coreboot

16 Gennaio 2017

NECSTLab

Me

Federico Izzo

federico.izzo42@gmail.com

github.com/Nimayer

A thanks to Nicola Corna

Who introduced me to coreboot and did the great part of the work on Intel ME

nicola@corna.info

github.com/corna

Index

- **What is coreboot?**
- **How do I install it?**
- **Intel ME**

What is coreboot?

coreboot is a project meant to replace the *proprietary firmware* (BIOS or UEFI) present in most computers

We could say that coreboot is an *open source BIOS*

However coreboot is not a proper BIOS

- A **BIOS** firmware:
  - performs hardware initialization
  - provides runtime calls for the OS
- **coreboot** does just the hardware initialization

Modern Windows versions and Linux don't use BIOS calls anymore

- You can still run DOS using SeaBIOS on coreboot

Benefits

- FOSS software
  - Safer
  - Hackable
  - BIOS backdoor free
- Very fast! (0.5/1 s from off to Linux kernel boot)
- Written almost completely in 32-bit C language
  - Unlike commercial BIOSes that are written in 16-bit assembler
- Follows the rule "*initialize the hardware, then get out of the way*"

Downsides

- Few hardware supported
- Complex compilation
- Hard to install
- New CPU generations make development and installation harder
  - Intel Boot Guard

How does it work?

coreboot code is split in four main stages:

- Bootblock
- Romstage
- Ramstage
- Payload

Bootblock

In this stage coreboot:

- Reads CMOS configuration
- Decides in which mode to start (*Normal* or *Fallback*)

Romstage

This is the most critical stage, here coreboot initializes RAM memory and Intel ME.

- Initializes debugging peripherals
- Initializes the chipset
- Configures the memory
- Allocates the shared memory Intel ME requires

Ramstage

During this stage coreboot initializes the remaining peripherals and then jumps into the payload.

After this stage coreboot has done its work and won't execute any code until **suspension** or **shutdown**.

Payloads

Now that the hardware is initialized we can let another software continue the boot process.

The most interesting payloads are:

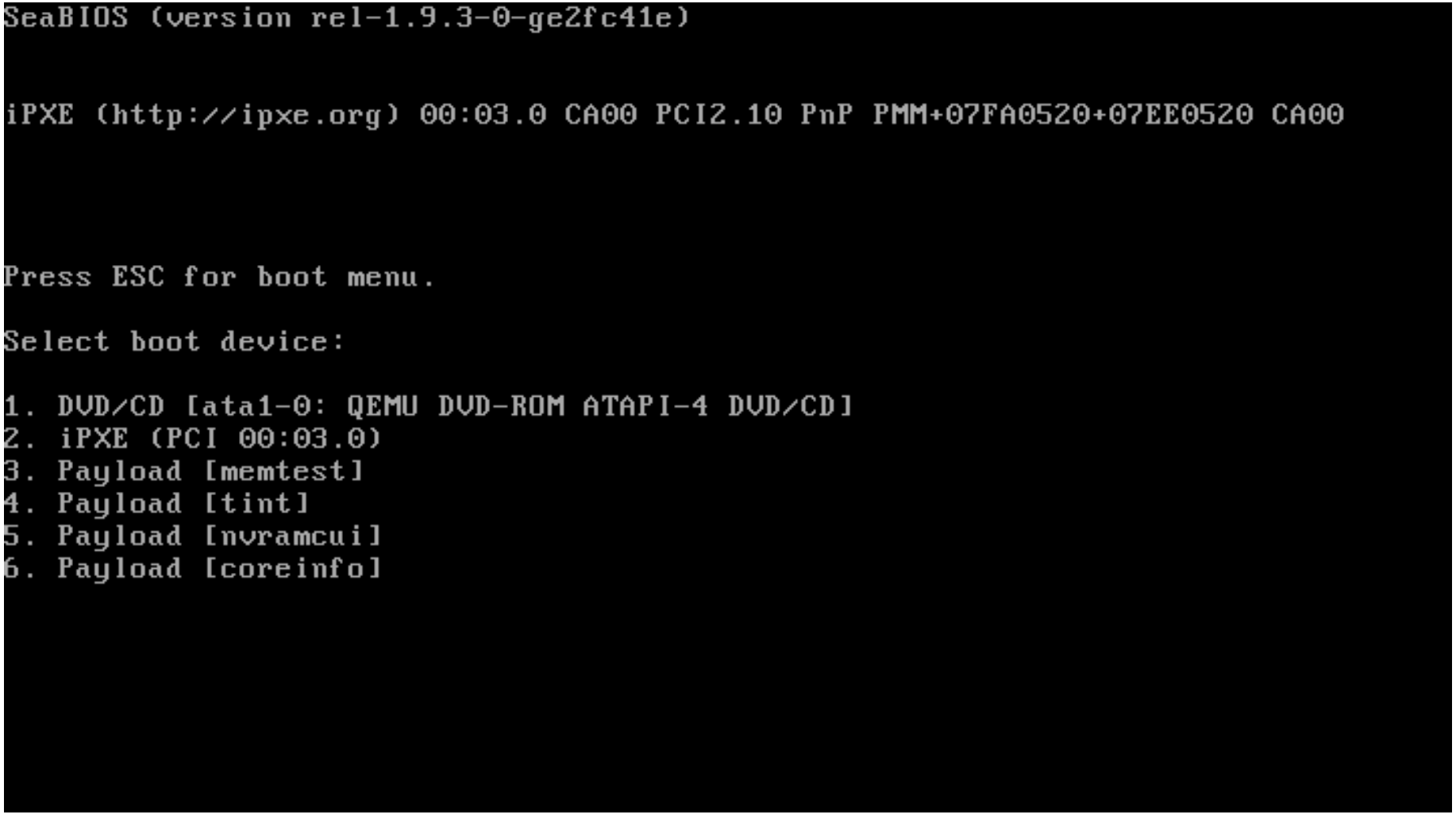
- SeaBIOS
- Tianocore (UEFI)
- GRUB
- Linux

Payloads

There are also **secondary payloads** that can be booted:

- nvramcui: *configuration utility*
- coreinfo: *information dump*
- Memtest86+: *memory test*
- Tint: *tetris*
- GRUB invaders: *you get the idea*

SeaBIOS

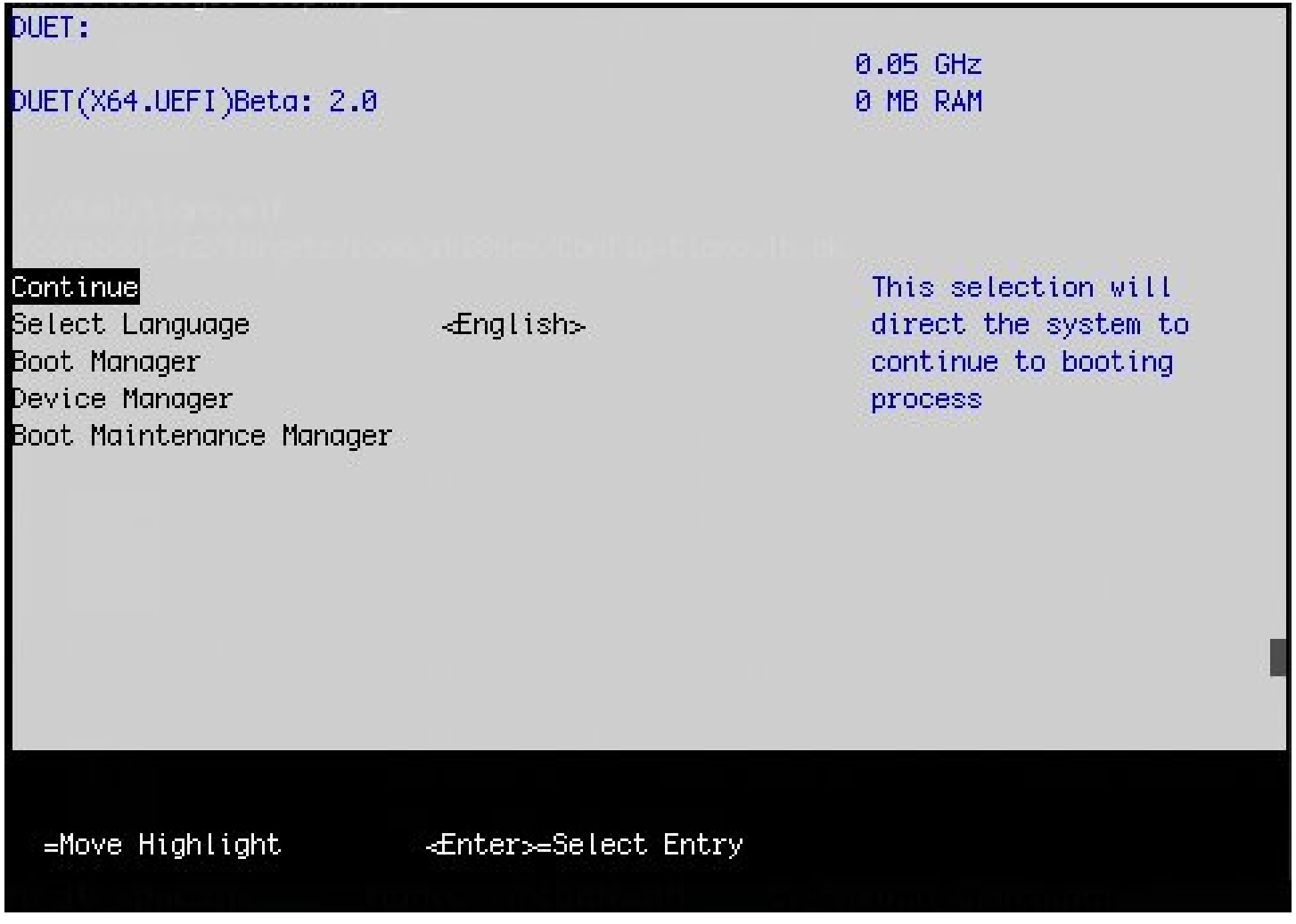

SeaBIOS (version rel-1.9.3-0-ge2fc41e)
iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07FA0520+07EE0520 CA00
Press ESC for boot menu.
Select boot device:
1. DVD/CD [ata1-0: QEMU DVD-ROM ATAPI-4 DVD/CD]
2. iPXE (PCI 00:03.0)
3. Payload [memtest]
4. Payload [tint]
5. Payload [nvramcui]
6. Payload [coreinfo]

SeaBIOS

A complete x86 BIOS implementation.

coreboot + SeaBIOS provides you a *complete BIOS system*, good starting point for a coreboot setup.

Tianocore


DUET:
0.05 GHz
DUET(X64.UEFI)Beta: 2.0
0 MB RAM
Continue
Select Language
<English>
Boot Manager
Device Manager
Boot Maintenance Manager
This selection will direct the system to continue to booting process
=Move Highlight
<Enter>=Select Entry

Tianocore

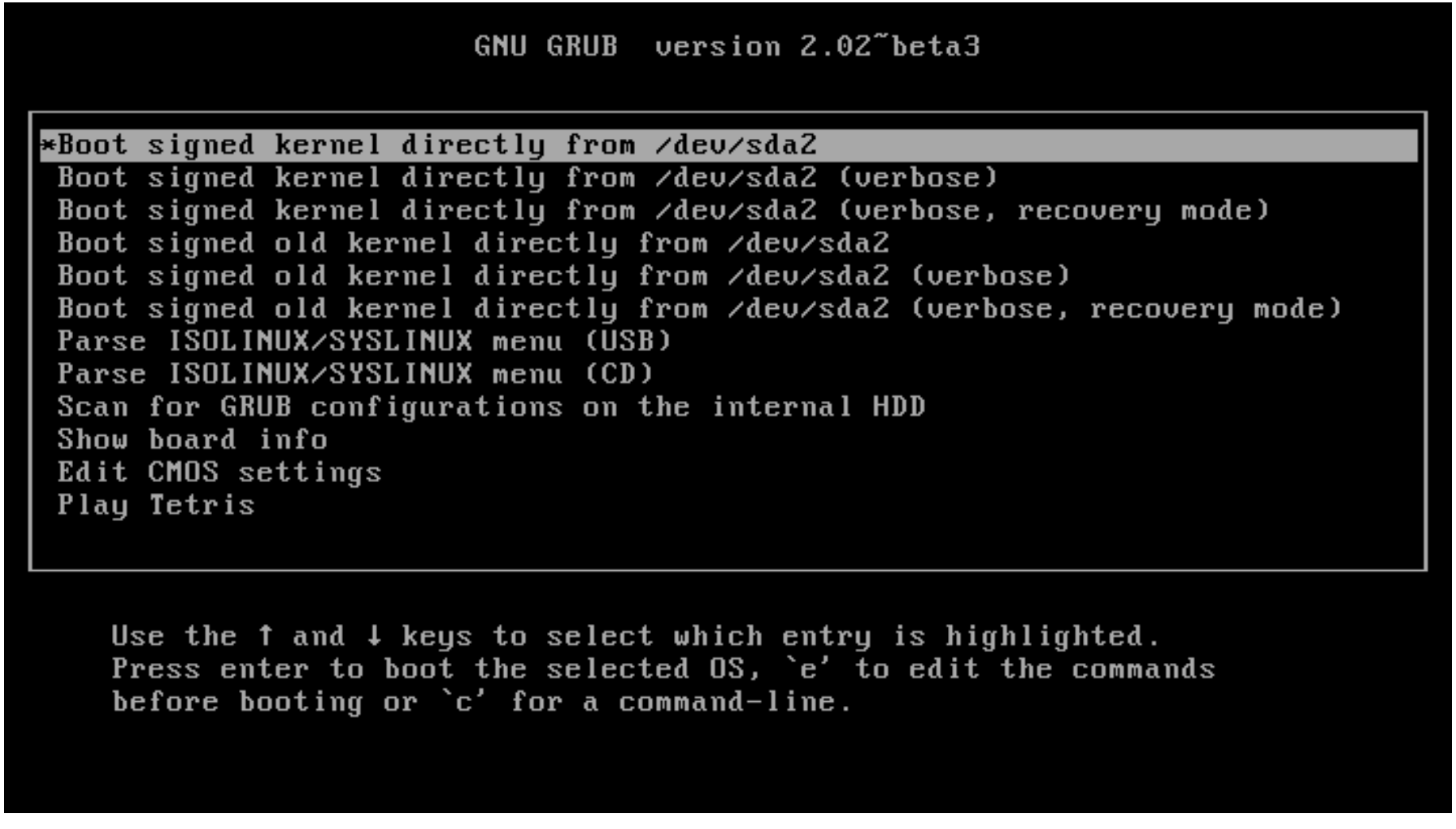
Tianocore is Intel's UEFI *reference implementation*, released under open source licenses.

Duet is part of Tianocore, it should give you *UEFI support on coreboot*

if you are able to make it work, I failed.

Tianocore can also include SeaBIOS as CSM, to get an UEFI + BIOS system.

GRUB


GNU GRUB version 2.02~beta3
*Boot signed kernel directly from /dev/sda2
Boot signed kernel directly from /dev/sda2 (verbose)
Boot signed kernel directly from /dev/sda2 (verbose, recovery mode)
Boot signed old kernel directly from /dev/sda2
Boot signed old kernel directly from /dev/sda2 (verbose)
Boot signed old kernel directly from /dev/sda2 (verbose, recovery mode)
Parse ISOLINUX/SYSLINUX menu (USB)
Parse ISOLINUX/SYSLINUX menu (CD)
Scan for GRUB configurations on the internal HDD
Show board info
Edit CMOS settings
Play Tetris
Use the ↑ and ↓ keys to select which entry is highlighted.
Press enter to boot the selected OS, `e' to edit the commands
before booting or `c' for a command-line.

GRUB

You already know GRUB.

Probably you don't know that GRUB can be run directly from coreboot, without a BIOS.

This is due to the fact that Linux does not use BIOS legacy calls.

GRUB

It has some advantages with respect to SeaBIOS:

- Faster
- Has less code
- Built-in crypto
  - Can unlock LUKS volumes
  - Can verify kernel/initramfs signatures

Linux

```
[    4.040228] hub 4-0:1.0: 5 ports detected
[    4.040627] Initializing USB Mass Storage driver...
[    4.040815] usbcore: registered new interface driver usb-storage
[    4.040821] USB Mass Storage support registered.
[    4.040989] usbcore: registered new interface driver libusual
[    4.041313] mice: PS/2 mouse device common for all mice
[    4.041451] usbcore: registered new interface driver xpad
[    4.041456] xpad: X-Box pad driver
[    4.044607] usbcore: registered new interface driver hiddev
[    4.044729] usbcore: registered new interface driver usbhid
[    4.044734] usbhid: USB HID core driver
[    4.045228] snd_xenon: iobase_phys=0x200ea001600 iobase_virt=0xd000080081074600
[    4.045262] xenon_snd: give me an interrupt, please!
[    4.045271] snd_xenon: irq=40
[    4.045292] snd_xenon: descr_base_virt=0xc00000001836e000, descr_base_phys=0x1836e000
[    4.045510] snd_xenon: driver initialized
[    4.046474] ALSA device list:
[    4.046479]   #0: Xenon AudioPCI at 0x200ea001600 irq 64
[    4.046546] TCP cubic registered
[    4.046555] NET: Registered protocol family 17
[    4.046621] Registering the dns_resolver key type
[    4.417774] ata2.00: 488397168 sectors, multi 16: LBA48 NCQ (depth 0/32)
[    4.441602] ata2.00: configured for UDMA/133
[    4.456622] scsi 1:0:0:0: Direct-Access     ATA      ST9250315AS      0002 PQ: 0 ANSI: 5
[    4.476138] sd 1:0:0:0: [sda] 488397168 512-byte logical blocks: (250 GB/232 GiB)
[    4.476733] sd 1:0:0:0: Attached scsi generic sg1 type 0
[    4.510622] sd 1:0:0:0: [sda] Write Protect is off
[    4.525934] sd 1:0:0:0: [sda] Mode Sense: 00 3a 00 00
[    4.541440] sd 1:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
[    4.592956] usb 2-2: new high speed USB device using ehci_hcd and address 3
[    4.593545]  sda: unknown partition table
[    4.626206] sd 1:0:0:0: [sda] Attached SCSI disk
[    4.748153] scsi2 : usb-storage 2-2:1.0
[    4.869613] usb 2-3: new high speed USB device using ehci_hcd and address 4
[    5.359608] usb 4-1: new low speed USB device using ohci_hcd and address 2
[    5.545259] input: SINO WEALTH USB KEYBOARD as /devices/pci0000:00/0000:00:05.0/usb4/4-1/4-1:1.0/input/input0
[    5.566939] generic-usb 0003:258A:0001.0001: input,hidraw0: USB HID v1.10 Keyboard [SINO WEALTH USB KEYBOARD] on usb-0000:00:05.0-1/input0
[    5.569613] Sending DHCP requests .
[    5.610876] input: SINO WEALTH USB KEYBOARD as /devices/pci0000:00/0000:00:05.0/usb4/4-1/4-1:1.1/input/input1
```

Linux

coreboot can boot directly a Linux Kernel from the onboard ROM.

Has some drawbacks: you need to flash again the ROM each time you want to update the kernel or even change the cmdline.

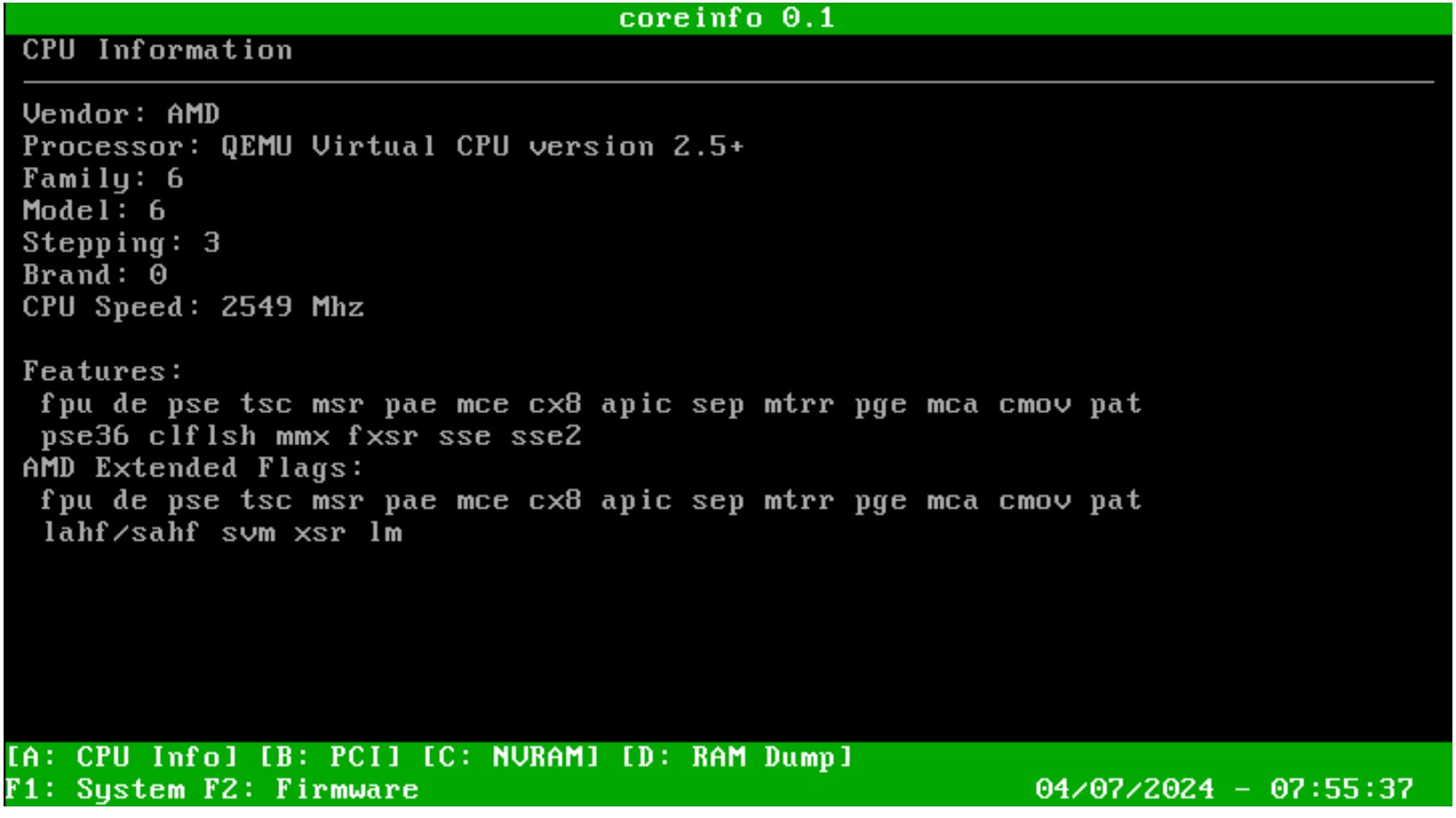
It gives you even more flexibility than GRUB,

For example look at the HEADS bootloader which uses tpm for firmware and filesystem *measurement*.

nvramcui

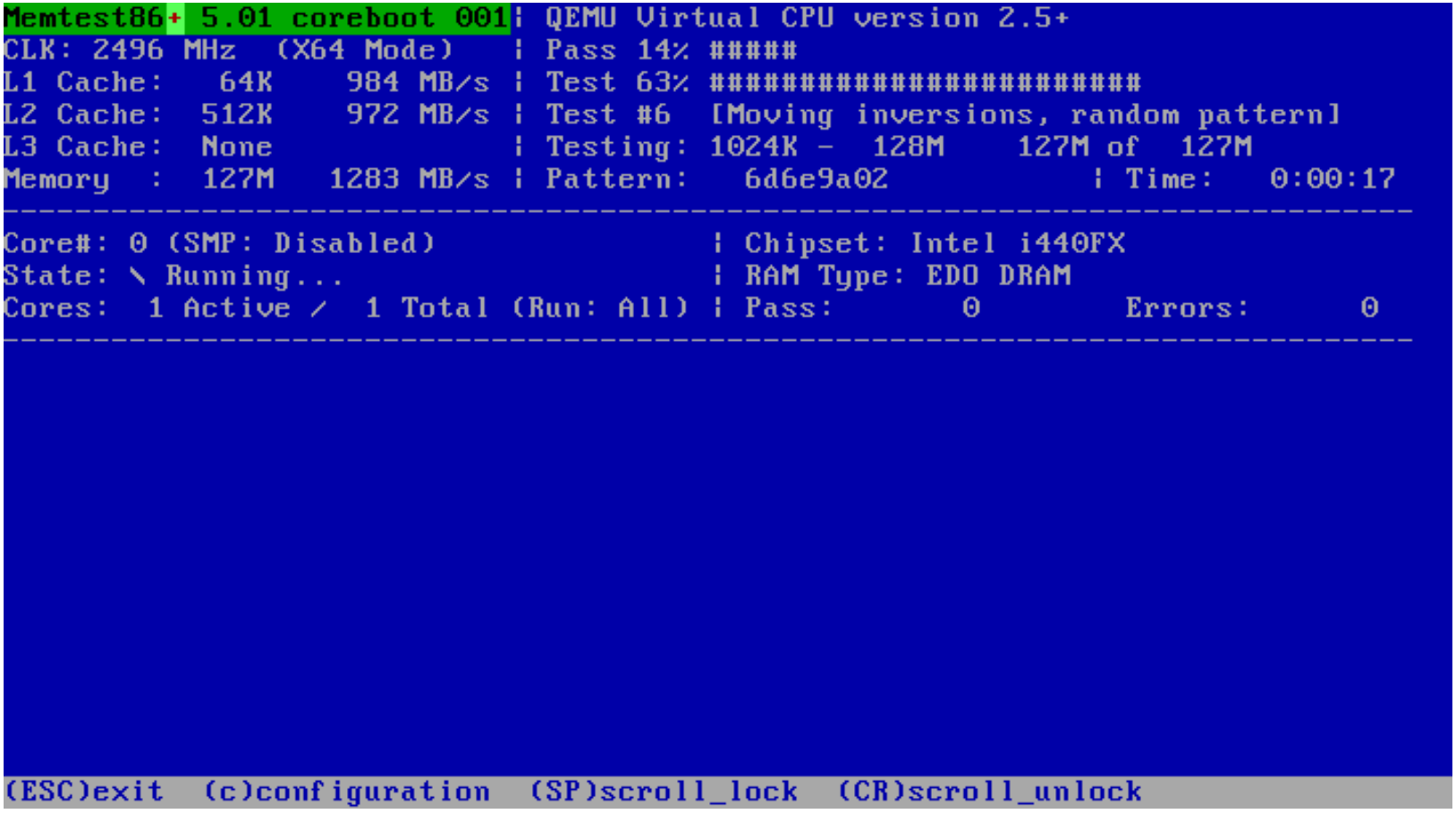
An utility to change CMOS configuration.

coreinfo



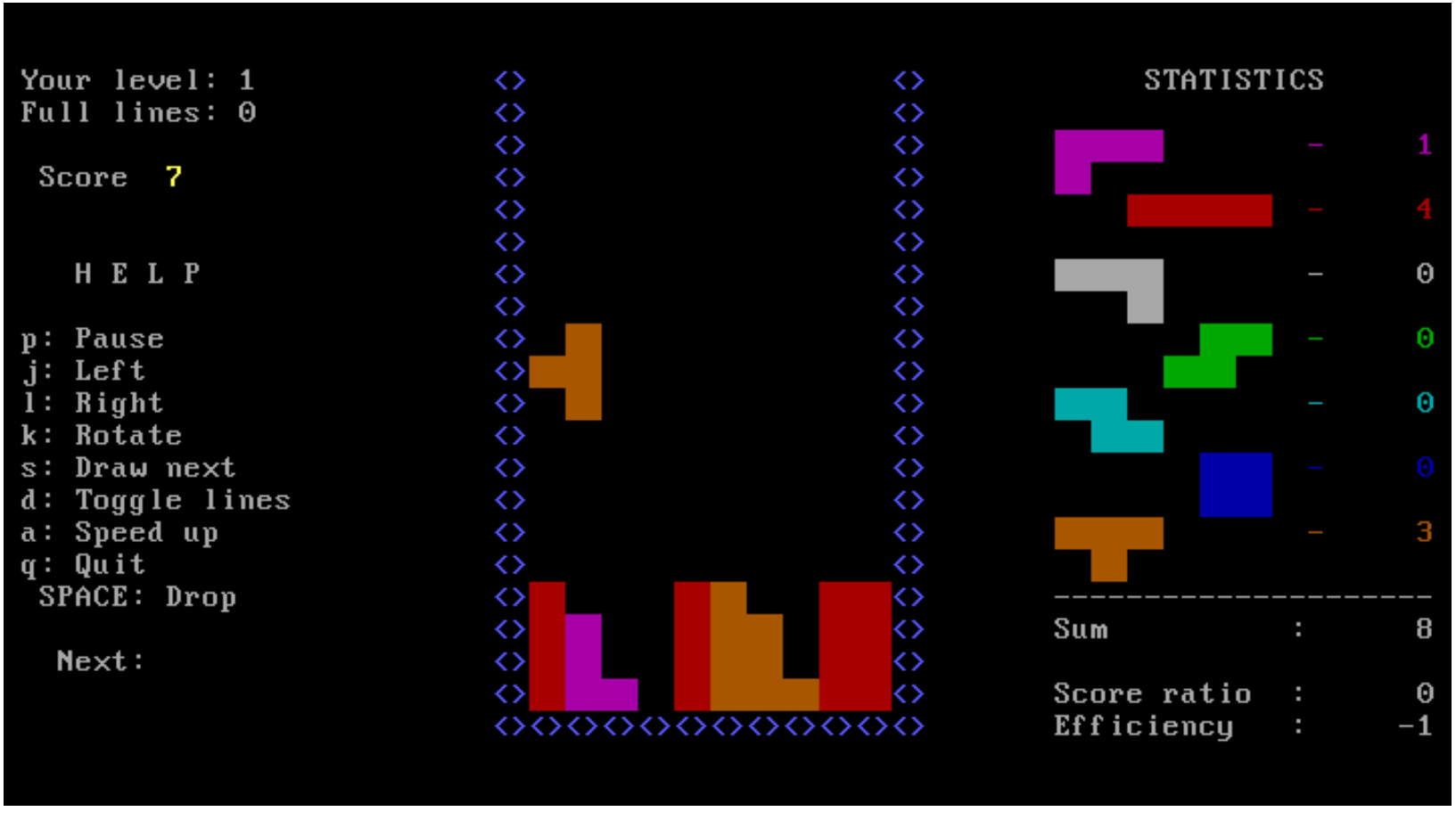
An utility to view system info.

Memtest86+



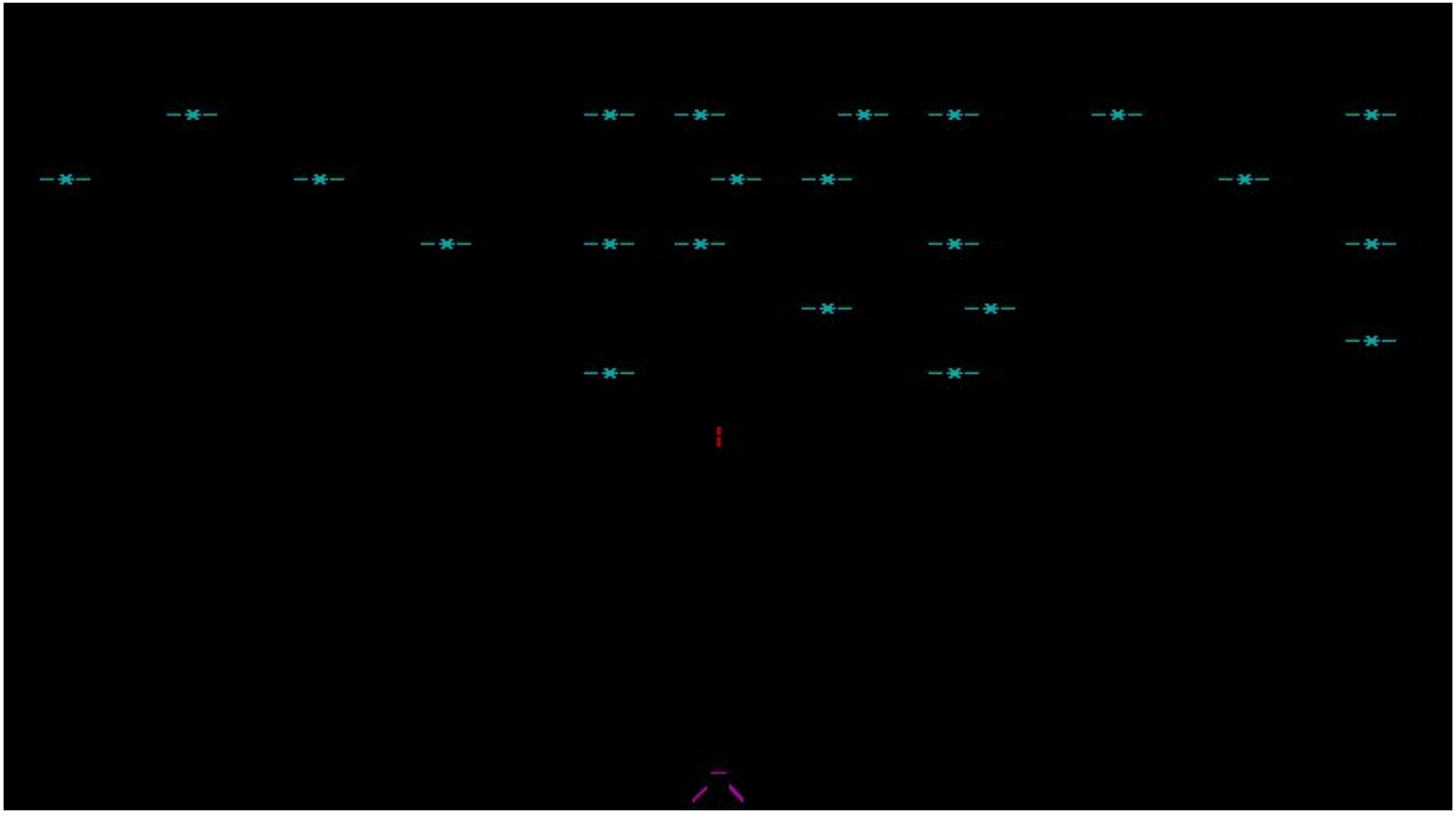
A tool to check the RAM health.

TinT (Tint is not Tetris)



TETRIS!!!

GRUB invaders



Space invaders!!!

coreboot: how do I install it?

The installation is divided into four steps:

- Prepare the building environment
- Dump your original BIOS
- Compile coreboot
- Flash the coreboot image

The building environment

here you can find the official guide, that follows a questionable order.

What you have to do is:

- Clone the coreboot repository

```
$ git clone --recursive http://review.coreboot.org/p/coreboot
$ cd coreboot
```

- Compile the *cross-compiler*, coreboot runs in 32bit mode

```
make crossgcc-i386 CPUS=4
```

- Configure coreboot

```
make menuconfig
```

Try it with QEMU!

It is possible to try coreboot+payload on QEMU before messing with the hardware

Do `make menuconfig` to configure coreboot

check that the *Mainboard* menu looks like this:

- vendor: Emulation
- model: QEMU x86 q35/ich9

Leave the menuconfig and do `make -jN` to compile

The `coreboot.rom` file inside the `build` subfolder is your image

You can run QEMU using

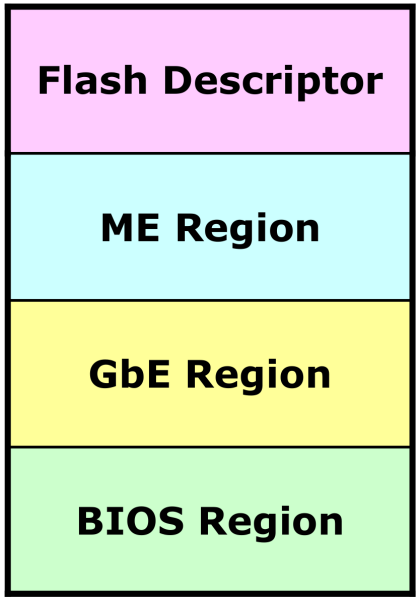
```
qemu-system-x86_64 -M q35 -bios build/coreboot.rom
```

To build an image for your laptop

you will need a *dump* of the flash content, to extract:

- Intel Flash Descriptor
- Intel ME Firmware
- Gigabit Ethernet Firmware
- Intel GPU VBIOS (optional)

What there is inside an Intel PC flash:



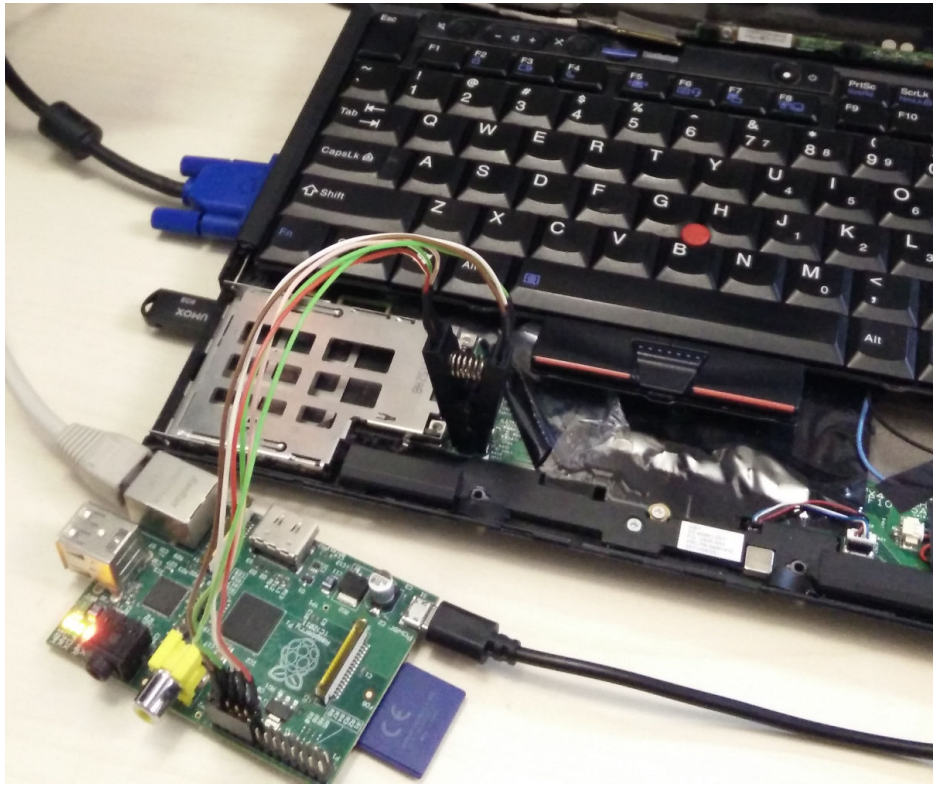
The Intel ME region is accessible **only by ME itself**, also, the BIOS region can be **write-protected**.

However it is possible to read or write the entire flash by connecting an external programmer to the flash chip.

Dumping the hard way

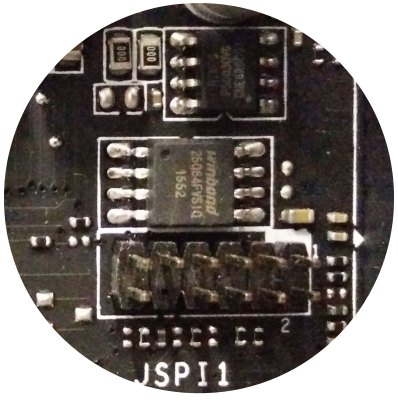
The flash chip uses the **SPI** protocol,

So we can read its content using the SPI interface of a Raspberry Pi or a similar board with 3.3V GPIO



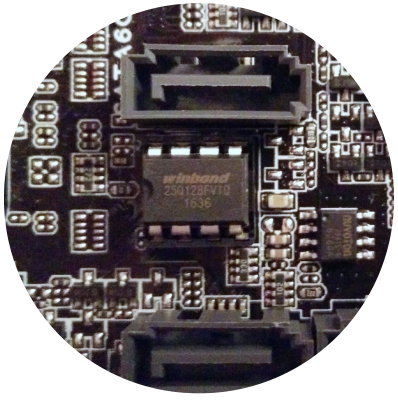
Find the flash

SOIC-8



SOIC-16

DIP-8



PLCC-32

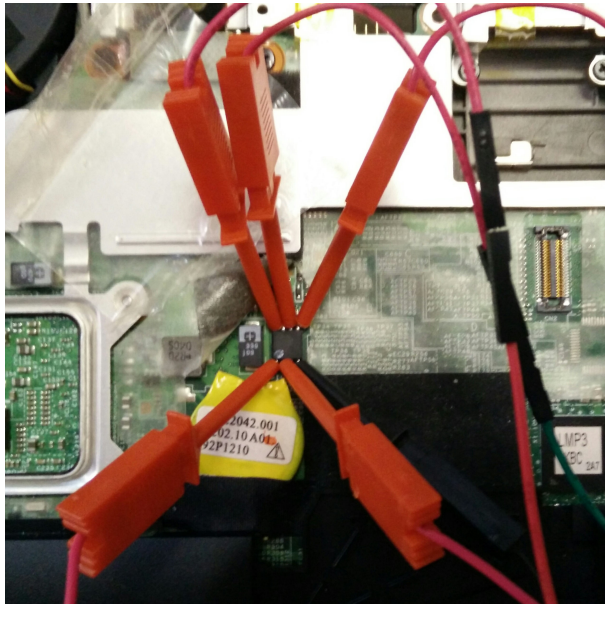
Clips!

You can find the flash chip **pinout** inside its datasheet

You can use these to connect the chip

SOIC-8 testclip

SMD clips

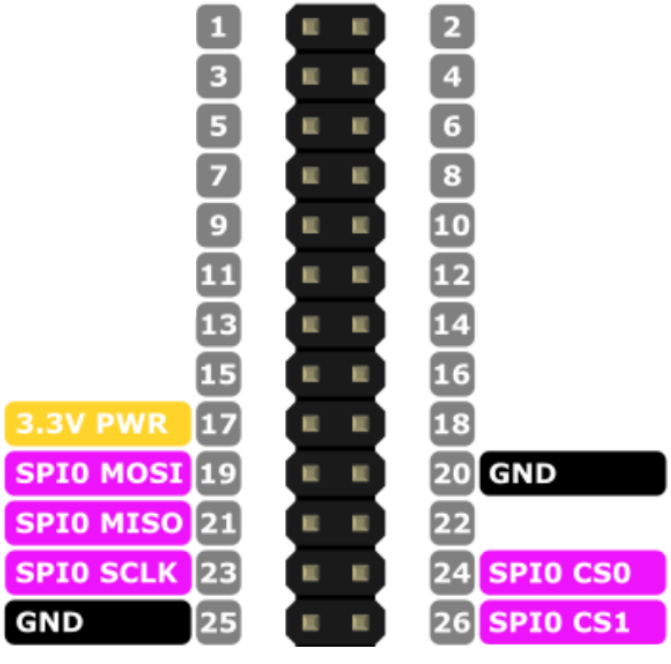


I found the SMD clips more reliable

Connect the wires

First of all **unplug your charger and remove the battery**

Raspberry Pi pins



to be connected in this order

RPi	**Flash**
GND	GND
CS0	CS
SPI0 SCLK	CLK
3.3V PWR	3.3V
SPI0 MISO	MISO
SPI0 MOSI	MOSI

Flashrom

Compile **flashrom** from the github repo or install it from your package manager

The Raspberry Pi command is:

```
flashrom -p linux_spi:dev=/dev/spidev0.0 -r dump.bin
```

Flashrom may ask you to specify your chip model if he cannot detect it automatically, you can use the option `-c <chipname>`

(e.g. on a Thinkpad X220 the option would be `-c W25Q64.V`)

A good practice is to make two dumps and compare the results (using `diff`) to be more safe

Extract the blobs

The utility `ifdtool` included in the coreboot tree can be used to extract our dump

- Compile the utility

```
cd coreboot/util/ifdtool
make
```

- Extract the flash regions

```
mkdir extracted_dump
cp dump.bin extracted_dump/
./util/ifdtool/ifdtool -x extracted_dump/dump.bin
```

- You will find the extracted flash regions in the folder:
  - BIOS
  - ME blob
  - GbE blob
  - Flash Descriptor

Configuration

coreboot uses a Linux kernel like configuration

Use `make menuconfig` to open the configuration tool and the **help** button to get a description of the elements.

I will show you a standard configuration, it's up to you to try the other settings (hint: normal/fallback)

Configuring coreboot pt.I

The main options to set are:

- Mainboard
  - Mainboard vendor: *your computer brand*
  - Mainboard model: *your computer model*
  - Rom chip size: *the flash chip size*
- Chipset
  - Include microcode in CBFS: *Generate from tree*
  - Add Intel descriptor.bin file: *we extracted it before*
  - Add Intel ME/TXT firmware: *same thing*
  - Add gigabit ethernet firmware: *same thing*

Configuring coreboot pt.II

- Devices
  - Use native graphics initialization: *usually works*
  - Enable PCIe Clock Power Management: *good idea*
- Display
  - Keep VESA framebuffer: *graphical mode instead of text*
- Generic Drivers
  - Enable TPM support
- Payload
  - Add a payload: *SeaBIOS or one of your choice*
  - Secondary Payloads: *see here*

Compiling

To compile run `make -jN`

The resulting image will be in `coreboot/build/coreboot.rom`

Flashing coreboot

To flash the image the first time we need to use the SPI connection, as we did for the *dump*

From the next time we can flash directly **from linux** because in coreboot the write protection of the BIOS/ME blob is optional

The command to flash using a Raspberry Pi is:

```
flashrom -c <chipname> -p linux_spi:dev=/dev/spidev0.0 -w coreboot.rom
```

force_I_want_a_brick

Once you have booted Linux, you can update coreboot using:

```
flashrom -c <chipname> -p internal:laptop=force_I_want_a_brick -w coreboot.rom
```

After updating coreboot, the best thing is to **turn off completely your computer** in order to run the newly flashed BIOS/ME blob

Intel ME

Intel ME

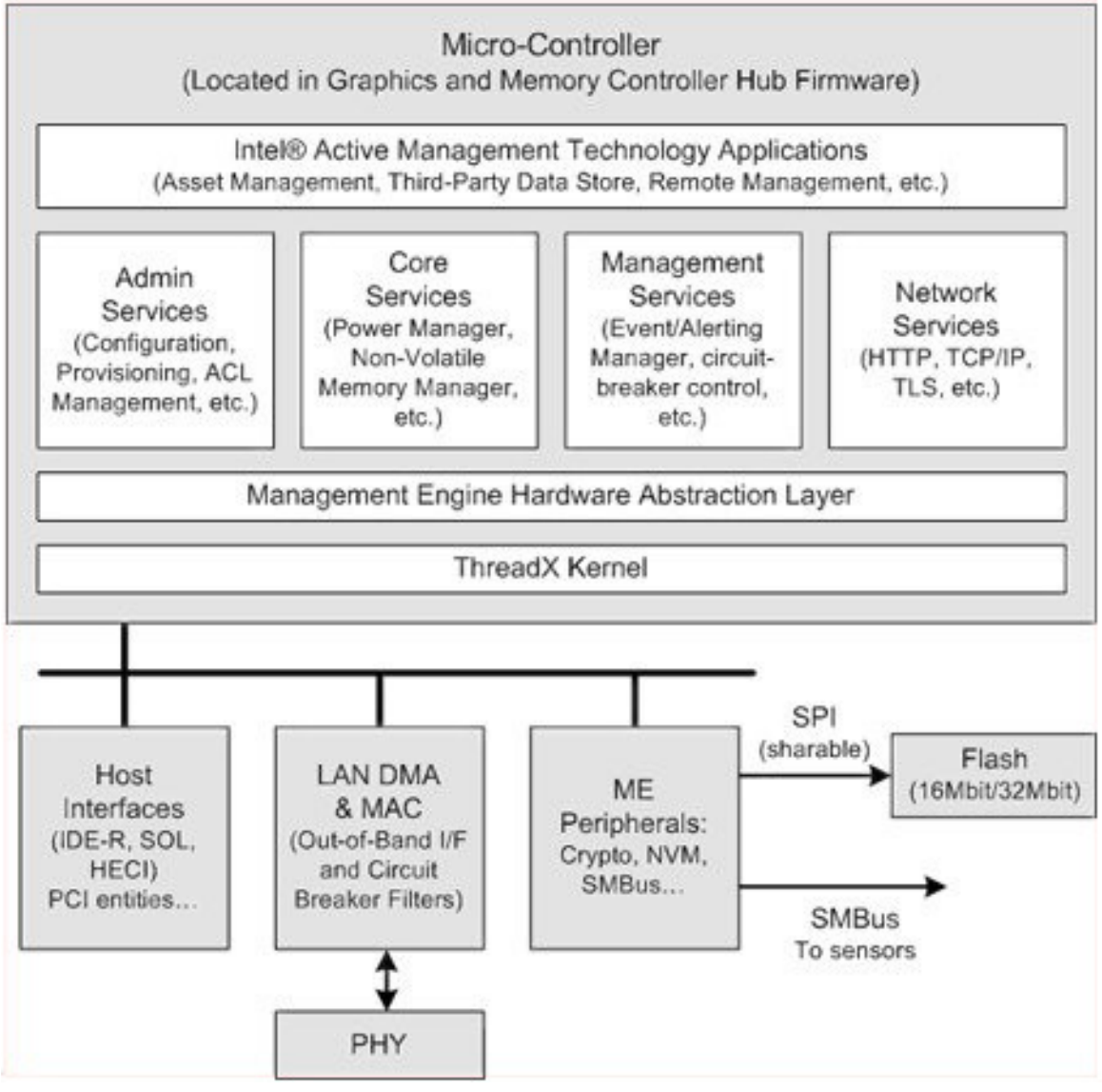
Intel Management Engine is a **secondary processor** integrated in all Intel motherboard chipsets from 2008 onwards.

It is mainly used for Intel AMT (Advanced Management Technology) on CPUs with vPRO enabled.

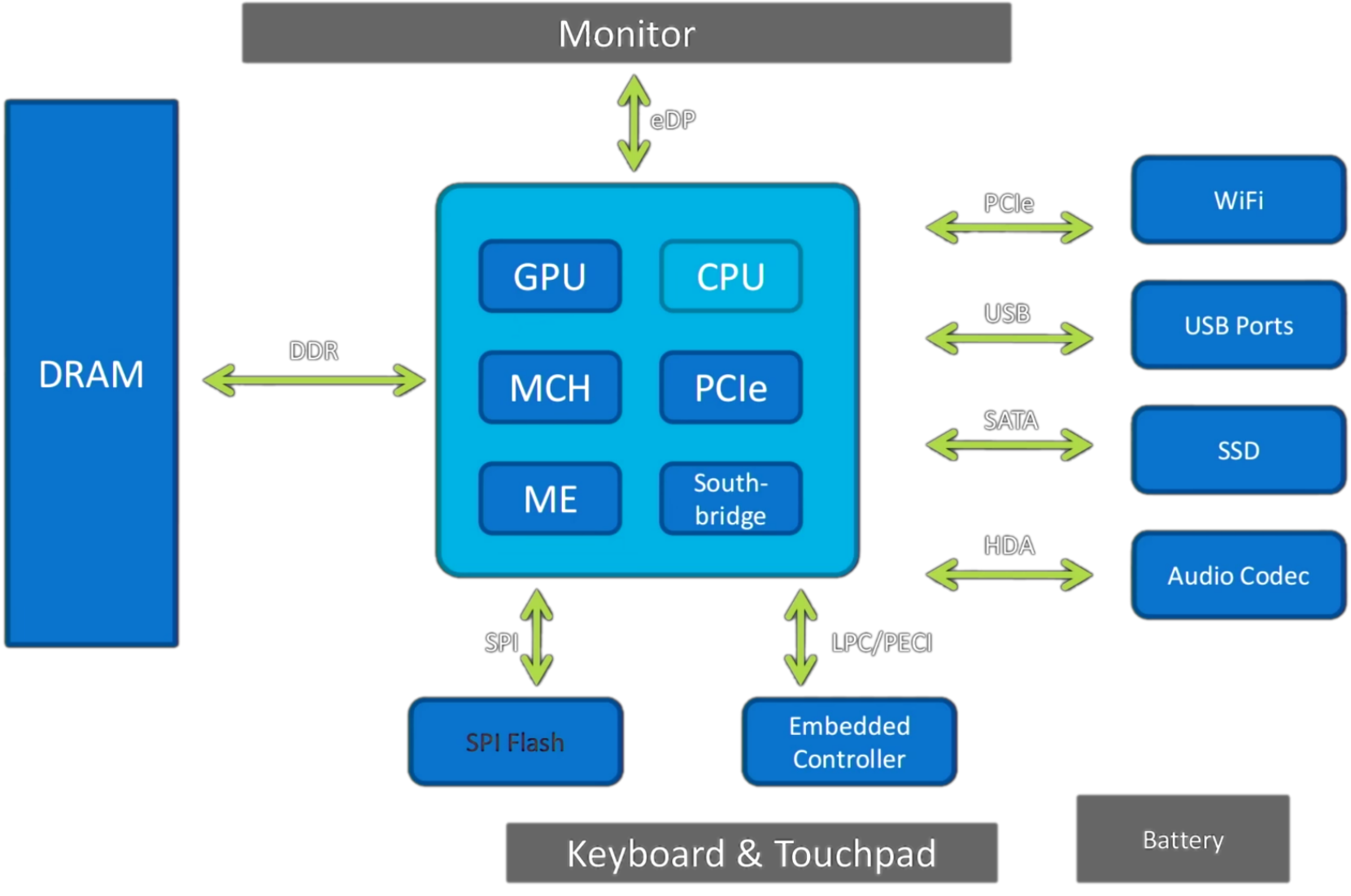
Intel AMT is an *out-of-band* management technology, offering:

- network tunnel over untrusted network
- remote power control
- remote KVM
- network packet filter
- PAVP for DRM media
- more ...

Intel ME


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

Intel ME


Monitor
eDP
DRAM
DDR
GPU
CPU
MCH
PCIe
ME
South-
bridge
PCIe
WiFi
USB
USB Ports
SATA
SSD
HDA
Audio Codec
SPI
LPC/PECI
SPI Flash
Embedded
Controller
Keyboard & Touchpad
Battery

ME capabilities

Intel ME has access to:

- Any memory region
- The PCI bus
- The GPU
- Wired and wireless NIC (with dedicated MAC address)
- more ...

The firmware

Its firmware is proprietary, so not security auditable, and it's signed with RSA by Intel

It's not encrypted but a lot of modules are Huffmann compressed with unknown hardware dictionary, so their code cannot be easily accessed.

How do I disable it?

How do I disable it?

Until 1st generation Core CPUs (Nehalem) it was possible to remove the ME firmware by modifying the *Intel Flash Descriptor* (see the libreboot page)

From Nehalem onwards, if the firmware is removed, the Computer turns off after 30 minutes; this is probably done to avoid the bypass of Intel Anti-Theft (now discontinued)

Result

In all modern Intel computers we have a perfect backdoor framework, not removable, and with complete access to all the machine resources.

Also, ME is active even in S5 power state (computer off)

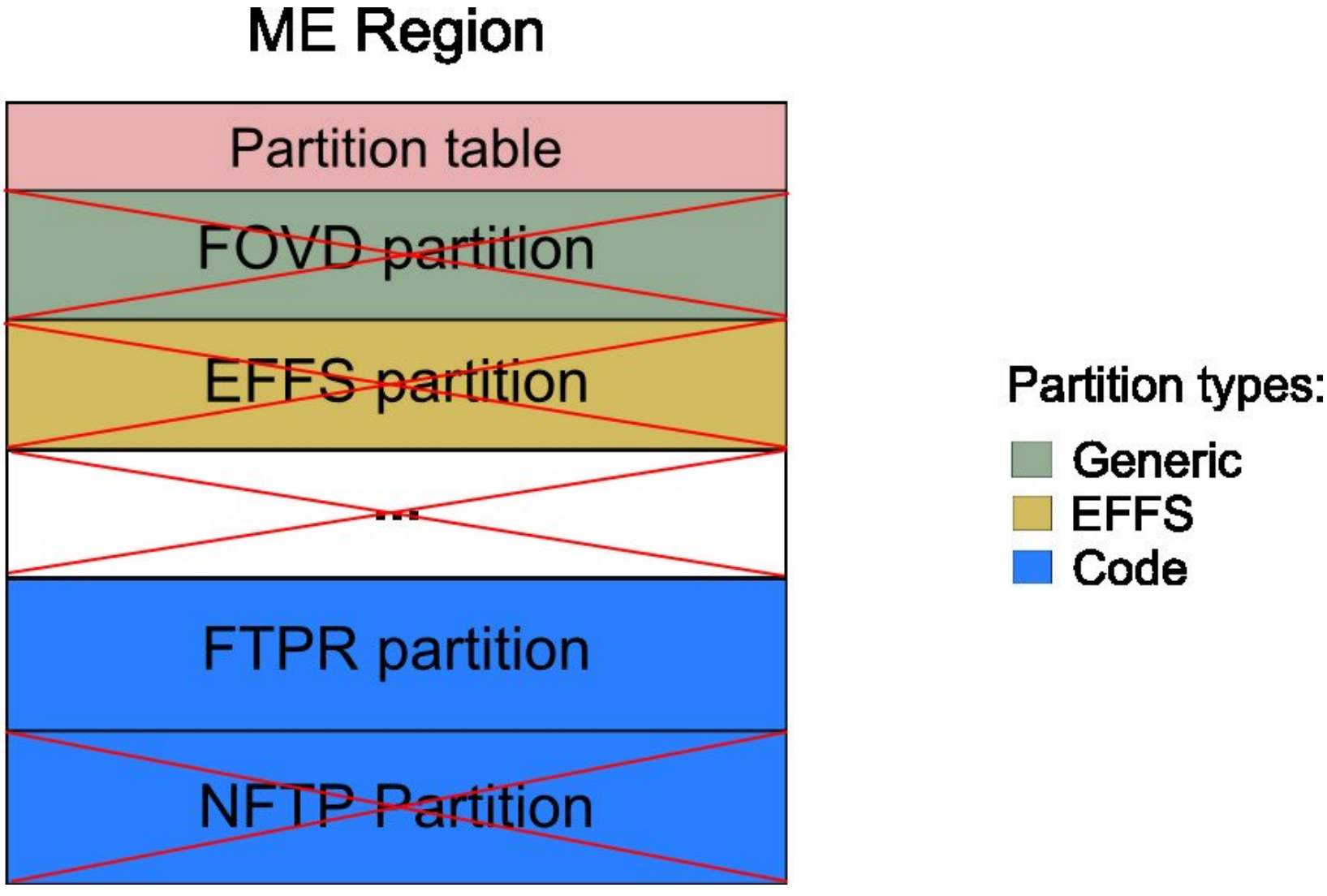
Is there anything that we can do?

Is there anything that we can do?

In September 2016 Trammell Hudson discovered that wiping the first 4KB of Intel ME firmware from his Thinkpad X230 with coreboot, the Computer would still turn on and won't shut down after 30 minutes.

From this discovery he started digging and found that it is possible to remove:

- All the partitions but the main one
- All the LZMA compressed modules from the main partition


ME Region
Partition table
FOVD partition
EFFS partition
...
FTPR partition
NFTP Partition
Partition types:
Generic
EFFS
Code

We can wipe the code

Even not removing completely Intel ME, this process strongly limits its capabilites, in fact it removes code for:

- Network access (contained in the removed NFTP partition)
- PAVP (Protected Audio-Video Path)
- The JVM (meant to enable the use of DRM applets)

me_cleaner.py

In November 2016, me and Nicola began testing with Intel ME to replicate Hudson's results and see how much more code we could remove.

To aid this purpose, Nicola wrote a python script to remove as much code as possible from an Intel ME firmware image.

github.com/corna/me_cleaner

Our findings

We confirmed Trammell Hudson's work, in particular:

- The partition table can be removed (an internal one will be used)
- All the partitions can be removed except FTPR (the main one)
- All the LZMA modules inside the FTPR can be removed
- Also all the Huffmann modules but one (BUP) can be removed.

This way we are left with just 50KB of code (compressed size)

- This works on Platforms from Sandy Bridge to Broadwell.
- These modifications work even with an OEM BIOS

Drawbacks?

Removing the code appears to hang Intel ME

But apart from AMT, there are more things ME can do in a system:

- Platform clock configuration
- Remote thermal monitoring
- Silicon Workaround

Until now ~30 `me_cleaner` users reported no bugs.

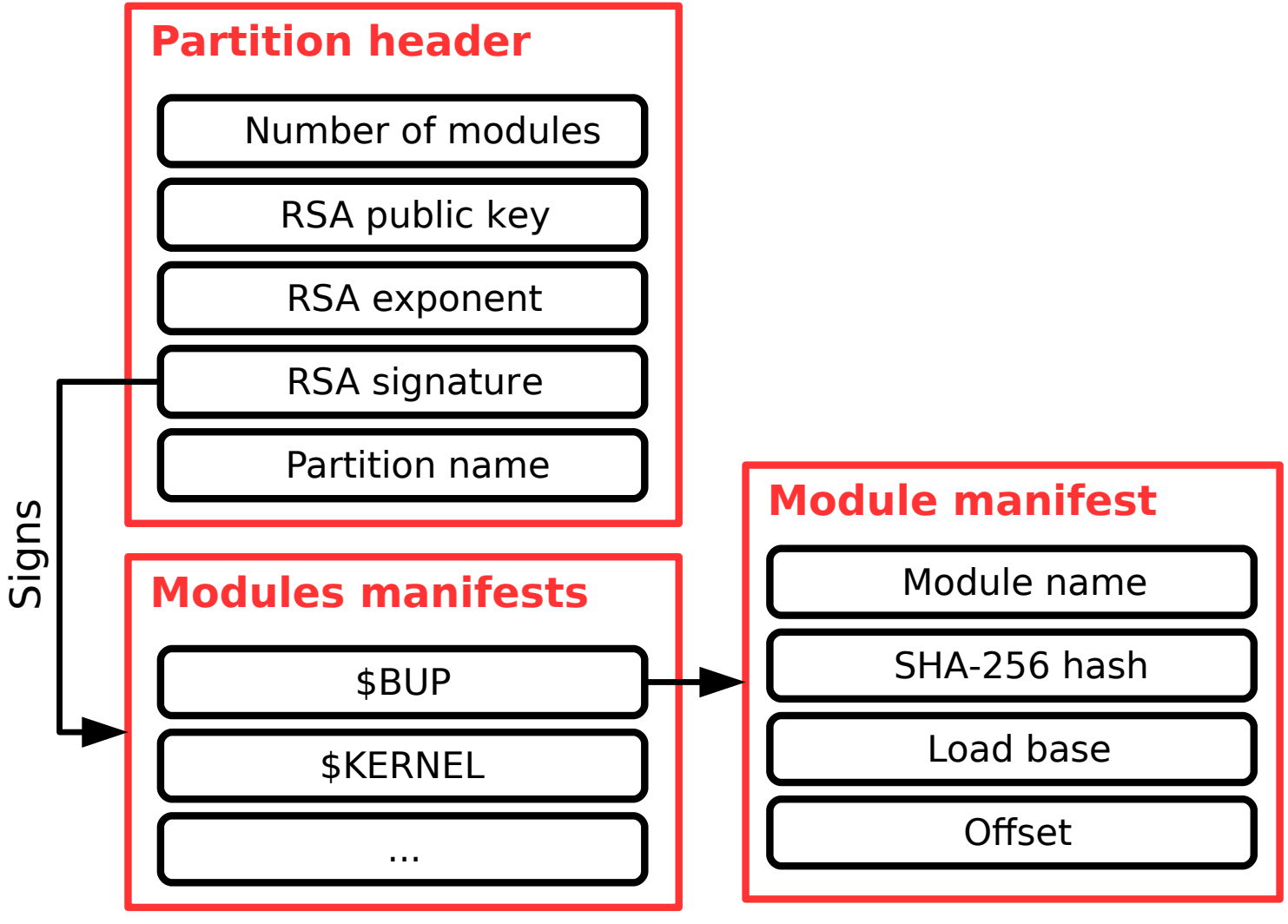
Signatures

The code is **signed at partition level**:

- removing an entire partition *doesn't break a signature*,
- but removing modules from a partition *should break it*.

Then why the system boots even if we remove some modules?

The signatures work like this:



The signatures work like this:

Partition header -*sign*-> manifest list -*hash*-> manifest

The *module manifest list* signature is valid because it contains the old hashes (we don't modify them).

But those hashes are broken **because we removed the code**.

Apparently the hash check is done before executing each module.

So we **can not run unsigned code**, but we can boot the system.

The last question is:

How can I be sure that Intel ME does not have a backup ROM inside with a fallback firmware?

The ROMB partition

Luckily the reverse engineering work of Igor Skochinsky answer us: Inside some versions of ME firmware, there is a special partition called **ROMB** (ROM Bypass)

This partition contains code that overrides the internal ROM, used to fix bugs in early silicon.

The ROMB partition

Igor analized this *update partition* and found out it contains:

- Common C functions (memcpy, memset, strcpy...)
- ThreadX routines (Intel ME RTOS)
- Low level hardware access API

The ROMB partition

The internal ROM appears to:

- Do basic hardware init
- Check the FTPR partition signature
- Load the BUP module and jumps to it

Proof of this can be found inside "Intel Confidential" documents that you can find on Google

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**.

Try to google *"loading an ME FW binary image"* or *"management engine system tools"*, including quotation marks.

Final notes

If you want to test `me_cleaner` on your computer,

- You can find a guide here
- And here there are more details

Final notes

Here is a rough compatibility table

Gen	with vPRO	without vPRO
Core2	OK[1]	OK[1]
Nehalem	WIP	WIP
Sandy - Ivy Bridge	OK[2]	OK[2]
Haswell - Broadwell	NO	OK[2]
Skylake	NO	OK[3]

1 See this libreboot page (no code left)

2 Only BUP module left (~50KB)

3 Only FTPR partition left (~668KB)

More info:

Intel ME Firmware Structure

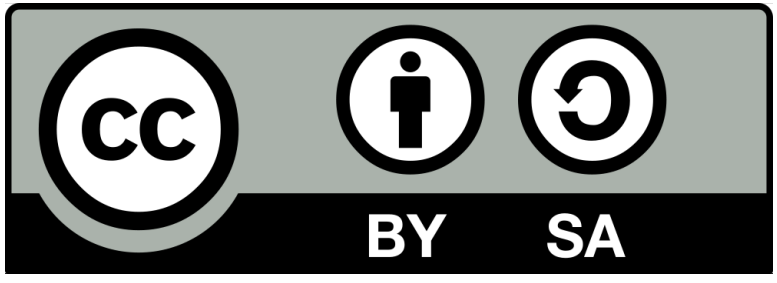
Igor Skochinsky - Rootkit in your laptop - 2012

Igor Skochinsky - Intel ME Secrets - 2014

Trammel Hudson - coreboot Mailing List

Nicola Corna - coreboot Mailing List

Thank you!



www.poul.org