Advanced x86:
BIOS and System Management Mode Internals
SPI Flash Programming

Xeno Kovah && Corey Kallenberg

LegbaCore, LLC

All materials are licensed under a Creative Commons “Share Alike” license.

http://creativecommons.org/licenses/by-sa/3.0/

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

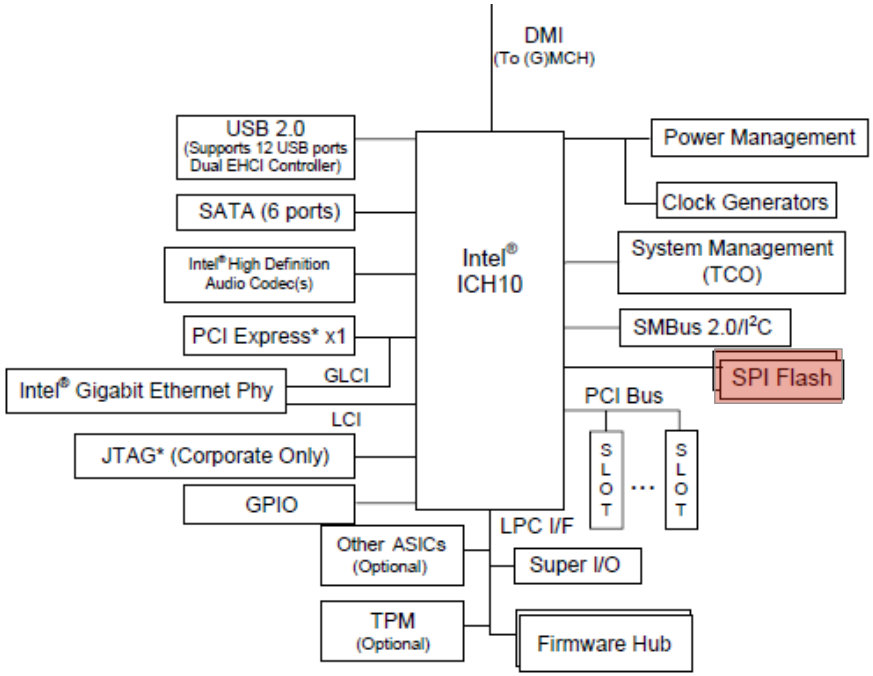
Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Attribution condition: You must indicate that derivative work
"Is derived from John Butterworth & Xeno Kovah’s ’Advanced Intel x86: BIOS and SMM’ class posted at http://opensecuritytraining.info/IntroBIOS.html”

SPI note:

- We're not really going to care about low level SPI protocol details
- We're just going to care about the way that it's exposed to the BIOS, so that we can understand the BIOS's view of the world, and therefore interpret its actions accordingly

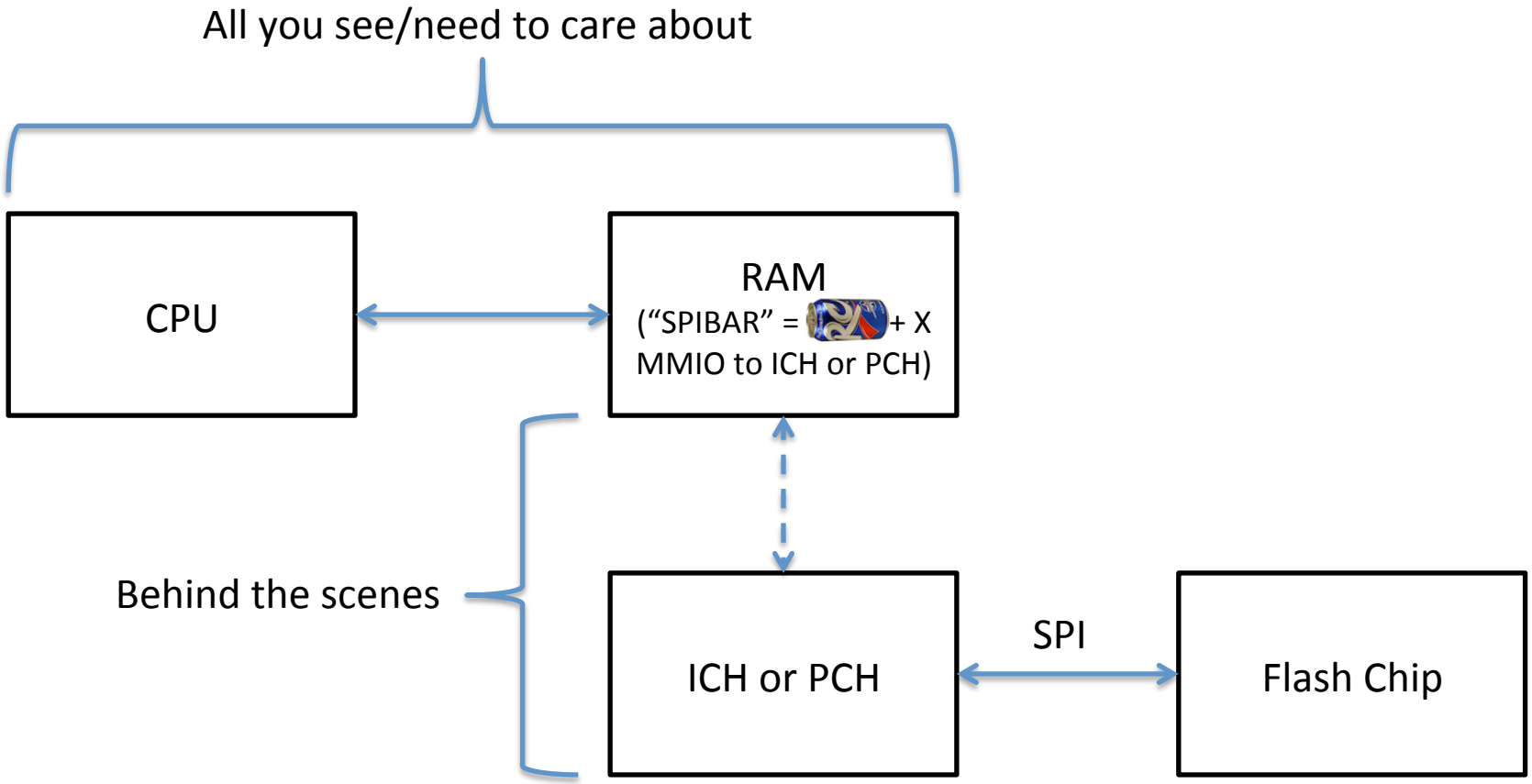
SPI (Serial Peripheral Interface) Flash



- Intel provides a programmable interface to the SPI flash device
  - System BIOS lives here
  - Other stuff does too
- Copernicus programs this interface to dump a binary of the SPI flash

Intel IO Controller Hub 10 Datasheet, page 31

How you should think of reading & writing the flash chip



Programming the SPI Flash

- *RCBA holds RCRB*
- RCRB + X = SPIBAR
  - (for X = 0x3800 on all newer systems, but not old ones)
- SPIBAR + Y = flash programming registers
- An app can choose either "Hardware Sequencing" (meaning the hardware picks the actual SPI commands that get sent for read/write) or Software Sequencing (meaning you pick the actual SPI commands)
  - For simplicity of discussion, we'll be referring to only those operations/details pertaining to Hardware Sequencing
    - Software Sequencing just offers a little more fine-grain control

All SPI registers in the following slides are from:
http://www.intel.com/content/www/us/en/io/io-controller-hub-10-family-datasheet.html

Since I didn't have anywhere better to put this…

21.1 Serial Peripheral Interface Memory Mapped Configuration Registers

The SPI Host Interface registers are memory-mapped in the RCRB (Root Complex Register Block) Chipset Register Space with a base address (SPIBAR) of 3800h and are located within the range of 3800h to 39FFh. The address for RCRB are in the RCBA Register (see Section 12.1.40). The individual registers are then accessible at SPIBAR + Offset as indicated in the following table.

These memory mapped registers must be accessed in byte, word, or DWord quantities.

Table 21-1. Serial Peripheral Interface (SPI) Register Address Map (SPI Memory Mapped Configuration Registers) (Sheet 1 of 2)

SPIBAR + Offset	Mnemonic	Register Name	Default
00h–03h	BFPR	BIOS Flash Primary Region	00000000h
04h–05h	HSFS	Hardware Sequencing Flash Status	0000h
06h–07h	HSFC	Hardware Sequencing Flash Control	0000h
08h–0Bh	FADDR	Flash Address	00000000h
0Ch–0Fh	—	Reserved	00000000h
10h–13h	FDATA0	Flash Data 0	00000000h
14h–4Fh	FDATAN	Flash Data N	00000000h

SPI Programming Flash Address Register

- Specifies starting address of the SPI I/O cycle
  - Flash address, not a system RAM address
  - Valid range is 0 to <size of flash chip – 1>

FADDR—Flash Address Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 08h	Attribute:	R/W
Default Value:	00000000h	Size:	32 bits

Bit	Description
31:25	Reserved
24:0	**Flash Linear Address (FLA)** — R/W. The FLA is the starting byte linear address of a SPI Read or Write cycle or an address within a Block for the Block Erase command. The Flash Linear Address must fall within a region for which BIOS has access permissions. Hardware must convert the FLA into a Flash Physical Address (FPA) before running this cycle on the SPI bus.

SPI Programming Data Registers

- Contains the data we just read from the SPI flash (up to 64 bytes), or data we're about to write to the flash chip
- R/W (since it can be used to specify data to write to flash)

FDATA0—Flash Data 0 Register
(SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 10h	Attribute:	R/W
Default Value:	00000000h	Size:	32 bits

FDATAN—Flash Data [N] Register
(SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 14h	Attribute:	R/W
	...		
	SPIBAR + 4Ch		
Default Value:	00000000h	Size:	32 bits

SPI Programming Control Register

HSFC—Hardware Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 06h	Attribute:	R/W, R/WS
Default Value:	0000h	Size:	16 bits

2:1	**FLASH Cycle (FCYCLE)** — R/W. This field defines the Flash SPI cycle type generated to the FLASH when the FGO bit is set as defined below: 00 = Read (1 up to 4 bytes by setting FDBC) 01 = Reserved 10 = Write (1 up to 4 bytes by setting FDBC) 11 = Block Erase

- Set the type to read (bits 2:1 == 00b)

SPI Programming Control Register

- Initiates the SPI I/O cycle
  - Used by programming app (Copernicus)
- Defines the number of bits to read (or write) in the I/O cycle

HSFC—Hardware Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 06h	Attribute:	R/W, R/WS
Default Value:	0000h	Size:	16 bits

13:8	**Flash Data Byte Count (FDBC)** — R/W. This field specifies the number of bytes to shift in or out during the data portion of the SPI cycle. The contents of this register are 0s based with 0b representing 1 byte and 111111b representing 64 bytes. The number of bytes transferred is the value of this field plus 1. This field is ignored for the Block Erase command.

0	**Flash Cycle Go (FGO)** — R/W/S. A write to this register with a 1 in this bit initiates a request to the Flash SPI Arbiter to start a cycle. This register is cleared by hardware when the cycle is granted by the SPI arbiter to run the cycle on the SPI bus. When the cycle is complete, the FDONE bit is set. Software is forbidden to write to any register in the HSFLCTL register between the FGO bit getting set and the FDONE bit being cleared. Any attempt to violate this rule will be ignored by hardware. Hardware allows other bits in this register to be programmed for the same transaction when writing this bit to 1. This saves an additional memory write. This bit always returns 0 on reads.

SPI Programming Status Register

- Indicates that an SPI I/O cycle is in progress
- Set automatically by hardware, nothing we need to really care much about

HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 04h	Attribute:	RO, R/WC, R/W
Default Value:	0000h	Size:	16 bits

5	**SPI Cycle In Progress (SCIP)**— RO. Hardware sets this bit when software sets the Flash Cycle Go (FGO) bit in the Hardware Sequencing Flash Control register. This bit remains set until the cycle completes on the SPI interface. Hardware automatically sets and clears this bit so that software can determine when read data is valid and/or when it is safe to begin programming the next command. Software must only program the next command when this bit is 0. **NOTE:** This field is only applicable when in Descriptor mode and Hardware sequencing is being used.

SPI Programming Status Register 2

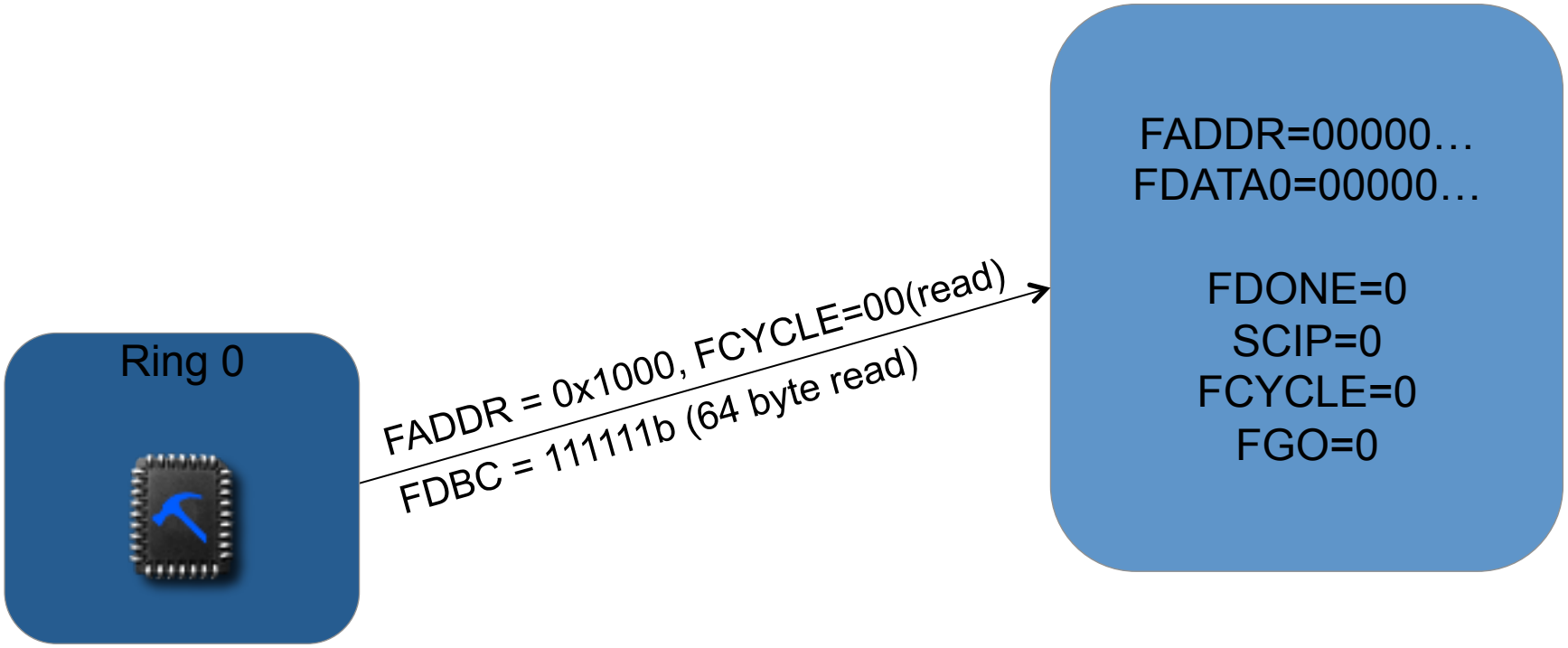
- Indicates the SPI I/O cycle has completed
- Software must poll on this bit to determine when the hardware is done reading/writing data

HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 04h	Attribute:	RO, R/WC, R/W
Default Value:	0000h	Size:	16 bits

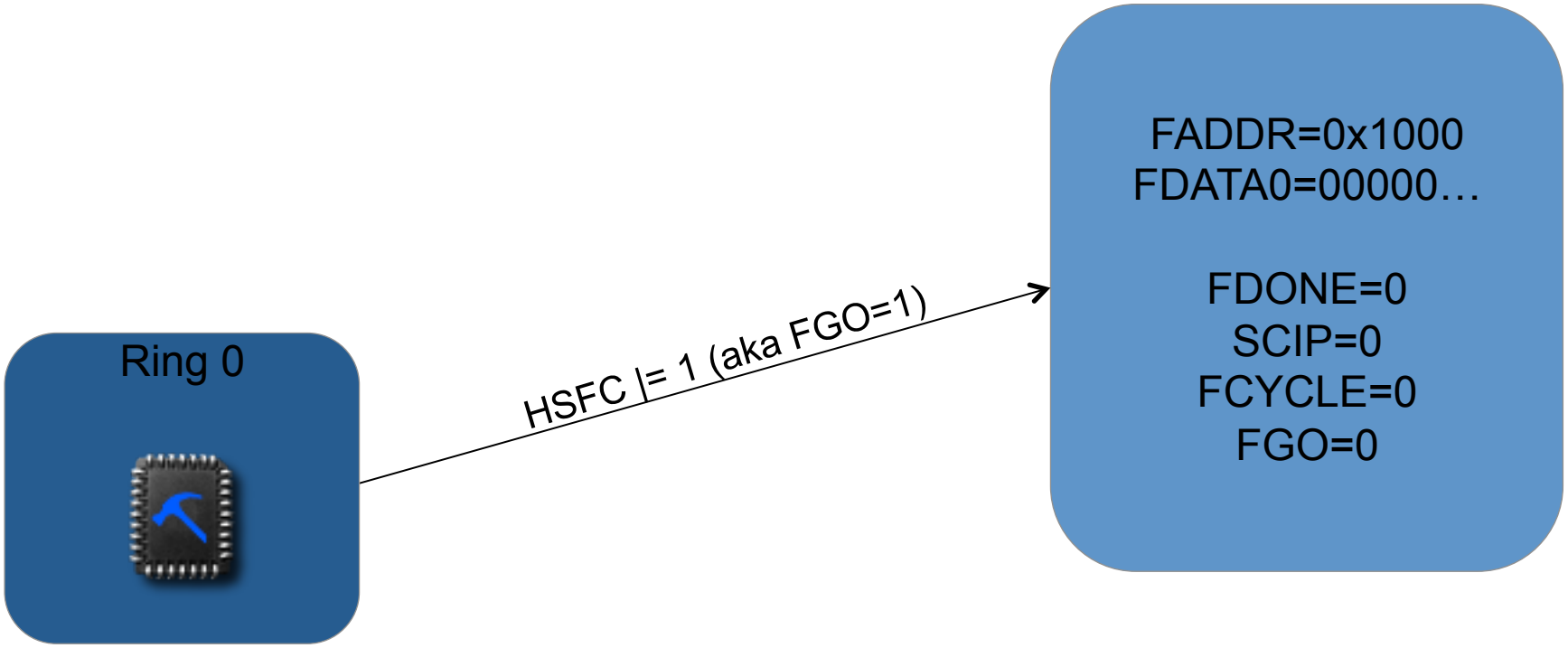
0	**Flash Cycle Done (FDONE)** — R/W/C. The ICH sets this bit to 1 when the SPI Cycle completes after software previously set the FGO bit. This bit remains asserted until cleared by software writing a 1 or hardware reset due to a global reset or host partition reset in an Intel ME enabled system. When this bit is set and the SPI SMI# Enable bit is set, an internal signal is asserted to the SMI# generation block. Software must make sure this bit is cleared prior to enabling the SPI SMI# assertion for a new programmed access. **NOTE:** This field is only applicable when in Descriptor mode and Hardware sequencing is being used.

Reading the flash chip



- BIOS reading software sets up the location it wants to read (as part of reading the entire chip) and how many bytes to read

Reading the flash chip



- BIOS reading software says to start the read

Reading the flash chip

FADDR=0x1000
FDATA0=00000…

FDONE=0
SCIP=1
FCYCLE=0
FGO=1

- Cycle in progress
- We need to poll on FDONE, waiting for it to be set to 1

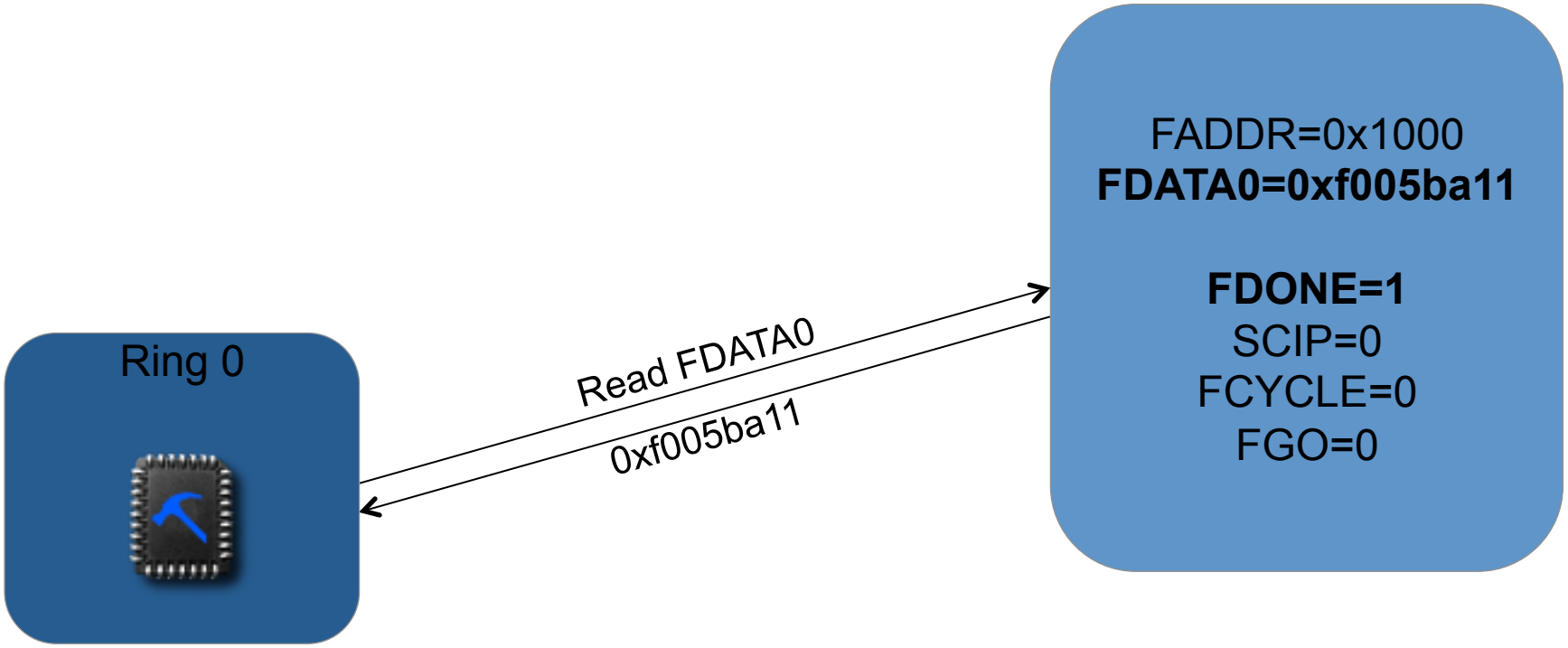
Reading the flash chip

FADDR=0x1000
FDATA0=0xf005ba11

FDONE=1
SCIP=0
FCYCLE=0
FGO=0

- Once the cycle is done (FDONE=1) we can do read the FDATA registers

Reading the flash chip



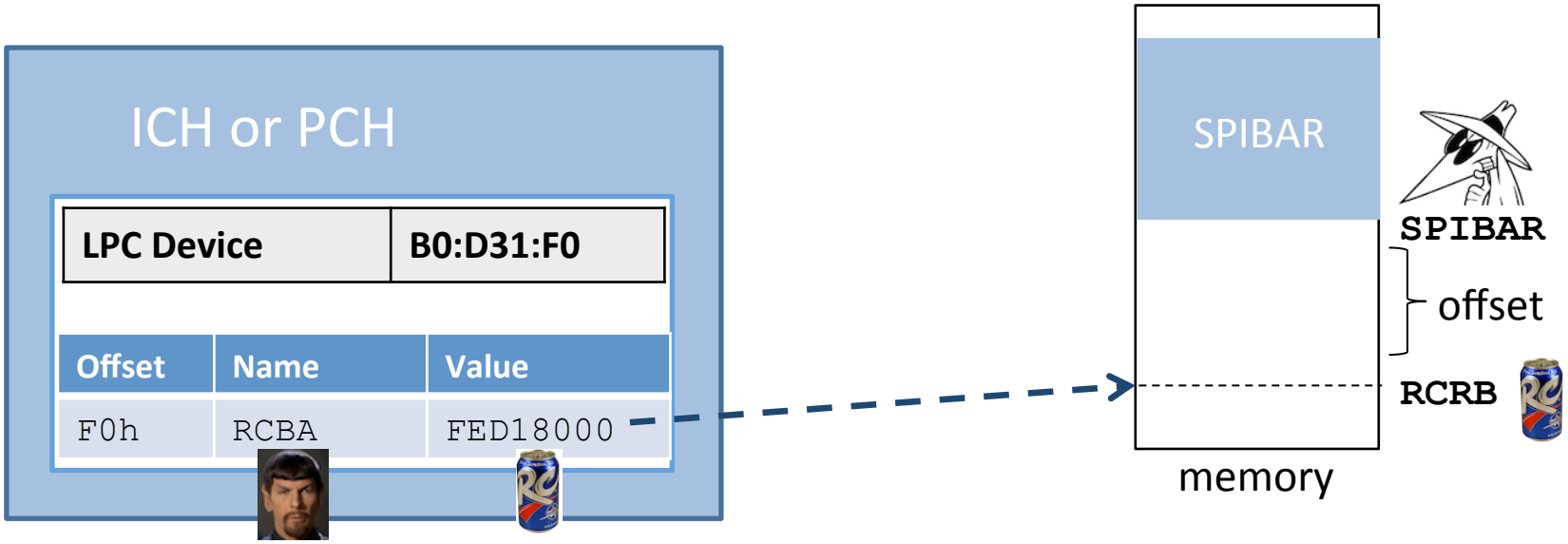
- BIOS reading software will get the contents out of the FDATA register(s) and store to memory and/or disk

Lab: RYOFC!

(Read Your Own Flash Chip!)

- We're going to be cool and manually program the flash registers in RWE to *read* the reset vector from our flash chip
- DISCLAIMER! DISCLAIMER! DISCLAIMER!
- ***Don't follow along unless you're super careful! You could accidentally brick your system! :D***
- ***I am not responsible if you brick your system (or your enemies' :P)***
- DISCLAIMER! DISCLAIMER! DISCLAIMER!

Register Access: SPI Base Address Register (SPIBAR)



- SPI extends it's Base Address Registers (BARs) to memory starting at an address called SPIBAR
- SPIBAR is offset from the Root Complex Register Block (RCRB)
- The offset from RCRB is chipset-dependent, but will be listed in the SPI section of the datasheet

Lab: Locate SPIBAR

Serial Peripheral Interface Memory Mapped Configuration Registers

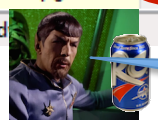
The SPI Host Interface registers are memory-mapped in the RCRB (Root Complex Register Block) Chipset Register Space with a base address (SPIBAR) of 3800h and are located within the range of 3800h to 39FFh. The address for RCRB can be found in RCBA Register see Section 13.1.35. The individual registers are then accessible at SPIBAR + Offset as indicated in Table 22-1.

These memory mapped registers must be accessed in byte, word, or dword quantities.

Bus 00, Device 1F, Function 00 - Intel Corporation ISA Bridge

0	03020100	07060504	0B0A0908	0F0E0D0C
00	29178086	02100107	06010003	00800000
10	00000000	00000000	00000000	00000000
20	00000000	00000000	00000000	02331028
30	00000000	000000E0	00000000	00000000
40	00001001	00000080	00001081	00000010
50	00000000	00000000	00000000	00000000
60	8A8B8A83	000000D1	808B838A	000000F8
70	00000000	00000000	00000000	00000000
80	3C040000	007C0901	00000000	003C0C81
90	00000000	00000000	00000000	00000000
A0	00000E20	00800239	004A1C2B	40000300
B0	00F00000	00000000	00010008	00000000
C0	00000000	00000000	00000000	00000000
D0	00000000	00000000	0000F080	00000008
E0	100C0009	03C40200	00000004	00000000
F0	FED18001	00000000	00030F86	00000000

Hard



déjà vu!

- Per your datasheet, the SPI host interface registers (SPIBAR) is located at an offset from RCRB
- In the Mobile 4-Series chipset on our lab laptop, is located at:
- RCRB + 3800h = FED1_B800h

Back to “Fun things to do in SMM”!

Attack 1 – Manipulate Copernicus output

- From within the OS, targeted hooks into Copernicus code
- From within the OS with “DDefy” [20] rootkit style hooks into file writing routines
- From within the HD controller firmware [21][22][23]
- From within the OS with a network packet filter driver
- From within the NIC firmware [24][25]
- Etc. Lots more options

Attack 2 – A new generic attack.

- It is possible for SMM to be notified when SPI reads or writes occur
- An attacker who controls the BIOS controls the setup of SMM
- In this way a BIOS-infecting attacker can perform a SMM MitM attack against those who would try to read the BIOS to integrity check it
- We call our SMM MitM “Smite’em, the Stealthy”

Eye of the dragon - FSMIE - hw sequencing

- **This is what allows an attacker in SMM to know when someone is trying to access the flash chip**

HSFC—Hardware Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers)

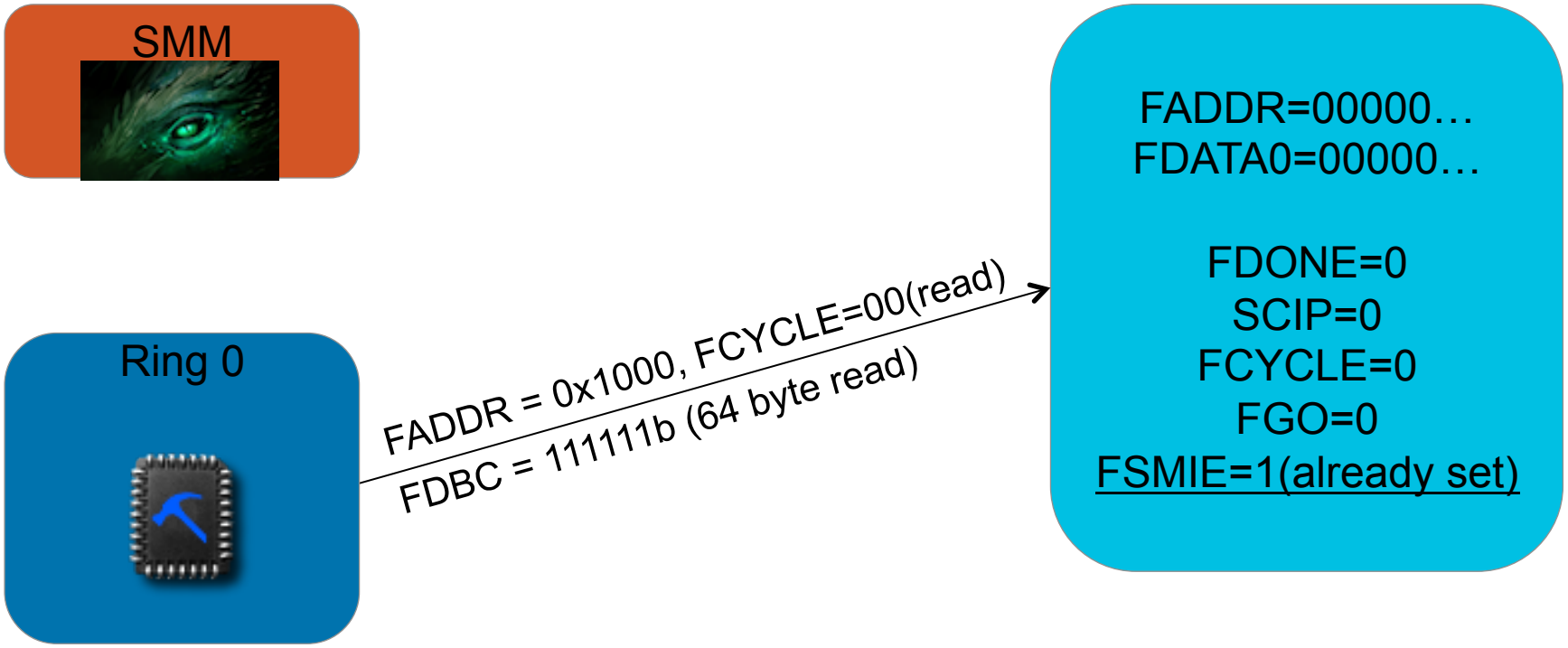
Memory Address:SPIBAR + 06h Attribute: R/W, R/WS
Default Value: 0000h Size: 16 bits

This register is only applicable when SPI device is in descriptor mode.

Bit	Description
15	**Flash SPI SMI# Enable (FSMIE)** — R/W. When set to 1, the SPI asserts an SMI# request whenever the Flash Cycle Done bit is 1.

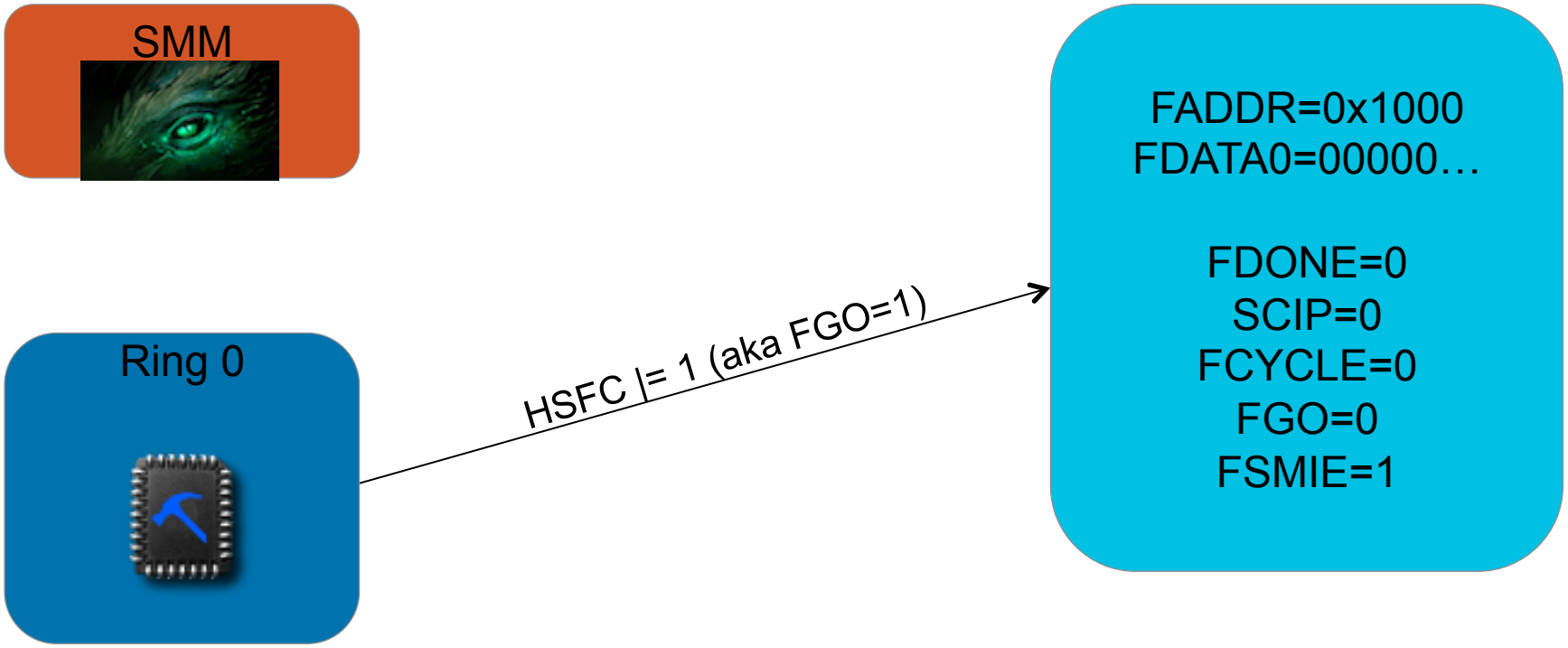
- **The Flash Cycle Done bit is set to 1 after every read and write**

Reading the flash chip in the presence of Smite'em



- **BIOS reading software sets up the location it wants to read (as part of reading the entire chip) and how many bytes to read**

Reading the flash chip in the presence of Smite'em



- **BIOS reading software says to start the read**

Reading the flash chip in the presence of Smite'em

SMM

Ring 0

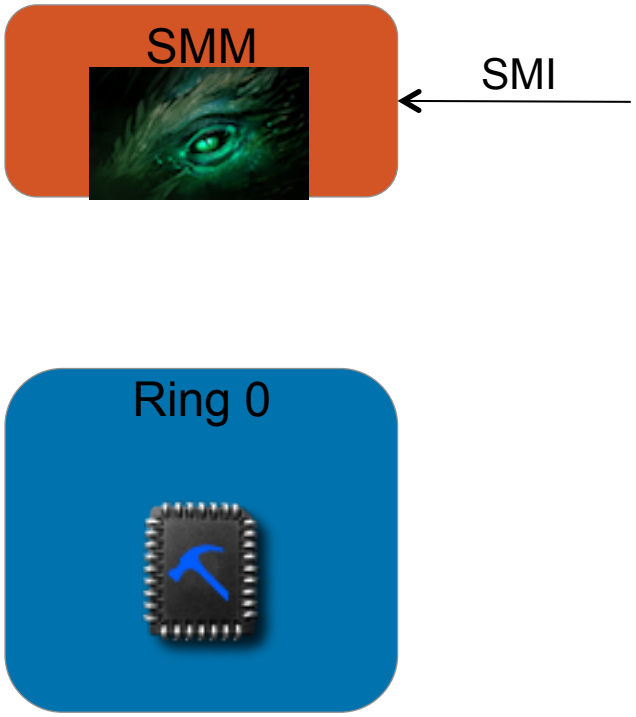
FADDR=0x1000
FDATA0=00000…

FDONE=0
SCIP=1
FCYCLE=0
FGO=1
FSMIE=1

- **Cycle in progress**

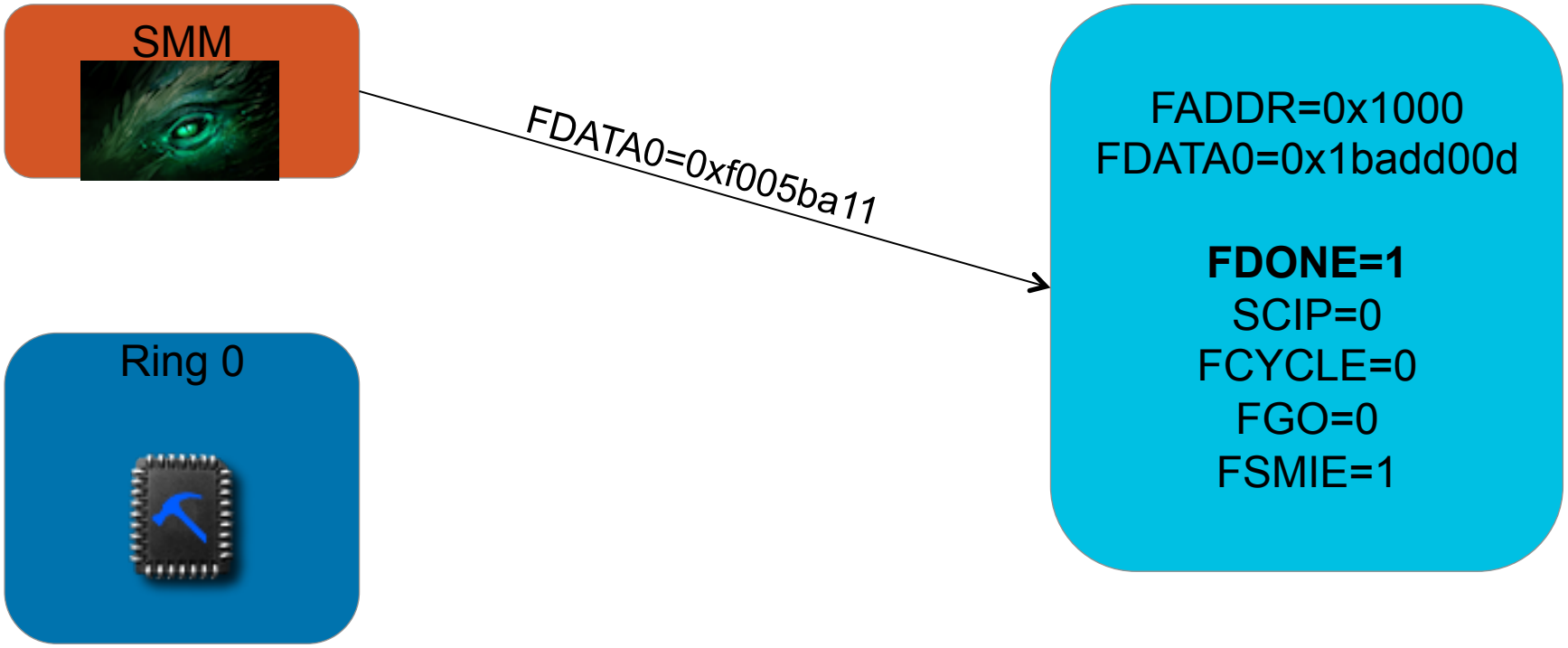
MITRE

Reading the flash chip in the presence of Smite'em



- **Once the cycle is done, and the data is available for reading, if the FSMIE = 1, an SMI is triggered, giving Smite'em the first look**

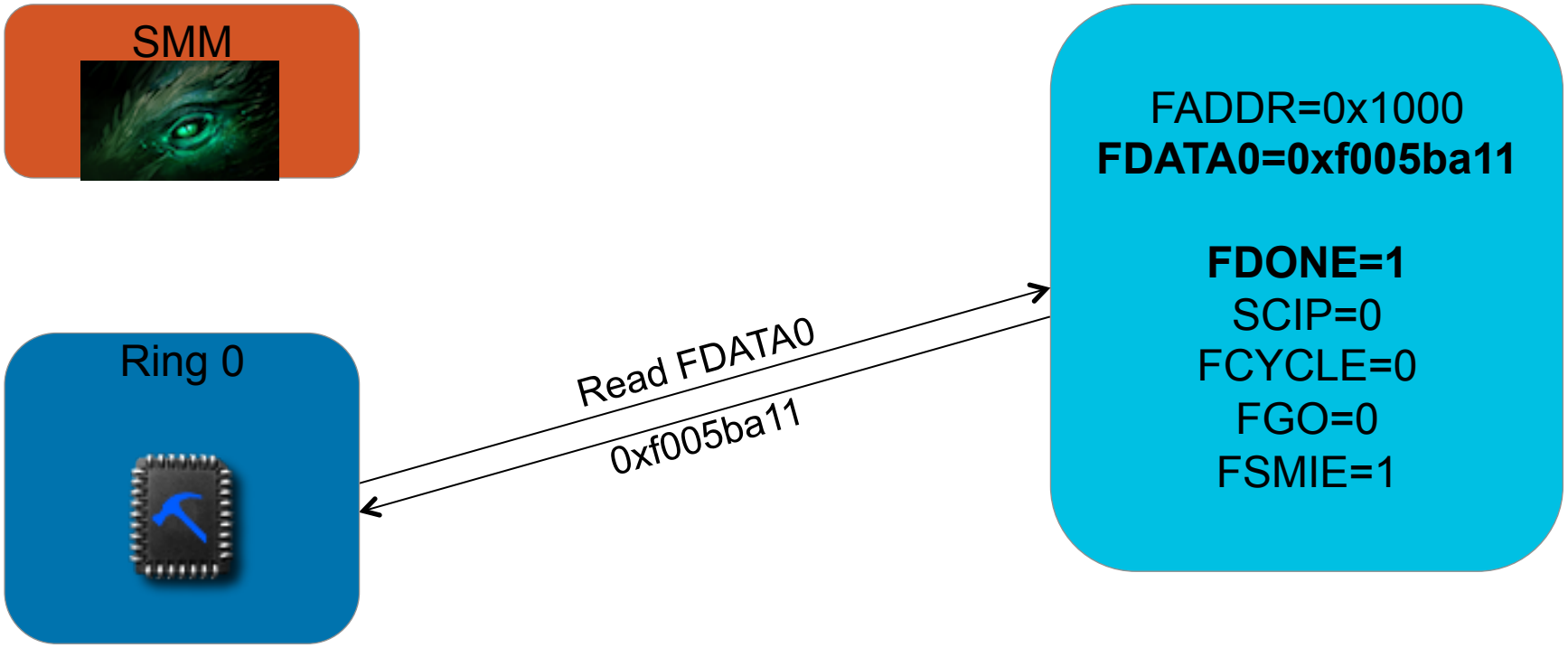
Reading the flash chip in the presence of Smite'em



- **Smite'em can change any data that would reveal its presence to the original benign data**

MITRE

Reading the flash chip in the presence of Smite'em



- **BIOS reading software will be misled!**

Think it can't happen?

- **Flashrom 0.9.7 source**

```
ichspi.c:                hsfc = REGREAD16(ICH9_REG_HSFC);
ichspi.c:                hsfc &= ~HSFC_FCYCLE; /* set read operation */
ichspi.c:                hsfc &= ~HSFC_FDBC; /* clear byte count */
ichspi.c:                hsfc |= (((block_len - 1) << HSFC_FDBC_OFF) & HSFC_FDBC);
ichspi.c:                hsfc |= HSFC_FGO; /* start */
```

- **If you don't account for hw/sw sequencing's FSMIE bit (as no previous software did), you will just lose and provide false assurances of a lack of BIOS compromise**

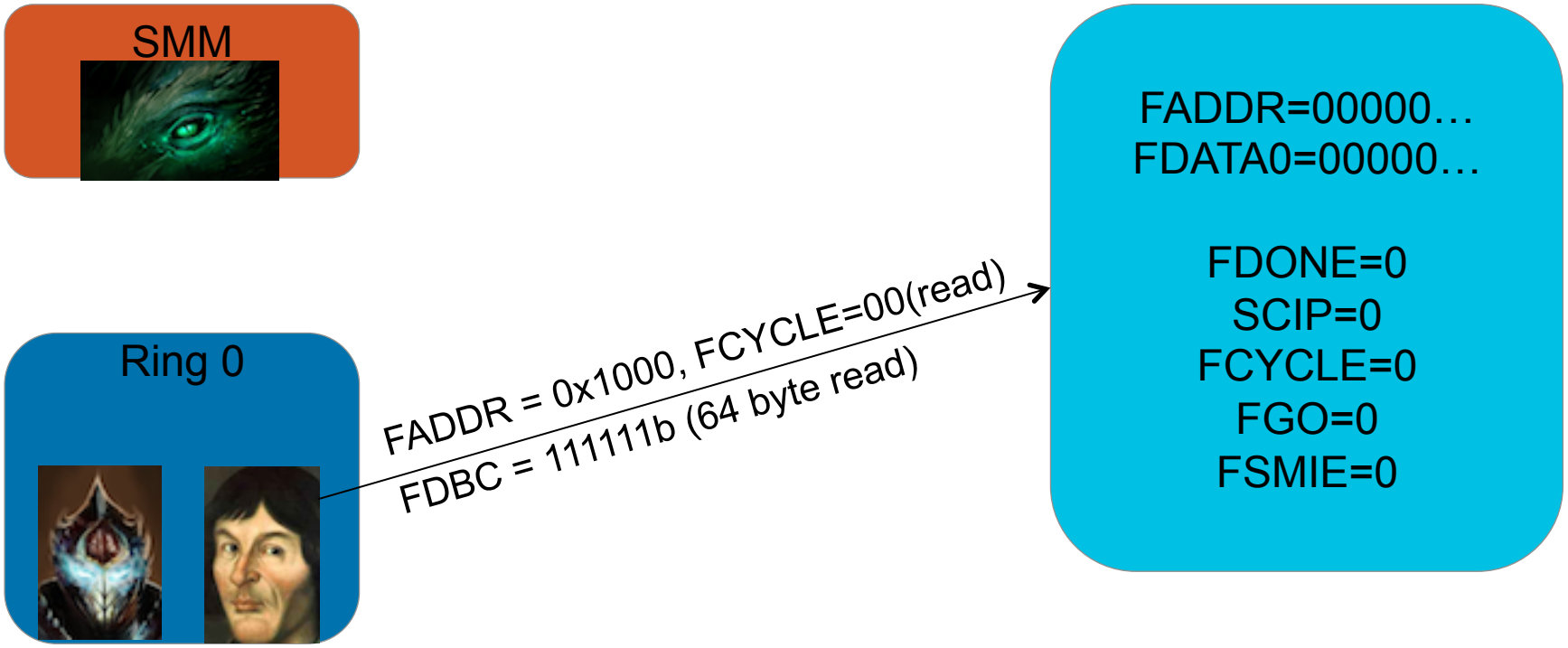
What you don't know can bite you

- **The basic solution would seem to be just for querying tools to set FSMIE = 0 before trying to read**
- **Multiple ways for an adversary to counter**
  - Kernel agent continuously setting FSMIE = 1
    - So you just clear it and check if it's getting re-set, and if so…?
  - VMX interception of MMIO to SPI space, falsifying that you successfully cleared FSMIE
    - But then if they're using VMX too, they can also just directly forge FDATA
  - Target your security software specifically
    - If your tool is good enough to detect attacker, he's incentivized to go after you specifically

Terror at 35,000 feet (high level overview) Another attack…

- **Let's assume that Smite'em wants to pick another generic, low-effort way to avoid detection (i.e. doesn't want to use VMX until absolutely necessary)**
- **Smite'em recruits a *Dragon Knight* avatar**
  - Could be kernel-based code or a DMA device and independent of CPU
- **Avatar polls SPI configuration registers to detect if an SPI cycle is in progress**
- **Upon detecting an SPI cycle in progress, the avatar triggers an SMI**
- **Smite'em running in SMM has exclusive access to the CPU, and can stall until the cycle completes and then replace the data read from flash before Copernicus can read it**

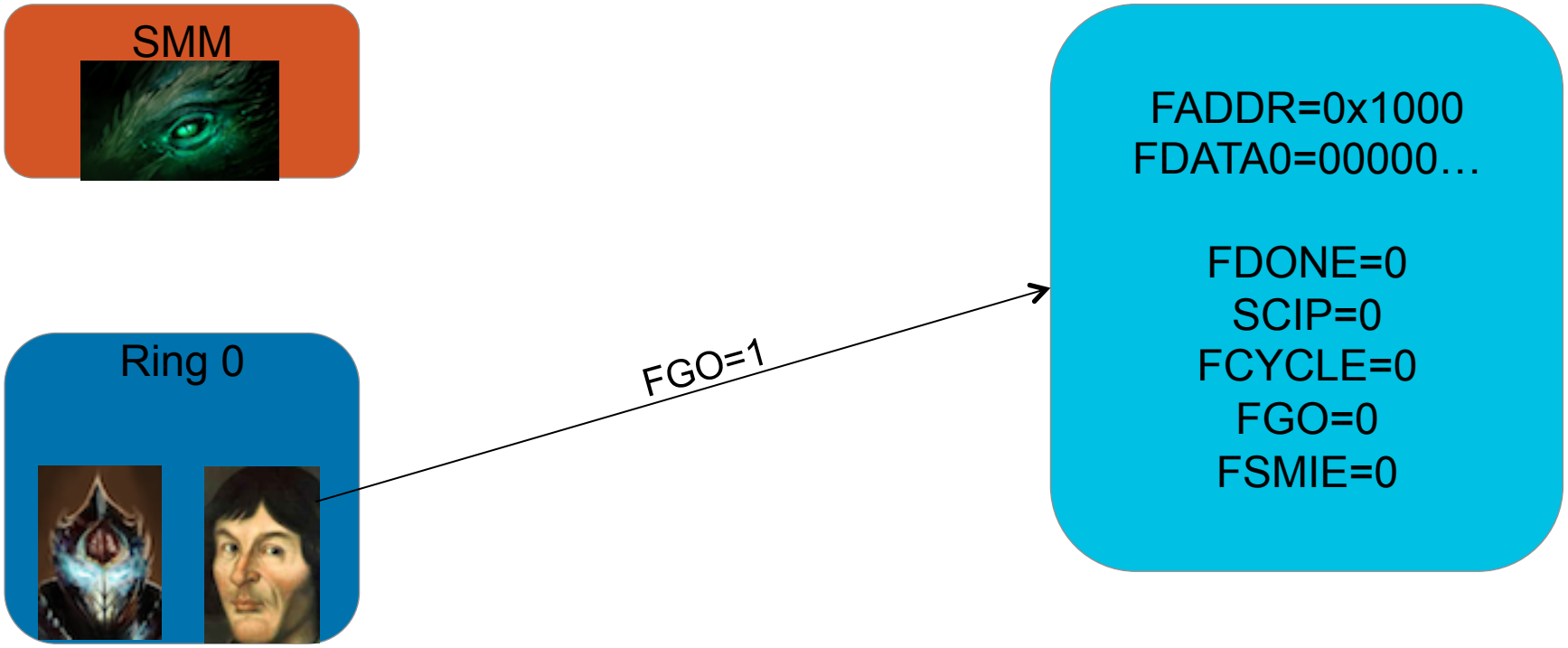
Reading the flash chip in the presence of Smite'em



- **Copernicus sets up the location it wants to read (as part of reading the entire chip) and how many bytes to read**

MITRE

Reading the flash chip in the presence of Smite'em

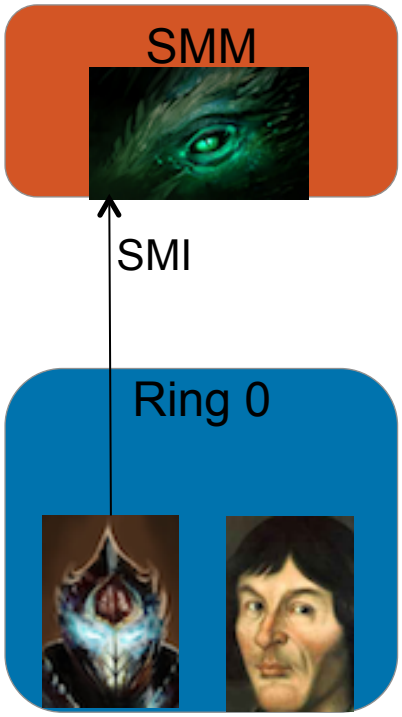


- **Then says go**

Reading the flash chip in the presence of Smite'em

- **Copernicus sets up the location it wants to read (as part of reading the entire chip) and how many**

Reading the flash chip in the presence of Smite'em



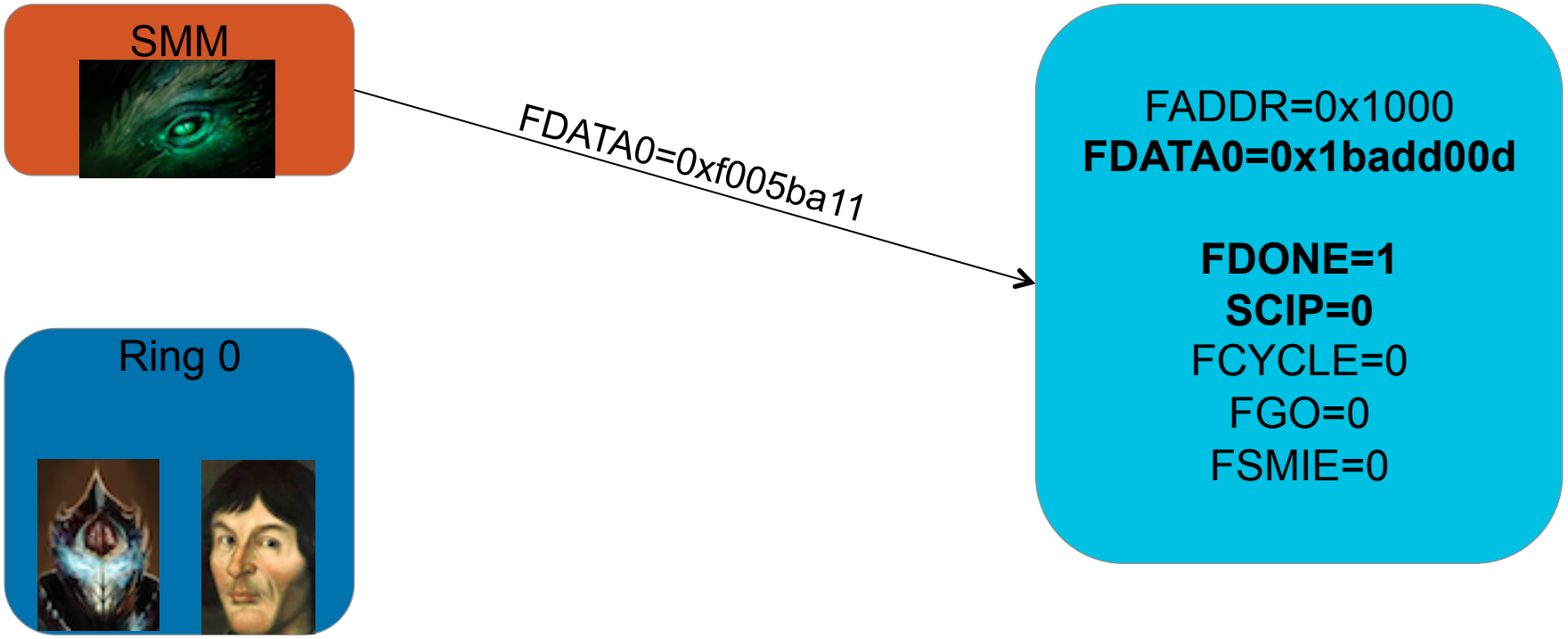
FADDR=0x1000
FDATA0=0x1badd00d

FDONE=1
SCIP=0
FCYCLE=0
FGO=0
FSMIE=0

- **Once it sees the data, it tries not to race with Copernicus, but instead stops itself and Copernicus by signaling Smite'em with an SMI**

MITRE

Reading the flash chip in the presence of Smite'em



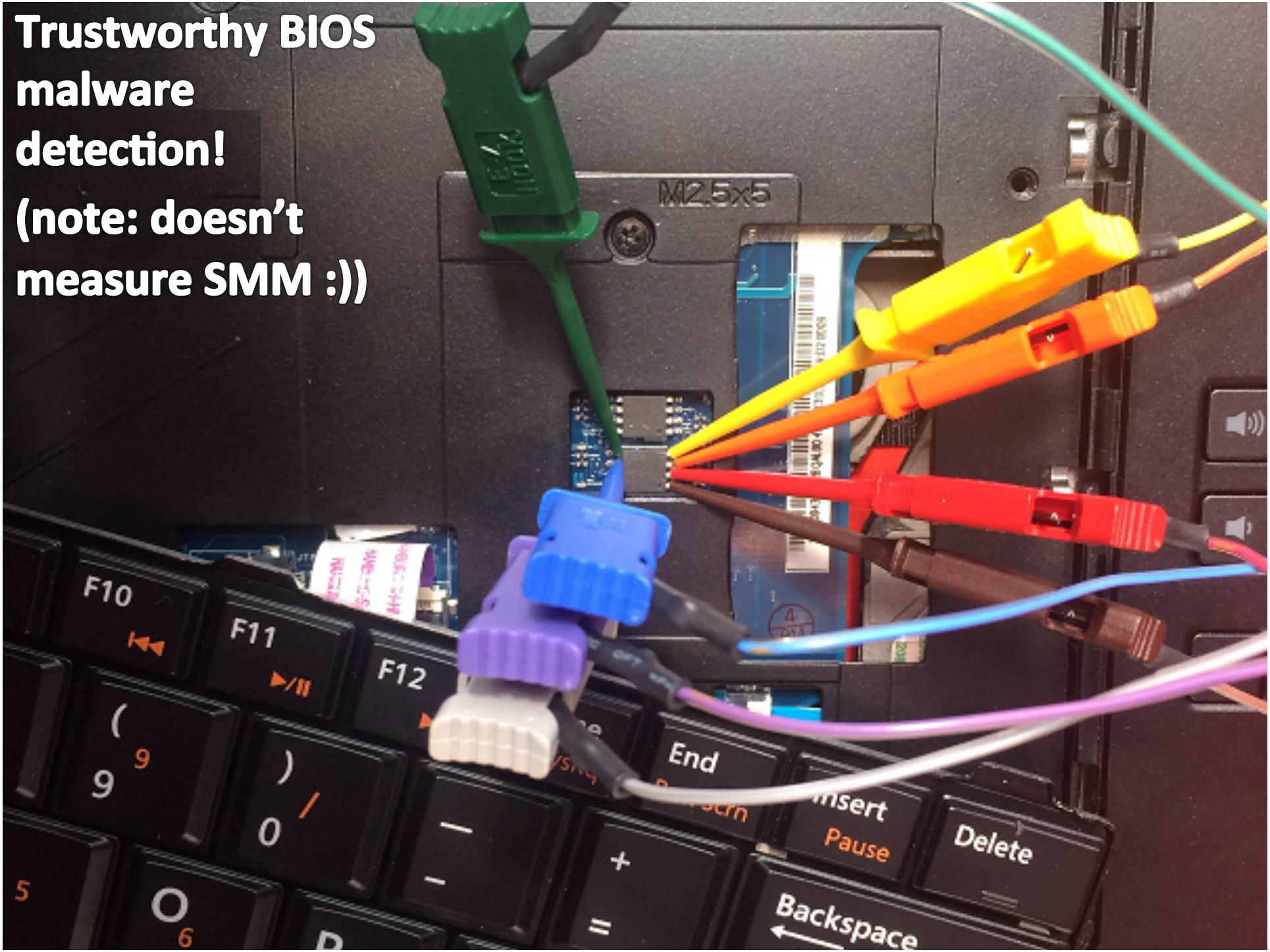
- **Smite'em then cleans up as usual**

Better luck next time?

- We then implemented a basic "Copernicus 2" that used Intel TXT to try and defeat this
  - We care about *trustworthy* detection tools. It's no good to have detection tools if your attacker can easily bypass them and give you a *false sense of trust* that your system is clean
    - But we know that we have to start somewhere
- We thought it would work because the manuals implied that TXT automatically suppresses SMIs until they're explicitly turned back on
  - It turned out that it does on *old* hardware, but newer hardware turns on SMIs automatically, so our TXT code is back to being vulnerable to SMM subversion until STMs are available
    - That's part of why we want STMs so much
  - :'(
  - TT
- Our only consolation is that Copernicus 2 *does* take OS level attackers off the table…But really if we're trying to detect BIOS level attackers, we need to counter SMM in order for it to matter

Coming soon?(ever?)

- “Copernicus 3” would take our existing research on “timing-based attestation” and port it into TXT-land
- Unfortunately since I said that idea while I was at MITRE, even though I didn’t implement it, probably MITRE owns the idea and I can’t do it without their permission/cooperation :-/
- So for today, your only option is…


Trustworthy BIOS malware detection!
(note: doesn't measure SMM :))
M2.5x5
F10
F11
F12
End
Insert
Pause
Delete
Backspace