Intel ME Secrets

Hidden code in your chipset and how to discover what exactly it does

Igor Skochinsky
Hex-Rays

RECON 2014
Montreal

Outline

- High-level overview of the ME
- Low-level details
- ME security and attacks
- Dynamic Application Loader
- Results
- Future work

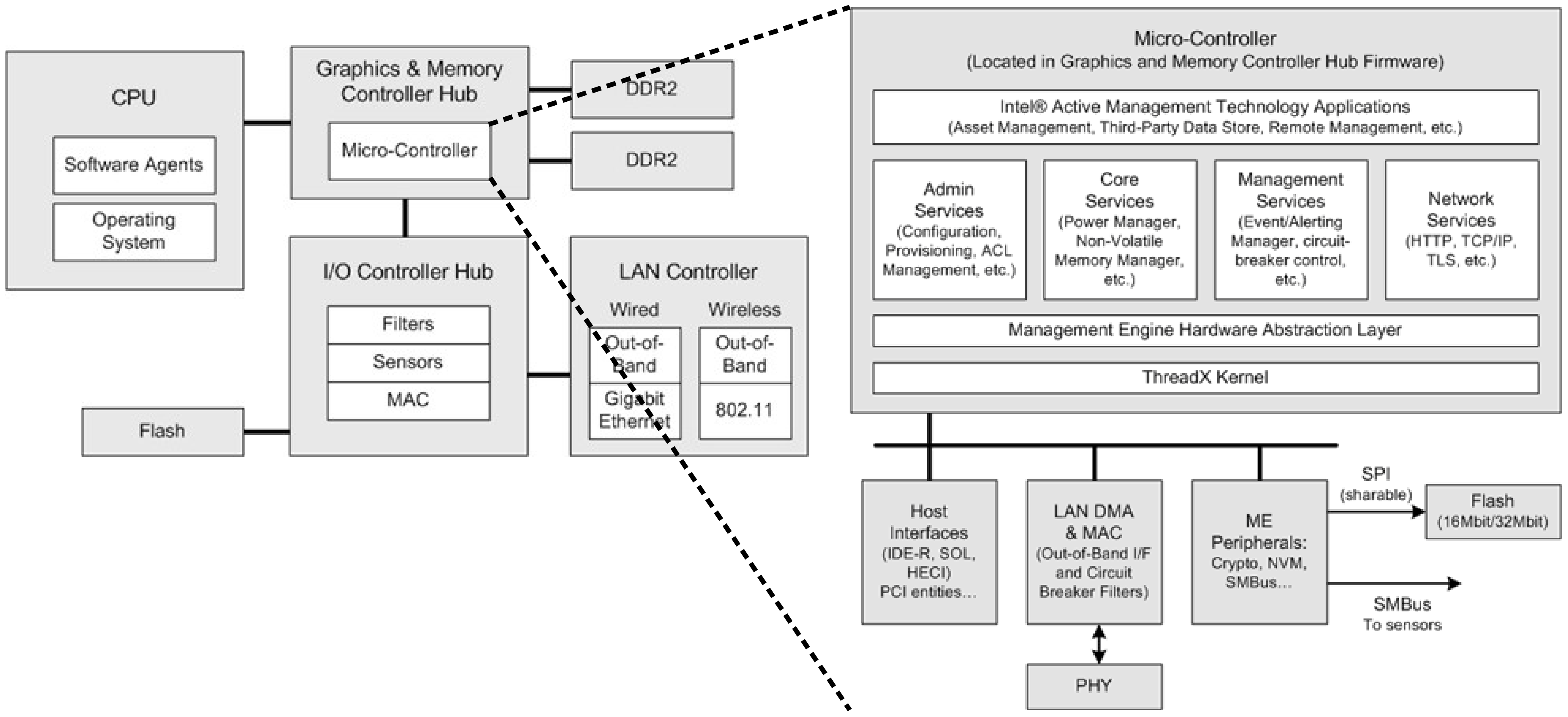
About myself

- 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/

ME: High-level overview

- 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 main CPU's knowledge

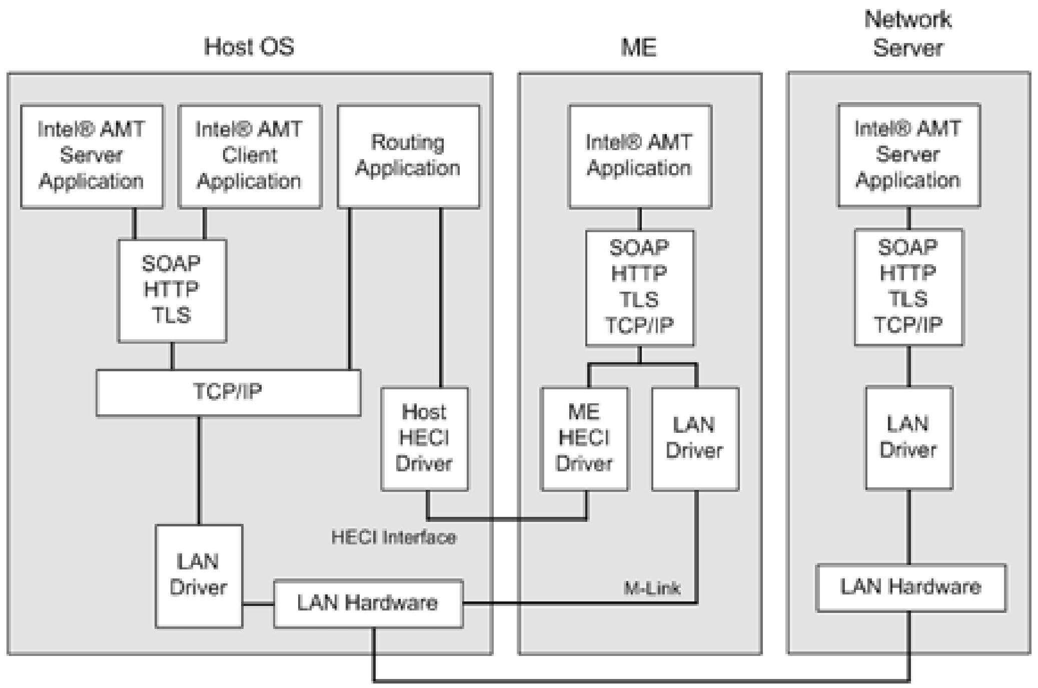
ME: High-level overview


CPU
Software Agents
Operating System
Graphics & Memory Controller Hub
Micro-Controller
DDR2
DDR2
I/O Controller Hub
Filters
Sensors
MAC
Flash
LAN Controller
Wired
Wireless
Out-of-Band
Gigabit Ethernet
Out-of-Band
802.11
Micro-Controller (Located in Graphics and Memory Controller Hub Firmware)
Intel® Active Management Technology Applications (Asset Management, Third-Party Data Store, Remote Management, etc.)
Admin Services (Configuration, Provisioning, ACL Management, etc.)
Core Services (Power Manager, Non-Volatile Memory Manager, etc.)
Management Services (Event/Alerting Manager, circuit-breaker control, etc.)
Network Services (HTTP, TCP/IP, TLS, etc.)
Management Engine Hardware Abstraction Layer
ThreadX Kernel
Host Interfaces (IDE-R, SOL, HECI) PCI entities...
LAN DMA & MAC (Out-of-Band I/F and Circuit Breaker Filters)
ME Peripherals: Crypto, NVM, SMBus...
SPI (sharable)
Flash (16Mbit/32Mbit)
SMBus To sensors
PHY

Credit: Intel 2009

ME: High-level overview

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

ME: High-level overview

Some of the ME components

- 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

ME: High-level overview

Intel Anti-Theft

- PC can be locked or disabled if it fails to check-in with the remote server at some predefined interval; if the server signals that the PC is marked as stolen; or on delivery of a "poison pill"
- Poison pill can be sent as an SMS if a 3G connection is available
- Can notify disk encryption software to erase HDD encryption keys
- Reactivation is possible using previously set up recovery password or by using one-time password

ME: Low-level details

ME: Low-level details

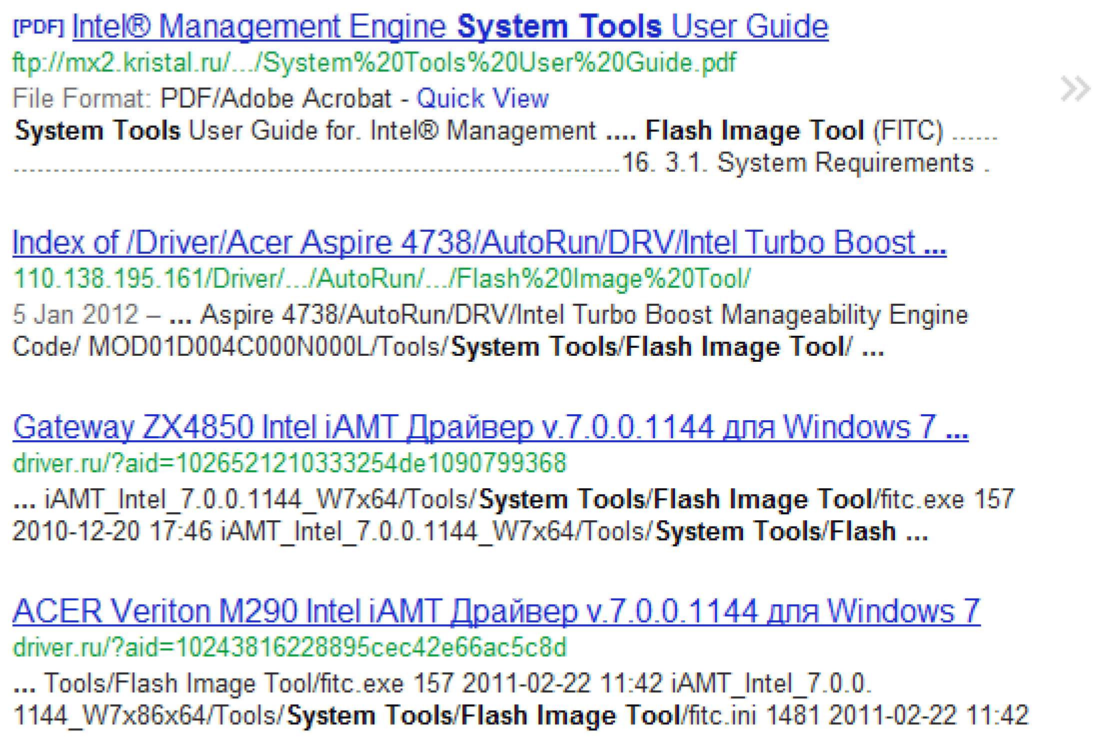
Sources of information

- 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

ME firmware kits

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**/ ...

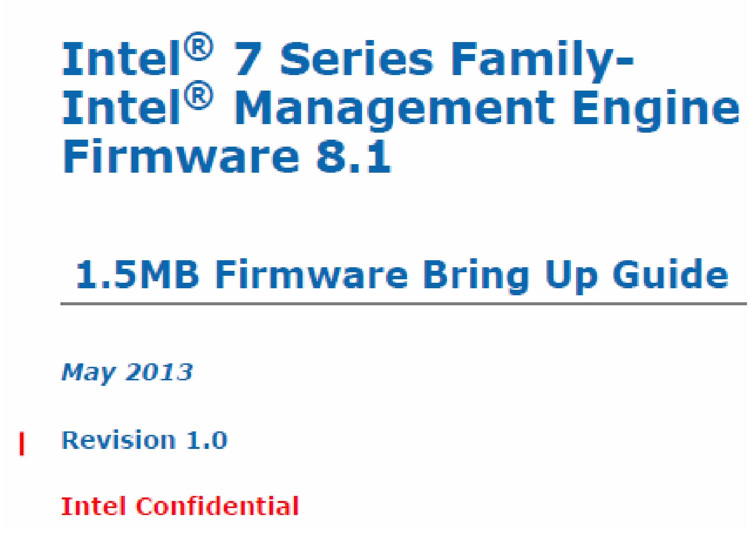
Gateway ZX4850 Intel iAMT Драйвер v.7.0.0.1144 для Windows 7 ...
driver.ru/?aid=1026521210333254de1090799368
... iAMT_Intel_7.0.0.1144_W7x64/Tools/**System Tools/Flash Image Tool**/fitc.exe 157 2010-12-20 17:46 iAMT_Intel_7.0.0.1144_W7x64/Tools/**System Tools/Flash** ...

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 :)

Intel FSP

- Intel Firmware Support Package was released in 2013
- Low-level initialization code from Intel for firmware writers
- Freely downloadable from Intel's site
- The package for HM76/QM77 includes 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

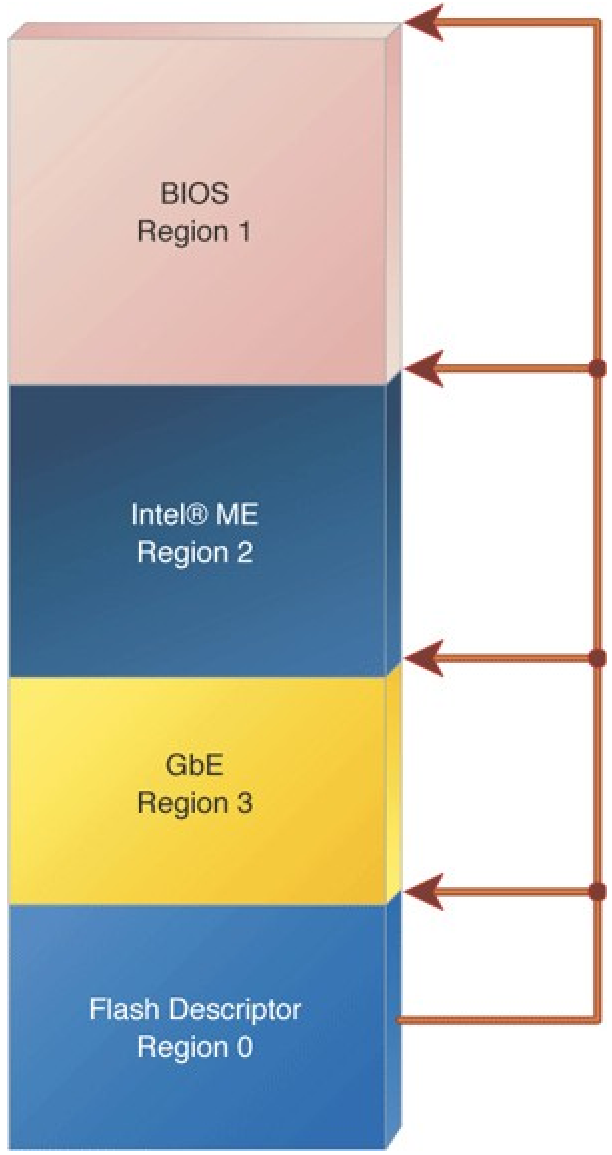
Intel Confidential

Documentation still contains "confidential" markings :)

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

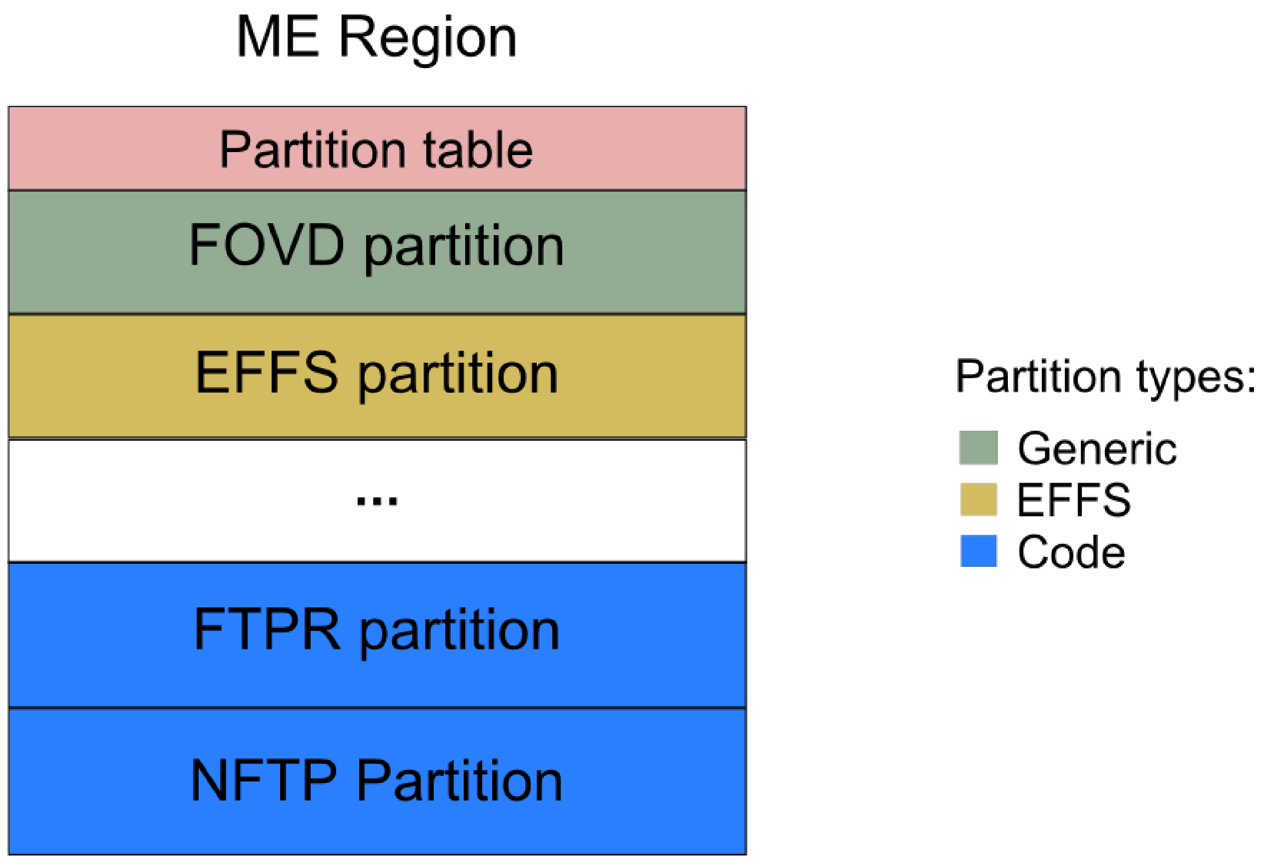
SPI flash layout

- 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.



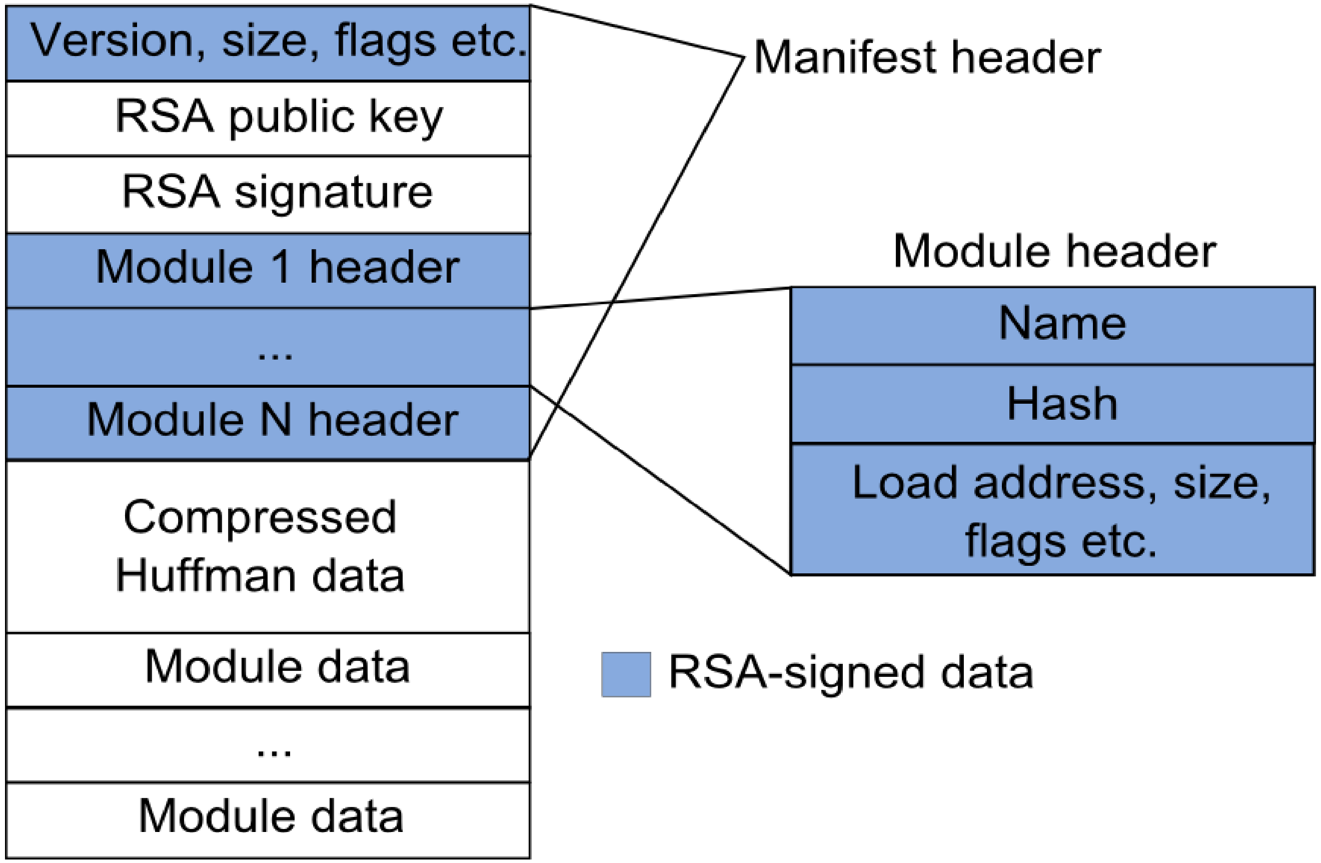
ME region layout

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



ME code partition

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



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

- Following discussion covers mostly Gen 2: Intel 5 Series (aka Ibex Peak) and later chipsets

ME code modules

Some common modules found in recent firmwares

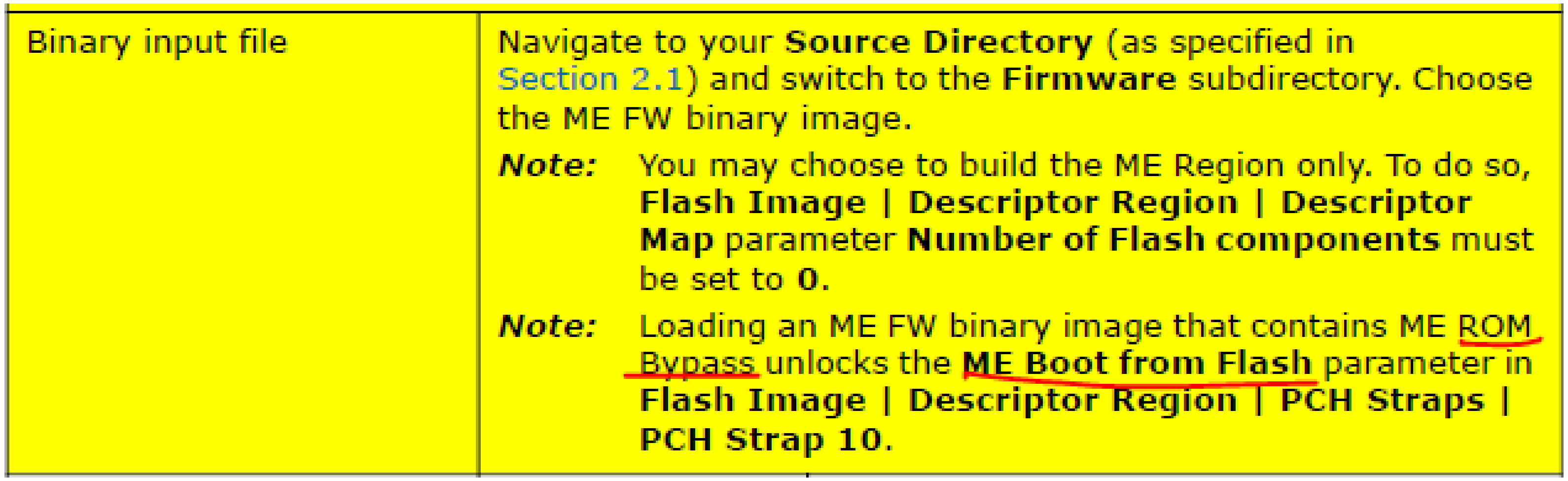
Module name	Description
BUP	Bringup (hardware initialization/configuration)
KERNEL	Scheduler, low-level APIs for other modules
POLICY	Secondary init tasks, some high-level APIs
HOSTCOMM	Handles high-level protocols over HECI/MEI
CLS	Capability Licensing Service – enable/disable features depending on 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

ME: code in ROM

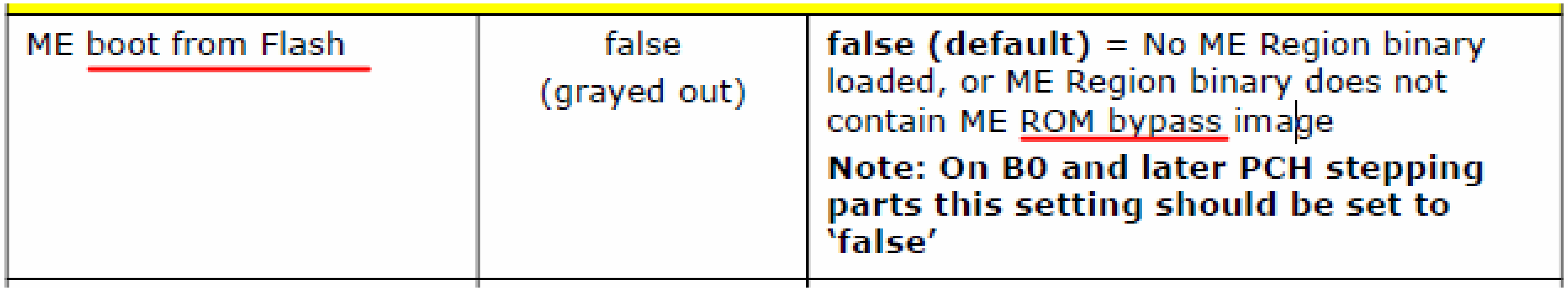
- To save flash space, various common routines are stored in the on-chip ROM and are not present in the firmware
- They are used in the firmware modules by jumping to hardcoded addresses
- This complicates reverse-engineering somewhat because a lot of code is missing
- However, one of the ME images I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

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



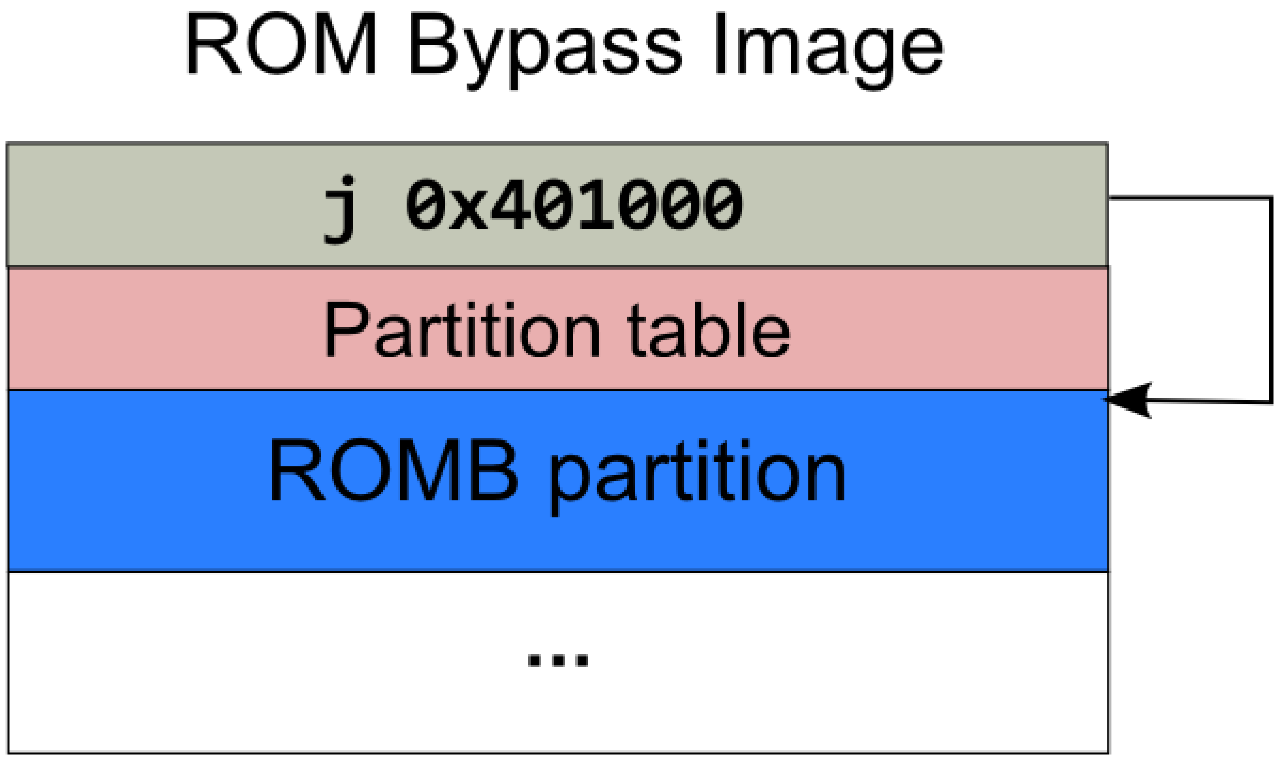
Binary input file	Navigate to your **Source Directory** (as specified in Section 2.1) and switch to the **Firmware** subdirectory. Choose the ME FW binary image. ***Note:*** You may choose to build the ME Region only. To do so, **Flash Image | Descriptor Region | Descriptor Map** parameter **Number of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



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

ME: ROM Bypass

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



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 :(

ME: Security and attacks

ME: 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

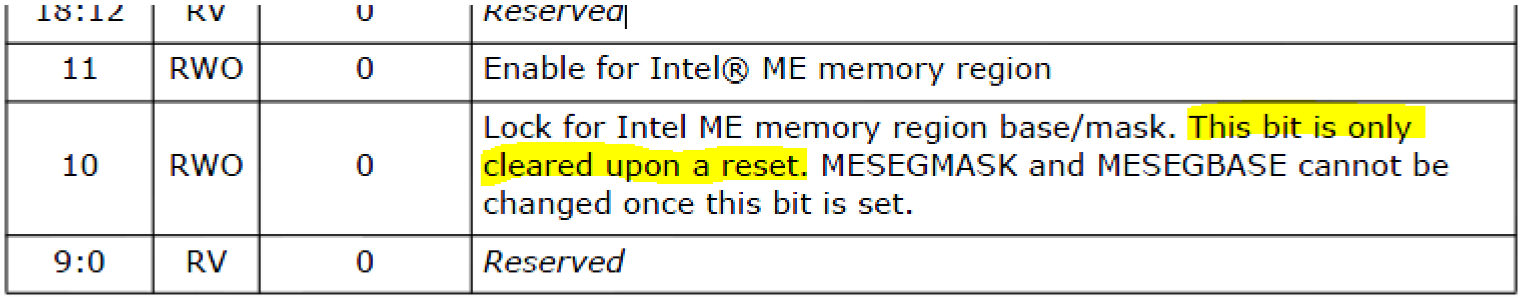
“During the design phase, a Firmware Signing Key (FWSK) public/private pair is generated at a secure Intel Location, using the Intel Code Signing System. The Private FWSK is stored securely and confidentially by Intel. Intel AMT ROM includes a SHA-1 Hash of the public key, based on RSA, 2048 bit modulus 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

ME: Unified Memory Architecture (UMA) region

- ME requires some RAM (UMA) 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	*Reserved*
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	*Reserved*

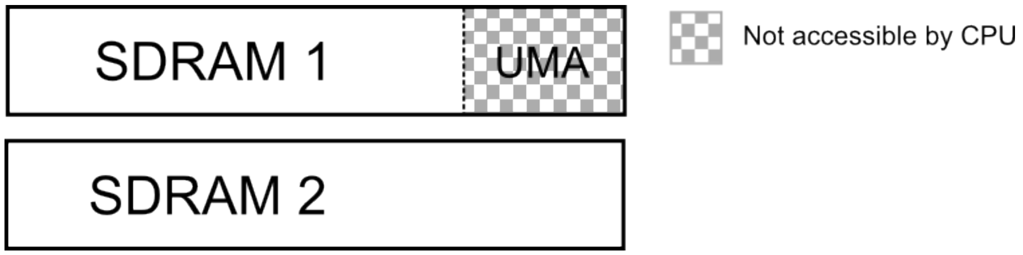
- A memory remapping attack was demonstrated by Invisible Things Lab in 2009, but it doesn't work on newer chipsets
- Cold boot attack might be possible, though...

ME: attacking UMA

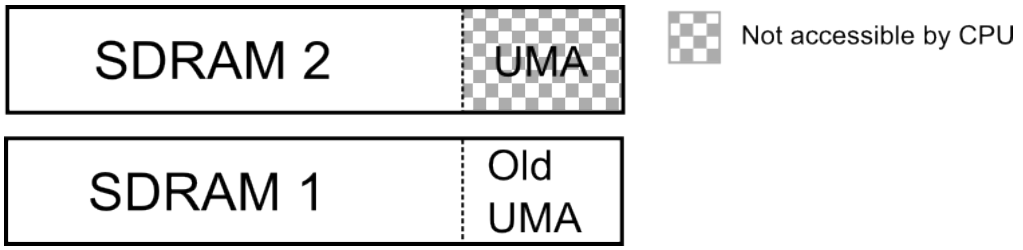
- I decided to try and dump the UMA region since it contains unpacked Huffman code and runtime data
- Idea #1: simply disable the code which sets the MESEG lock bit in the BIOS
- [some time spent reversing memory init routines...]
- Patched out the code which sets the lock bit
- Updated necessary checksums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

- Tried 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)

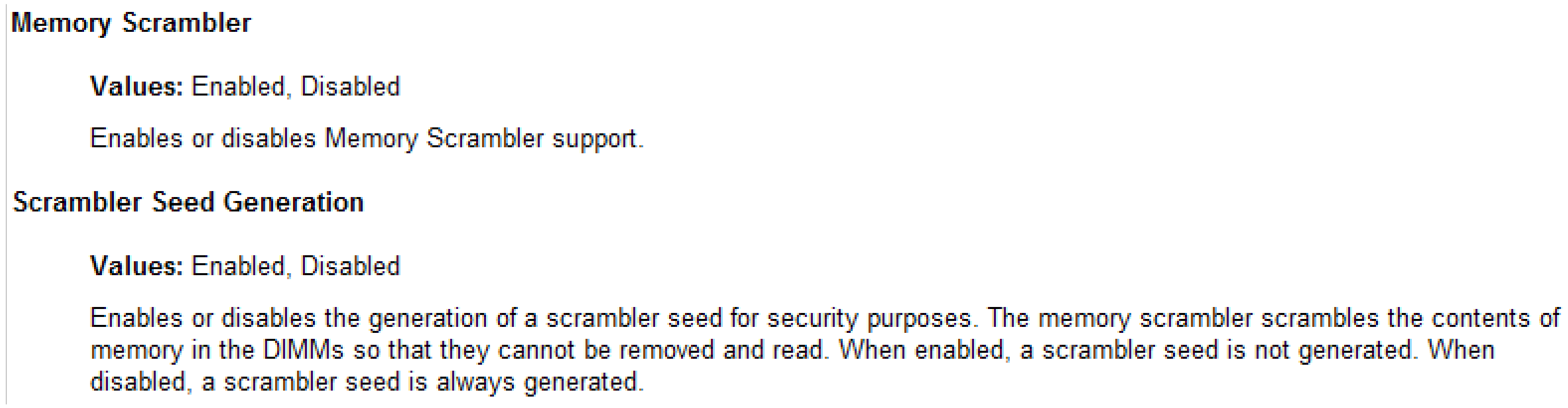
ME: attacking UMA

- 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
- Idea:

  1) Flash FPT that requests 32MB, reboot. BIOS will reserve top 32MB but ME will use only 16MB
  2) Flash FPT that requests 16MB, reboot. BIOS will reserve top 16MB, so previously used 16MB will be accessible again
- Unfortunately got garbage again. It seems that memory is reinitialized with different scrambling seed between 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" direclty (see my Breakpoint 2012 presentation)
- However, it is not enough – the memory is still garbage

ME: attacking UMA

- Idea #5: ?
- I still had some ideas to try but they require more time and effort
- So I started investigating code using other approaches
- For example...

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 can investigate how these two modules work
- There are differences from desktop but it's a start

JOM aka DAL

- JOM is a module which 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
- This feature is used to implement Intel 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's 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 JOM
- 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

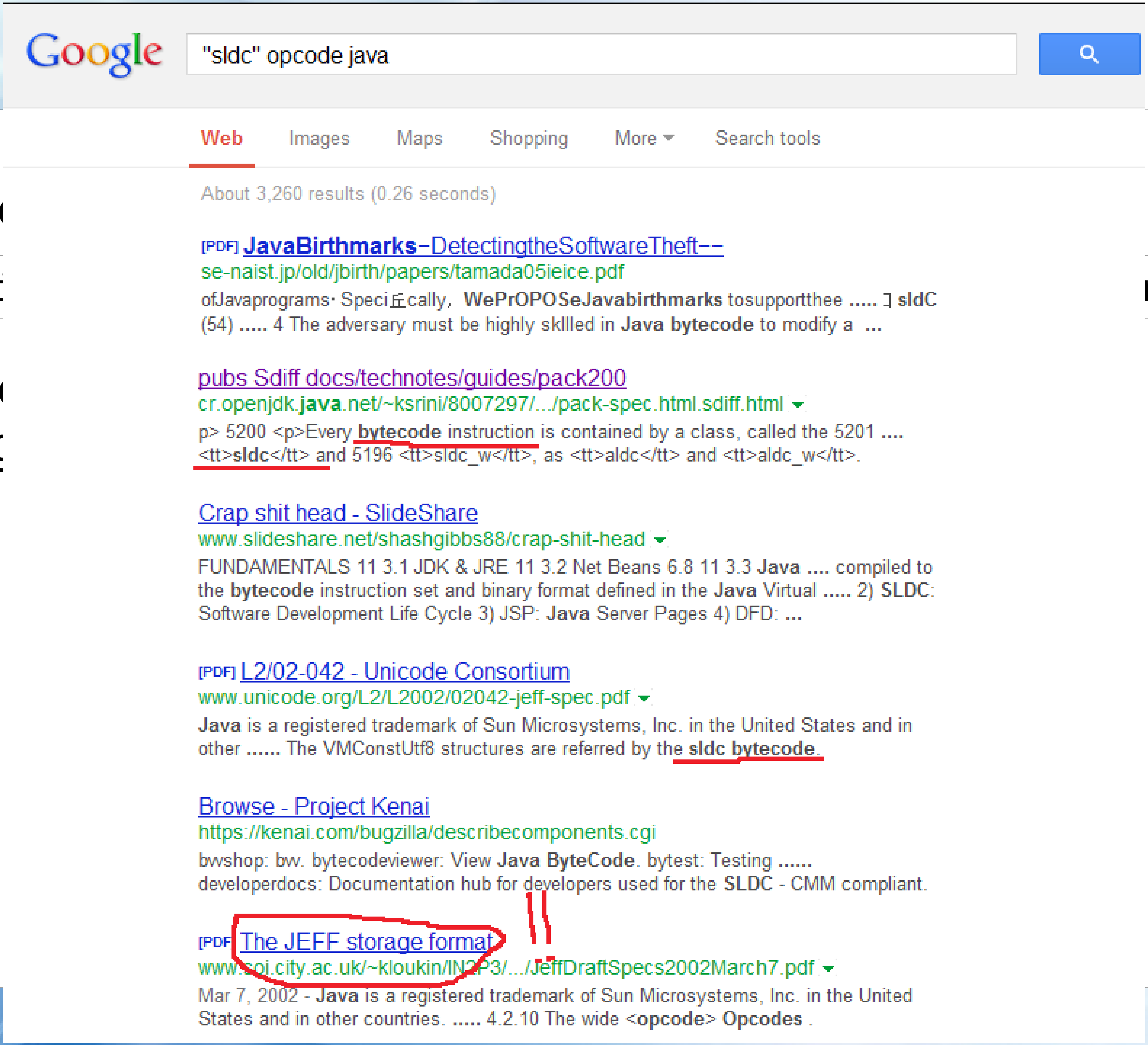
JOM aka DAL

- There was 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...

- There
  `.asci` `n"`
- There
  Goog



"sldc" opcode java

Web Images Maps Shopping More Search tools

About 3,260 results (0.26 seconds)

[PDF] JavaBirthmarks–DetectingtheSoftwareTheft––
se-naist.jp/old/jbirth/papers/tamada05ieice.pdf
ofJavaprograms· Speci丘cally, WePrOPOSeJavabirthmarks tosupportthee .....] sldC (54) ..... 4 The adversary must be highly sklIled 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.soi.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 .

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 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_invokeCommandInProcess;
  /* 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

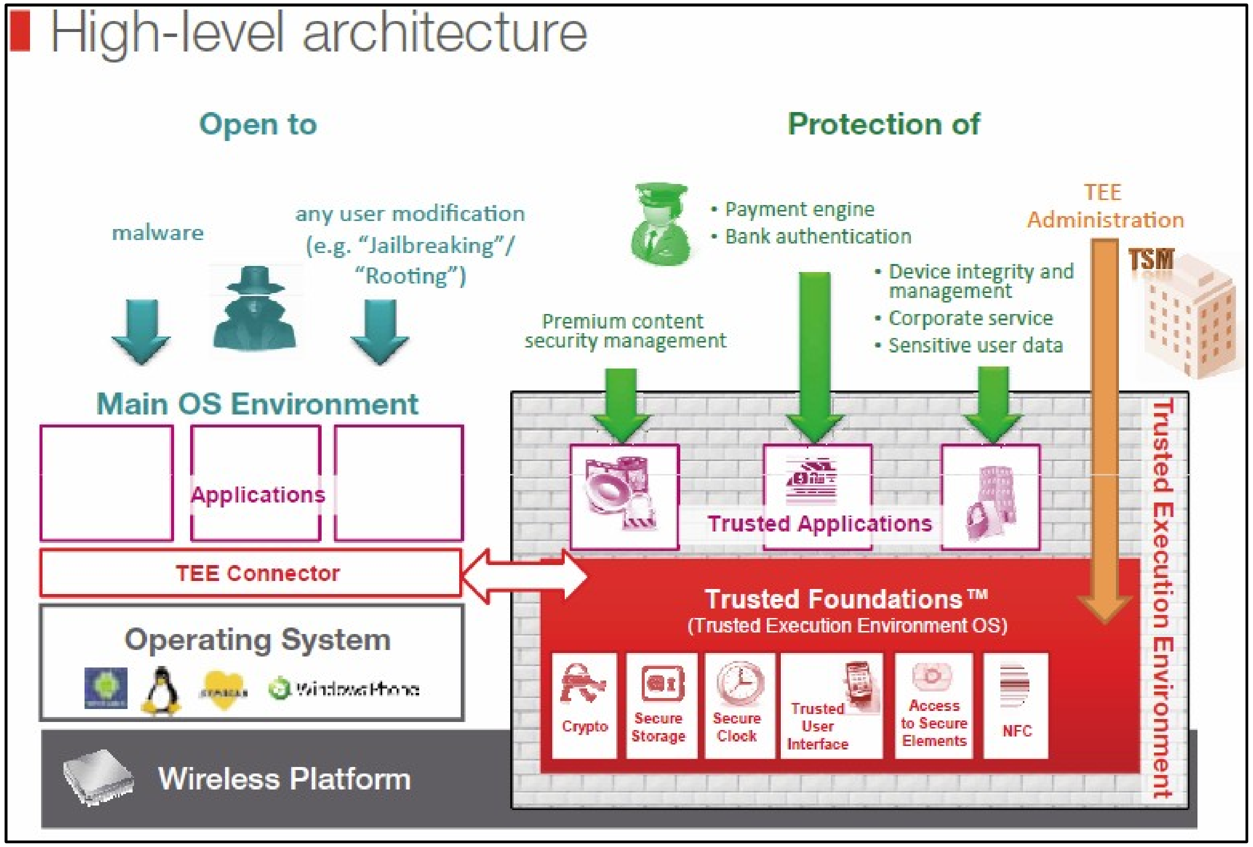
- Unfortunately, even if I create my own applets, I can't run them inside ME because...
- Applet binaries have a signed manifest header and are verified before running
- Still, there may be vulnerabilities in the protocol, which is pretty complicated
- Let's have a look at how it works...

IPT communication

- Intel provides several DLLs with high-level APIs which are usable from C/C++, Java, or .NET applications
- These DLLs send requests to the JHI service, using COM or TCP/IP (depending on the driver version)
- The service serializes requests and sends them over HECI/MEI to the ME
- ME dispatches the requests to JOM
- JOM parses the requests and passes them to the applet
- Reply undergoes the opposite conversion and is eventually sent back to the application
- Because arbitrary buffers can be sent and received, there is a potential for out-of-bounds memory read or write

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 Mobililty/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

Results

- 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
- The code of boot ROM, BUP and KERNEL modules has been discovered
- This allowed me to map many APIs used in other modules
- JEFF dumper is a good starting point for investigating DAL/IPT applets
- ARC support was released with IDA 6.4 and improved in IDA 6.5

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 newer firmwares for compressing the kernel and some other modules
  - Apparently the dictionary is hardcoded in silicon
  - There is 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 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

- Bay Trail/TXE
    - In Bay Trail (Atom-based SoC), a new variation of ME is used
    - Called Trusted Execution Engine (TXE), codename SEC
    - Instead of ARC, uses SPARC core(!)
    - No Huffman compression, only LZMA(!!)
    - So, all code (except Boot ROM) is available
    - Available KERNEL code should help recovering APIs for ARC firmwares too
    - SPARC emulators are available so the code can be emulated/fuzzed/debugged

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