- Idea #5: ?
- I still have some ideas to try but they require more time and effort
- So I tried other approaches
- For example...

ME variations

Server Platform Services

- On Intel's server boards, ME is present too
- However, it runs a different kind of firmware
- It's called Server Platform Services (SPS)
- It has a reduced set of modules, however it does include BUP and KERNEL
- Good news #1: BUP module is not compressed!
- KERNEL is Huffman "compressed", but...
- Good news #2: all blocks use trivial compression (i.e. no compression)
- So I now could investigate how these two modules work
- There are differences from desktop but it's a start

Trusted Execution Engine

- In Bay Trail (Atom-based SoC), another variation of ME is used
- Marketing name: Trusted Execution Engine (TXE);
codename: SEC/SeC
 - Note: not related to Trusted Execution **Technology** (TXT)
- Instead of ARC, uses SPARC core(!)
- No Huffman compression, only LZMA(!!)
- So, all code (except Boot ROM) is available for analysis
- The available KERNEL code can help recovering APIs for ARC firmwares too
- SPARC emulators are available so the code can be emulated/fuzzed/debugged

Trusted Execution Engine

- Here's what I've discovered so far
- The firmware format is the same, just with larger module headers
- ThreadX doesn't seem to be used anymore; all RTOS functionality (threads, semaphores etc.) is implemented directly inside KERNEL
- However, other common routines from boot ROM are still used
- Because most of the other modules used KERNEL wrappers for RTOS stuff, they haven't changed substantially
- Module set is reduced compared to desktop ME (e.g. network-related modules are missing)
- fTPM module implements TPM 2.0

Dynamic Application Loader

JOM aka DAL

- The "JOM" module appeared in ME 7.1
- It implements what Intel calls "Dynamic Application Loader" (DAL)
- It allows to upload and run applications (applets) inside ME dynamically (i.e. at runtime)
- This feature is used to implement Intel's Identity Protection Technology (Intel IPT)
- In theory, it allows a much easier way for running custom code on the ME
- Let's have a look at how it's implemented...

JOM aka DAL

- Some interesting strings from the binary:

```
Could not allocate an instance of
java.lang.OutOfMemoryError
linkerInternalCheckFile: JEFF format version not
supported
com.intel.crypto
com.trustedlogic.isdi
Starting VM Server...
```

- Looks like Java!

JOM aka DAL

- Apparently it includes a Java VM implementation
- In Intel ME drivers, there is a file "oath.dalp" with a Base64 blob
- After decoding, a familiar manifest header appears
- It has a slightly different module header format, and a single module named "Medal App"
- The module contains a chunk with signature "JEFF", which is mentioned in the strings of the JOM module
- Strings in this JEFF chunk also point to it being Java code
- However, the opcode values look different from normal Java
- I was so sure it's a custom format, I spent quite a lot of time reversing it from scratch

JOM aka DAL

- However, I came across one string in the module...

```
.ascii "Invalid constant offset in the SLDC instruction"
```

- There is no such instruction in standard Java. Let's try Google...

About 3,260 results (0.26 seconds)

[\[PDF\] JavaBirthmarks - Detecting the Software Theft --](#)

[se-naist.jp/old/jbirth/papers/tamada05ieice.pdf](#)

of Java programs. Specifically, WePropose Java birthmarks to support the ... sldC (54) 4 The adversary must be highly skilled in Java bytecode to modify a ...

[pubs Sdiff docs/technotes/guides/pack200](#)

[cr.openjdk.java.net/~ksrini/8007297/.../pack-spec.html.sdiff.html](#) ▾

p> 5200 <p>Every bytecode instruction is contained by a class, called the 5201 <tt>sldc</tt> and 5196 <tt>sldc_w</tt>, as <tt>aldc</tt> and <tt>aldc_w</tt>.

[Crap shit head - SlideShare](#)

[www.slideshare.net/shashgibbs88/crap-shit-head](#) ▾

FUNDAMENTALS 11 3.1 JDK & JRE 11 3.2 Net Beans 6.8 11 3.3 Java compiled to the bytecode instruction set and binary format defined in the Java Virtual 2) SLDC: Software Development Life Cycle 3) JSP: Java Server Pages 4) DFD: ...

[\[PDF\] L2/02-042 - Unicode Consortium](#)

[www.unicode.org/L2/L2002/02042-jeff-spec.pdf](#) ▾

Java is a registered trademark of Sun Microsystems, Inc. in the United States and in other The VMConstUtf8 structures are referred by the sldc bytecode.

[Browse - Project Kenai](#)

[https://kenai.com/bugzilla/describecomponents.cgi](#)

bwshop: bw. bytecodeviewer: View Java ByteCode. bytest: Testing developerdocs: Documentation hub for developers used for the SLDC - CMM compliant.

[\[PDF\] The JEFF storage format](#)

[www.soj.city.ac.uk/~kloukin/IN2P3/.../JeffDraftSpecs2002March7.pdf](#) ▾

Mar 7, 2002 - Java is a registered trademark of Sun Microsystems, Inc. in the United States and in other countries. 4.2.10 The wide <opcode> Opcodes .

n"

- How
- .asci
- There
- Goog

JEFF File Format

- Turns out the JEFF format *is* a standard
- Was proposed in 2001 by the now-defunct J Consortium
- Has been adopted as an ISO standard (ISO/IEC 20970)
- Draft specification is still available in a few places
- Optimized for embedded applications
- Combines several classes in one file, in a form which is ready for execution
- Shared constant pool also reduces size
- Introduces several new opcodes
- Supports native methods defined by the implementation

JEFF File Format

- I made a dumper/disassembler in Python based on the spec
- Dumped code in oath.dalp and the internal JEFF in the firmware
- No obfuscation was used by Intel, which is nice
- Most of the basic Java classes are implemented in bytecode, with a few native helpers
- There are classes for:
 - Cryptography
 - UI elements (dialogs, buttons, labels etc.)
 - Flash storage access
 - Implementing loadable applets

JEFF File Format

- Fragment of a class implementation (without bytecode)

```
Class com.intel.util.IntelApplet
private:
    /* 0x0C */ boolean m_invokeCommandInProgress;
    /* 0x00 */ OutputBufferView m_outputBuffer;
    /* 0x0D */ boolean m_outputBufferTooSmall;
    /* 0x04 */ OutputValueView m_outputValue;
    /* 0x08 */ byte[] m_sessionId;
public:
    void <init>();
    final int getResponseBufferSize();
    final int getSessionId(byte[], int);
    final int getSessionIdLength();
    final String getUUID();
    final abstract int invokeCommand(int, byte[]);
    int onClose();
    final void onCloseSession();
    final int onCommand(int, CommandParameters);
    int onInit(byte[]);
    final int onOpenSession(CommandParameters);
    final void sendAsynchMessage(byte[], int, int);
    final void setResponse(byte[], int, int);
    final void setResponseCode(int);
```

IPT applets

- The applet interface seems to be rather simple
- The OATH applet implementation looks like this:

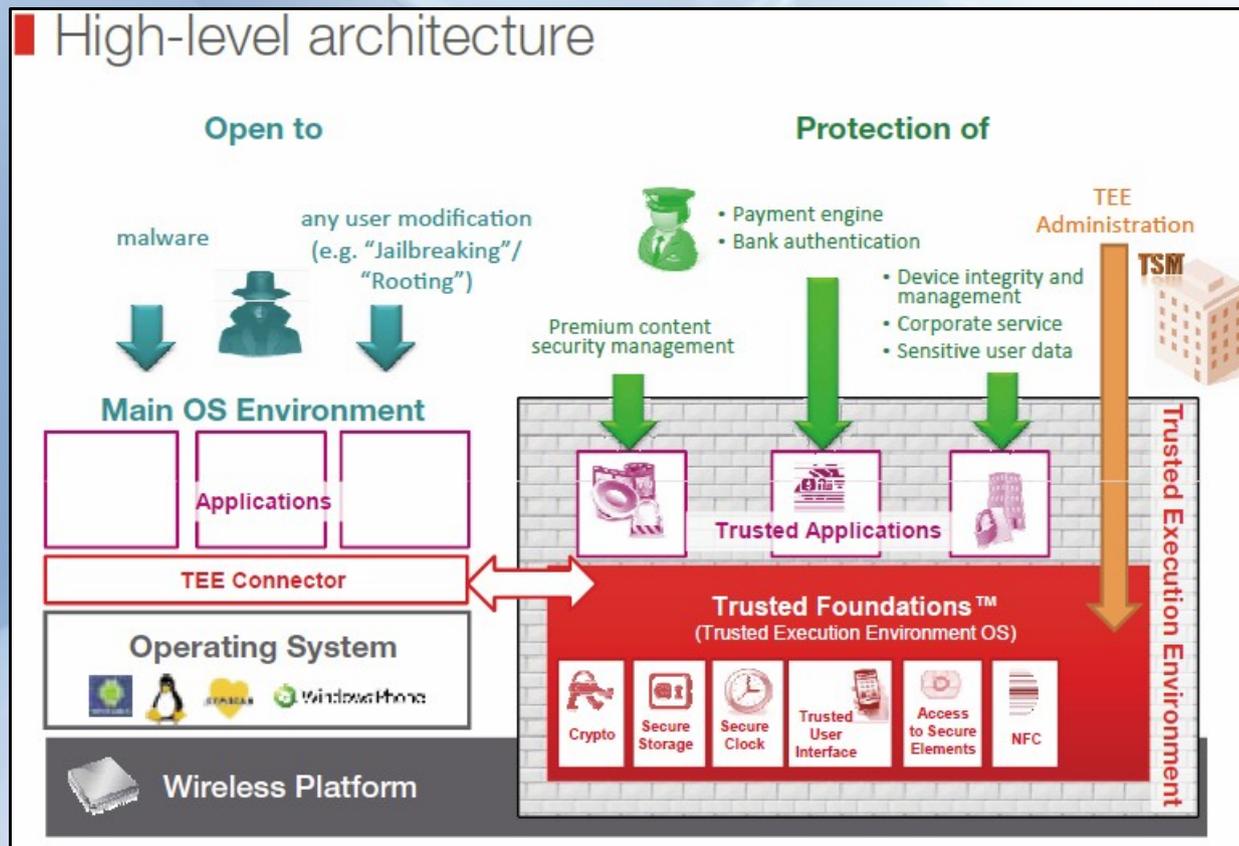
```
package com.intel.dal.ipt.framework;
public class AppletImpl extends com.intel.util.IntelApplet
{
    final int invokeCommand(int, byte[])
    {
        ...
    }
    int onClose()
    {
        ...
    }
    int onInit(byte[])
    {
        ...
    }
}
```

IPT applets

- Unfortunately, even if I create my own applets, I can't run them inside ME because...
- Applet binaries have a signed manifest header and are verified before running
- Still, there may be vulnerabilities in the protocol, which is pretty complicated

Trusted Execution Environment

- From the strings inside JOM, it's apparent that Intel is using a Trusted Execution Environment (TEE) provided by Trusted Logic Mobility (now Trustonic), called "Trusted Foundations"



Source:
Trusted Foundations flyer

Trusted Execution Environment

- Trusted Foundations is also used in several smartphones
- Implemented there using ARM's TrustZone
- Due to GPL, source code of drivers which communicate with Trusted Foundations is made available
- The protocol is not the same as what Intel uses
- For example, TrustZone communications employ shared memory, while ME/JOM only talks over HECI/MEI
- Still, there are some common parts, so it helps in reverse engineering

Trusted Execution Environment

- There is a TEE specification released by the GlobalPlatform association (Trusted Logic Mobility/Trustonic is a member)
- Describes overall architecture, client API and internal API (for services running inside TEE)
- Again, it does not exactly match what runs in the ME but is still a useful reference

<http://www.globalplatform.org/specificationsdevice.asp>

Demo (scripts/tools)

Results so far

- I *still* have not managed to run my own rootkit on the ME
- But I'm getting a more complete picture of how ME works
- Other researchers started looking into it as well
- The code of boot ROM, BUP and KERNEL modules has been discovered
- This allowed me to map out many APIs used in other modules
- ARC support was released with IDA 6.4 and improved in the following versions
- There was some interest so I will be releasing my scripts at this Breakpoint
- <https://github.com/skochinsky>

Future work

- Dynamic Application Loader
 - Make a JEFF to .class converter, or maybe a direct JEFF decompiler
 - Reverse and document the host communication protocol
 - Linux IPT client?
- EFFS parsing and modifying
 - Most of the ME state is stored there
 - If we can modify flash, we can modify EFFS
 - Critical variables are protected from tampering but the majority isn't
 - Complicated format because of flash wear leveling

Future work

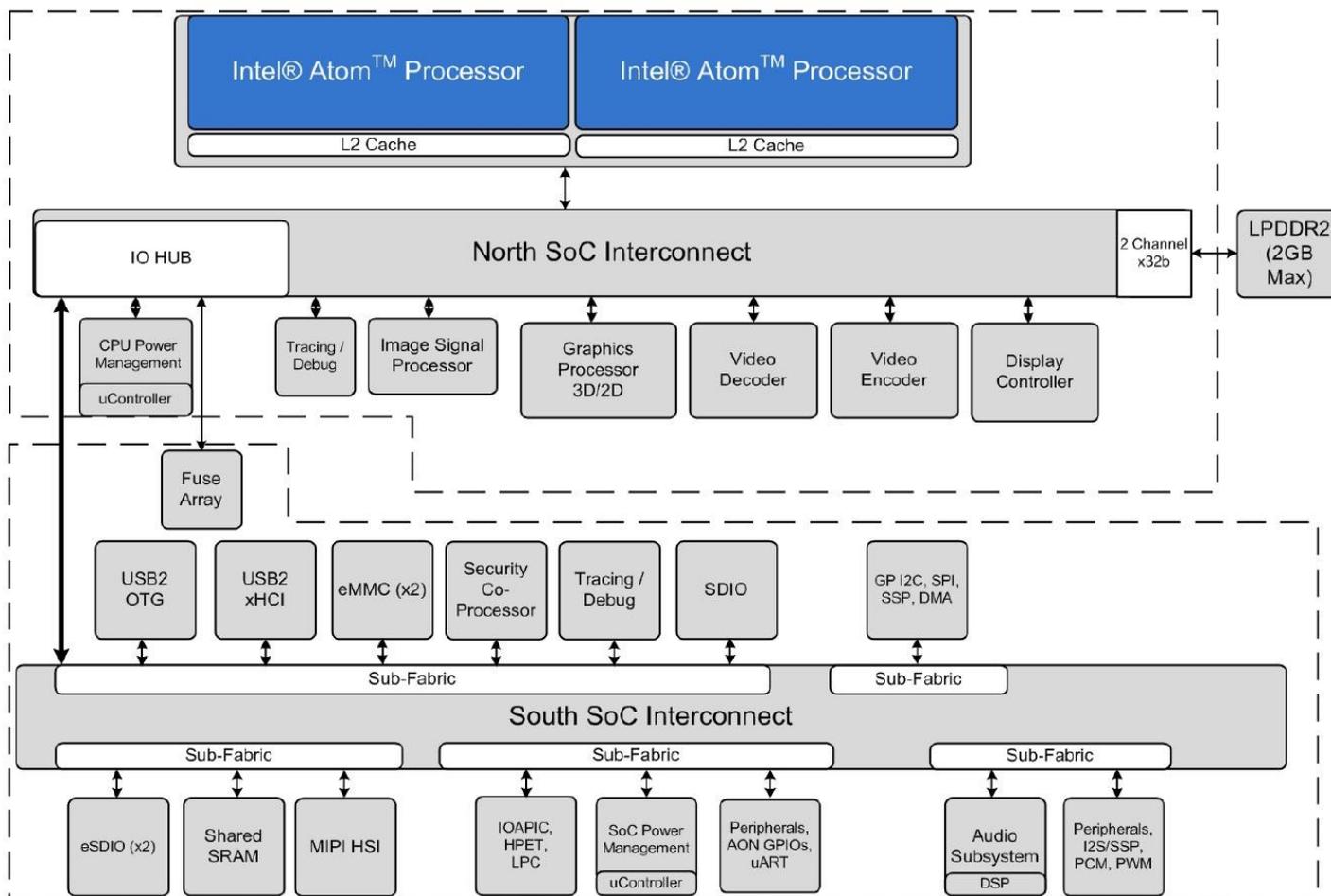
- Huffman compression
 - Used in Gen2 firmwares for compressing the kernel and some other modules
 - Apparently the dictionary is hardcoded in silicon
 - There was some progress with ME 6.x:
<http://io.smashthestack.org:84/me/>
 - Newer versions use a different dictionary :(
- ME ↔ Host protocols
 - Most modules use different message formats
 - A lot of undocumented messages; some modules seem to be not mentioned anywhere
 - Some of the client software has very verbose debugging messages in their binaries...
 - Anti-Theft is probably a good target

Future work

- BIOS RE
 - In early boot stages ME accepts some messages which are refused later
 - Reversing BIOS modules that talk to ME is a good source of info
 - Some messages can be sent only during BIOS boot
 - UEFITool by Nikolaj Schlej helps in editing UEFI images <https://github.com/NikolajSchlej/UEFITool>
 - Coreboot has support for ME on some boards
- Simulation and fuzzing
 - Open Virtual Platform (www.ovpworld.org) has modules for ARC600 and ARC700 (ARCompact-based)
 - Supposedly easy to extend to emulate custom hardware
 - Debugging and fuzzing should be possible

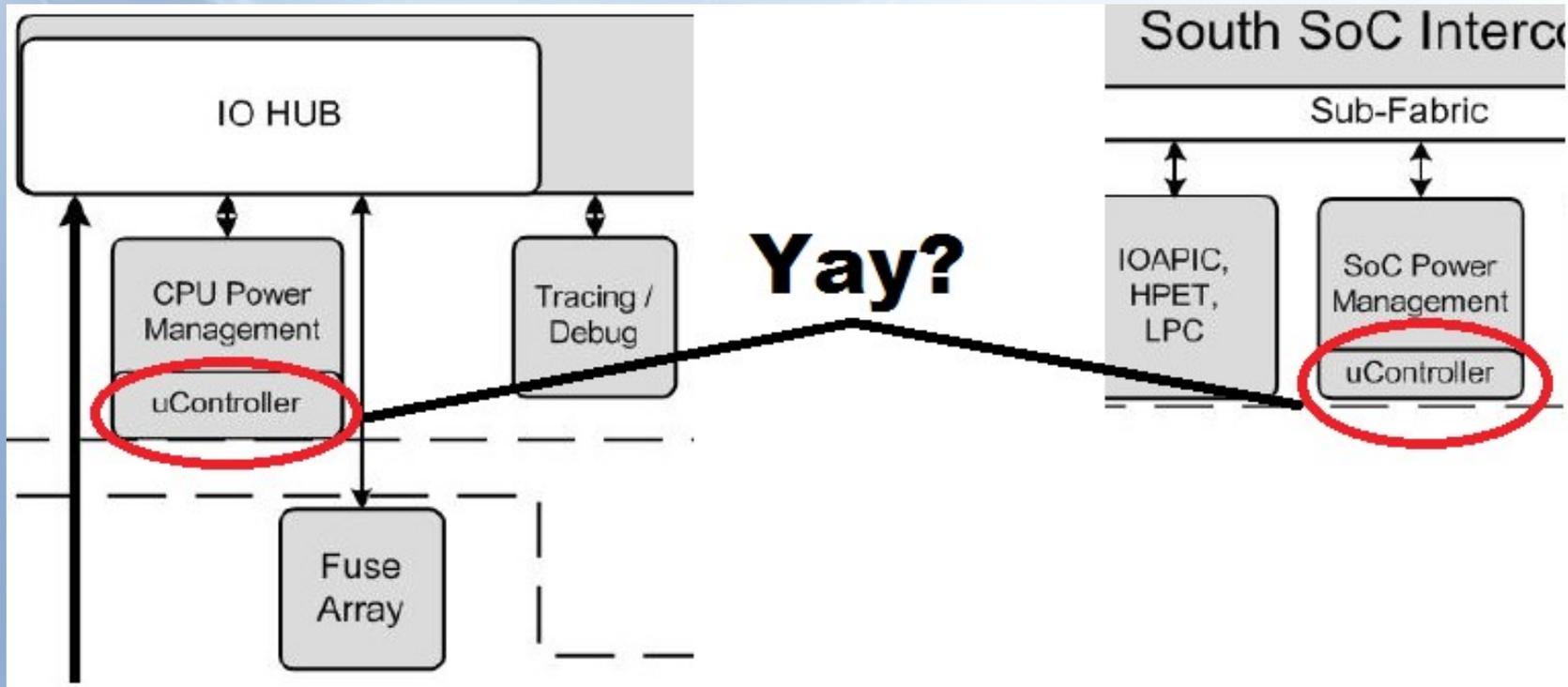
Future work: Atom SoCs?

Next Generation SoC – Block Diagram



IDF2012
INTEL DEVELOPER FORUM

Future work: Atom SoCs?



Future work: Atom SoCs?

- Intel System-on-Chip (SoC) variants (Moorestown, Medfield, Merrifield etc.), used in some phones and tablets
- In addition to the x86 core(s), also include mysterious blocks like "P-Unit" or "SCU"
- Apparently those have their own firmware(!)
- P-Unit seems to be an 8051 and SCU an ARC(!)
- From a quick glance they don't seem to be extremely hardened
- Communicate with the CPU over "sideband fabric"(??)
- The new Intel Edison has such a processor
- The firmware images are available...

<http://downloadmirror.intel.com/24271/eng/edison-image-ww36-14.zip>

References and links

<http://software.intel.com/en-us/articles/architecture-guide-intel-active-management-technology/>

http://software.intel.com/sites/manageability/AMT_Implementation_and_Reference_Guide/

<http://theinvisiblethings.blogspot.com/2009/08/vegas-toys-part-i-ring-3-tools.html>

<https://noggin.intel.com/technology-journal/2008/124/intel®-vpro™-technology>

http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/100402-Vassilios_Ververis-with-cover.pdf

<http://www.stewin.org/papers/dimvap15-stewin.pdf>

http://www.stewin.org/techreports/pstewin_spring2011.pdf

<http://www.stewin.org/slides/pstewin-SPRING6-EvaluatingRing-3Rootkits.pdf>

http://flashrom.org/trac/flashrom/browser/trunk/Documentation/mysteries_intel.txt

<http://review.coreboot.org/gitweb?p=coreboot.git;a=blob;f=src/southbridge/intel/bd82x6x/me.c>

http://download.intel.com/technology/product/DCMI/DCMI-HI_1_0.pdf

<http://me.bios.io/>

http://www.uberwall.org/bin/download/download/102/lacon12_intel_amt.pdf

Thank you!

Questions?

igor@hex-rays.com
skochinsky@gmail.com

ME internals: Huffman compression

- If huffman-compressed modules are present in a partition, a single compressed data stream is used for all of them
- The stream follows the manifest and starts with a header:

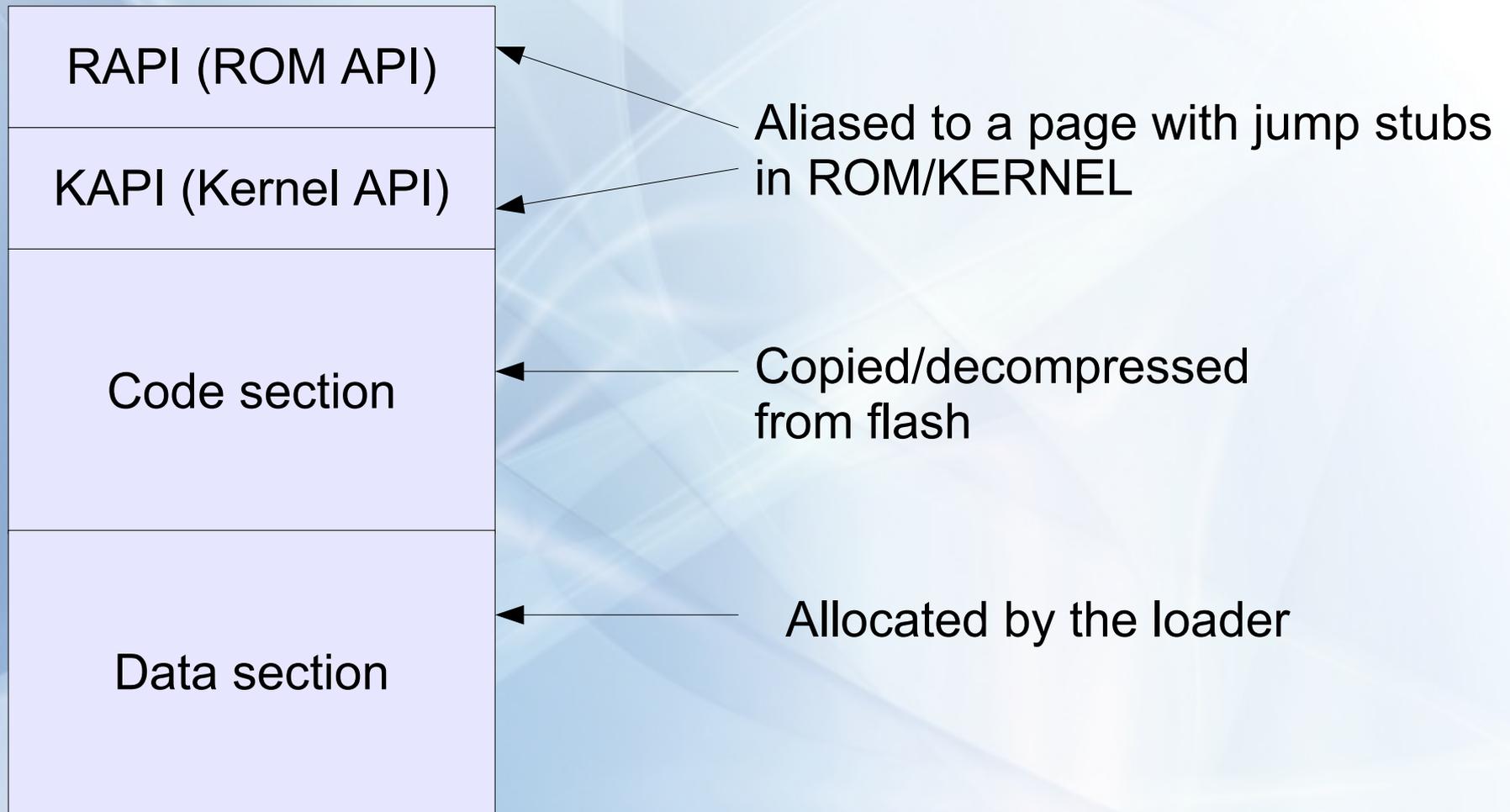
```
struct LutHeader {
    char    Signature[4]; // 00 'LLUT' or 'GLUT' or ' LUT'
    uint32  ChunkCount;   // 04 number of compressed chunks
    uint32  AddrBase;     // 08 base address of unpacked data
    uint32  SpiBase;      // 0C offset of the LUT in the ME region
    uint32  HuffLength;   // 10 Total length of the huff stream
    uint32  HuffStart;    // 14 offset to Huff data in ME region
    uint32  Flags;        // 18 bit0: enable 1K pages
    uint32  Reserved[5];  // 1C
    uint32  PageSize;     // 30 uncompressed size of each chunk
    uint16  version[2];   // 34 version of the compression tool
    char    Chipset[8];   // 38 'PCH A0' or 'CPT A0'
}
```

ME internals: Huffman compression

- Following the header is the chunk index table
- Each table entry is 32 bits: top 7 bits are flags and low bits are offset to the compressed data for the chunk
- The entry index determines the address of the unpacked data – single entry covers a 1K chunk (0x400 bytes)
- For example, in this table entries 0-3 are empty (zero) pages, 4 is uncompressed and 5 and 6 are compressed using two different dictionaries
- More info:
<http://io.smashthestack.org/me/>

index	entry	vaddr
0000:	80000000	[20040000]
0001:	80000000	[20040400]
0002:	80000000	[20040800]
0003:	80000000	[20040C00]
0004:	000D2740	[20041000]
0005:	400D2B40	[20041400]
0006:	C00D2F40	[20041800]

ME internals: code module memory layout



ME internals: RAPI (ROM API)

- One page of memory, aliased to a jump table in ROM
- Contains jumps to various APIs in ROM, and a few pointers to internal ROM variables
- Code in the code section calls the stubs in the RAPI page
- Layout changes between ME versions, but not drastically

xxxxx9DC	j	memcpy
xxxxx9E4	j	memset
xxxxx9EC	j	strncmp
xxxxx9F4	j	memchr
xxxxx9FC	j	memcmp
xxxxxA04	j	strcmp
xxxxxA0C	j	strlen

Fragment of a RAPI page.
First part of the address changes for each module while the page offset (last three digits) stays the same.

ME internals: KAPI (Kernel API)

- One page of memory, aliased to a jump table in KERNEL
- Has two versions: for privileged and non-privileged modules
- Consists of short stubs like this:

```
xxxxx090 kern_malloc:  
xxxxx090     mov     r8, 0x70014  
xxxxx098     b      kapi_dispatch_priv
```

- Low 16 bits of r8 are used as offset into the table of kernel APIs, high bits are flags (e.g. marking the call as privileged or non-privileged)
- The module's code calls addresses in the KAPI page

ME internals: inter-module calls

- Any module can expose additional APIs to others, by using a kernel API and a table of interfaces

```
mov    r0, r13      # pointer to result handle
mov    r1, =hciTable # entries
mov    r2, 2        # entry count
bl     kern_register_interfaces
```

hciTable:

```
# ID 0x1001 is used for notification function
IntfInitEntry <0x1001, HciInterface_table, 1, 0>
# other IDs are for arbitrary interfaces
IntfInitEntry <0x1037, Iface1037_table, 0x200, 0>
```

Iface1037_table:

```
.long 0xC00000      ; flags and number of methods
.long Iface1037_04 ; method 1
.long Iface1037_08 ; method 2
.long Iface1037_0C ; method 3
```

ME internals: inter-module calls

- Other modules can then request the interface by its ID:

```
ld      r1, =queryTbl  # table
mov     r2, 1          # count
bl      kapi_query_interfaces

queryTbl:
.long 0x1037          # interface ID
.long pIntf1037      # pointer to fill
```

- and call the methods from the table:

```
ld      r0, =pIntf1037 # load pointer
ld      r2, [r0,8]     # load method 2 ptr
add     r1, sp, 4      # set up arguments
jl      [r2]           # call Iface1037_08
```