Copernicus 2: SENTER the Dragon!

Xeno Kovah
John Butterworth
Corey Kallenberg
Sam Cornwell

MITRE

Can a tick[1], flea, or other BIOS malware hide from Copernicus?

- **A common question**
- **The answer: yes**
- **We built Copernicus to be something "best effort" that could be deployed quickly with minimal requirements, to try and catch any firmware malware "with their pants down"**
  - Where there was darkness, we said "let there be light!" ;)
  - When we live in a world where no one is checking their firmware, any firmware malware need not necessarily fear detection and thus can be vulnerable to a surprise detect
  - Just the act of existing costs attackers development time/money if they hadn't previously provided any self-protection
- **Now let's see what we need to do to make Copernicus actually trustworthy**

[1] see our previous BIOS Chronomancy work http://bit.ly/1g0Btz6

Attack 0 – DoS Copernicus

- Prevent Copernicus from running
- Possibly easily detected, but what are you going to do about it?

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

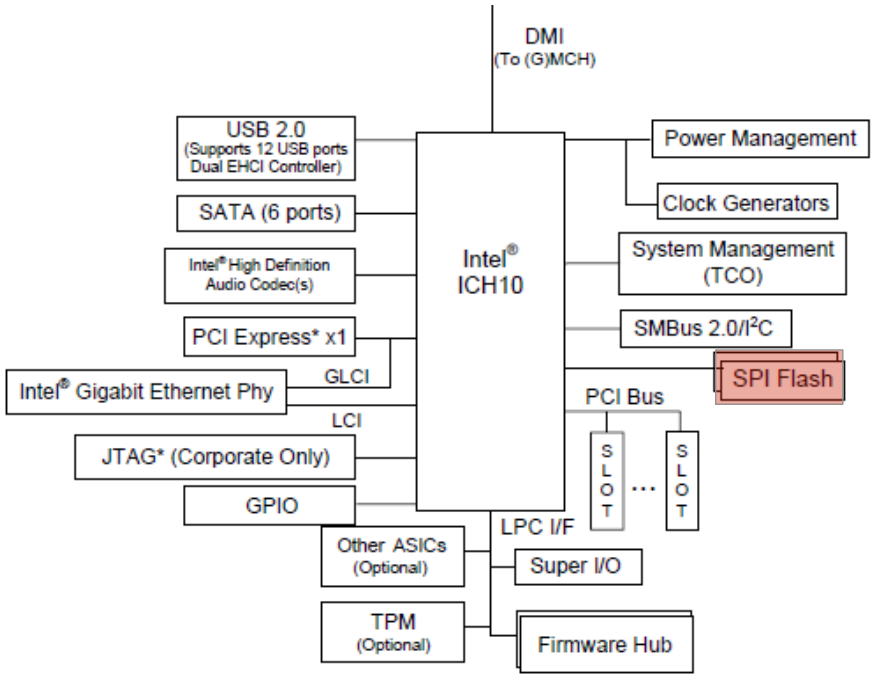
Attack 2 – A new attack. This has never been presented before.

- 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”

Smite'em: Engineering a dragon

- **Smite'em is a PoC attack that can MitM reads to the SPI Flash**
  - Thus it can conceal its presence even from applications that dump the SPI flash
  - Like Copernicus, Flashrom, Intel ChipSec, McAfee DeepDefender, Raytheon Pikeworks' firmware forensics, AFRL's stuff, etc
- **Multiple ways to design it**
  - Interrupt-driven – FSMIE bit
  - Polling – SCIP/FDONE bit
  - VMX-based
  - Targeted at specific defensive software

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

Programming the SPI Flash

- **SPI Host Interface registers are memory-mapped at an offset in the RCRB (Root Complex Register Block)**
- **An app can choose either Hardware Sequencing or Software Sequencing**
  - 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

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 read from the SPI flash (up to 64 bytes)
- R/W (since it can be used to specify data to write to flash)
- Smite'em overwrites this data from within SMM

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

- **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**

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**
- **Smite'em polls this bit to ensure the SPI I/O cycle has completed before forging the data in the FDATA registers**

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.

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
Default Value: 0000h
Attribute: R/W, R/WS
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**

Eye of the dragon - FSMIE - sw sequencing

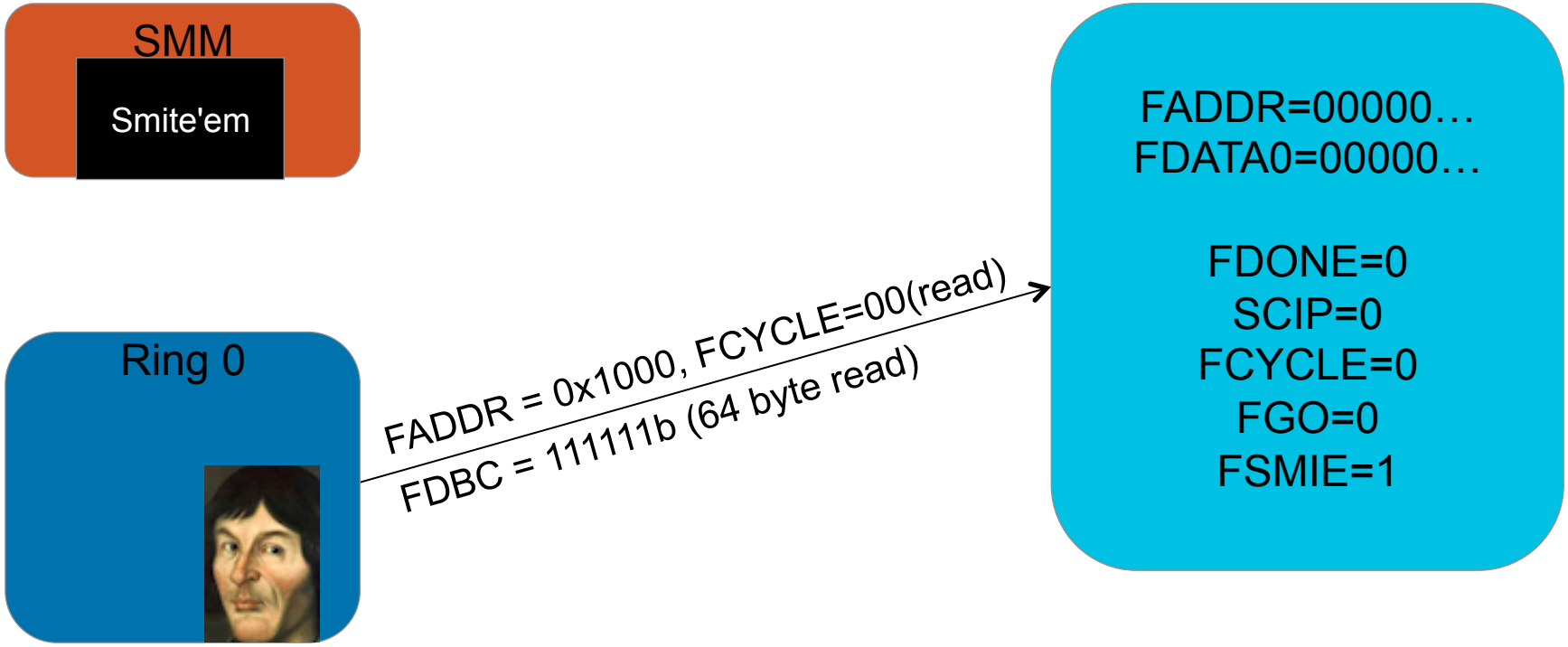
- **And here's the bit that gives the same functionality if someone is using software sequencing to access flash**

SSFC—Software Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers)

Memory Address:	SPIBAR + 91h	Attribute:	R/W
Default Value:	000000h	Size:	24 bits

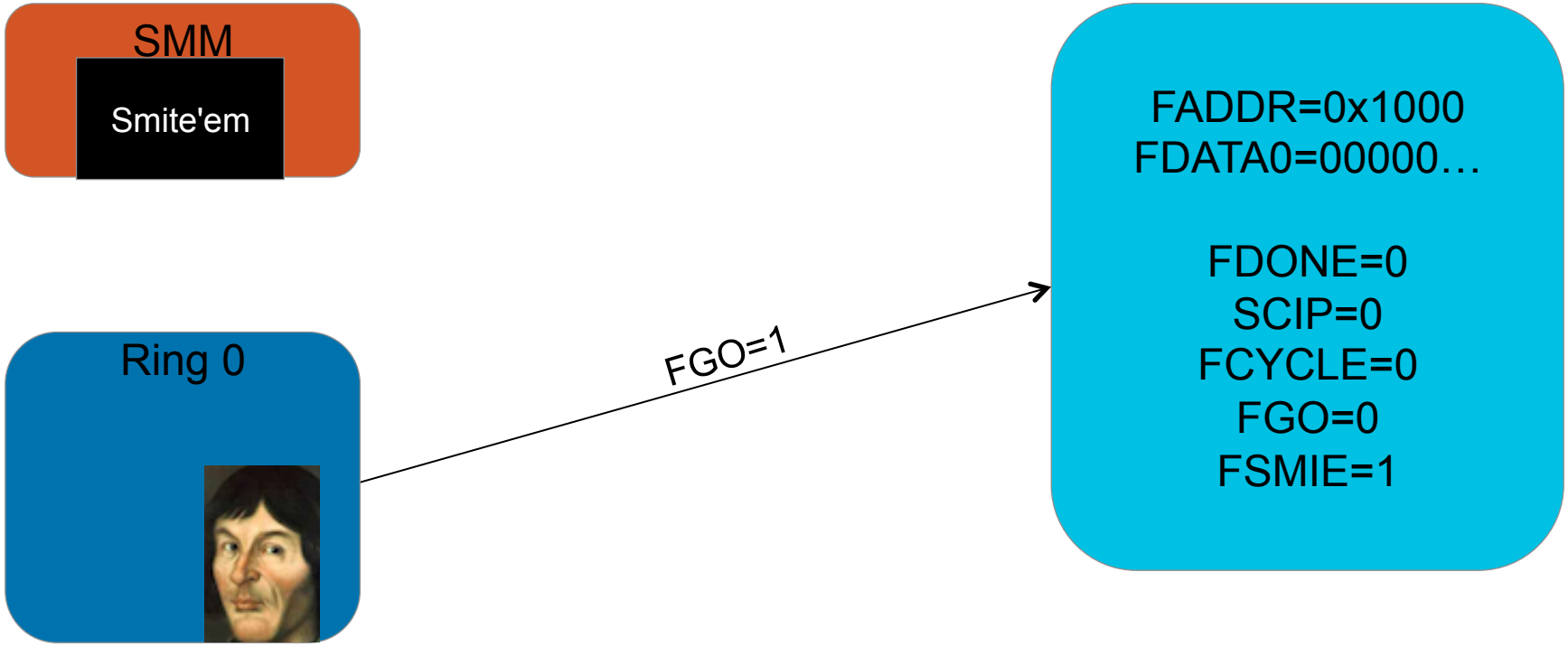
Bit	Description
15	**SPI SMI# Enable (SME)** — R/W. When set to 1, the SPI asserts an SMI# request whenever the Cycle Done Status bit is 1.

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**

Reading the flash chip in the presence of Smite'em



- **Copernicus says to start the read**

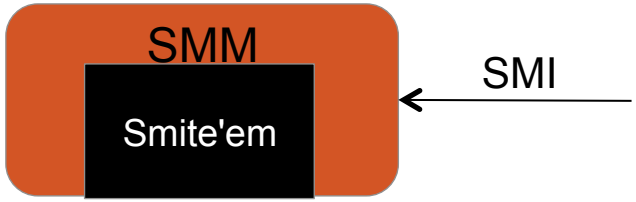
Reading the flash chip in the presence of Smite'em

FADDR=0x1000
FDATA0=00000…

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

- **Cycle in progress**

Reading the flash chip in the presence of Smite'em

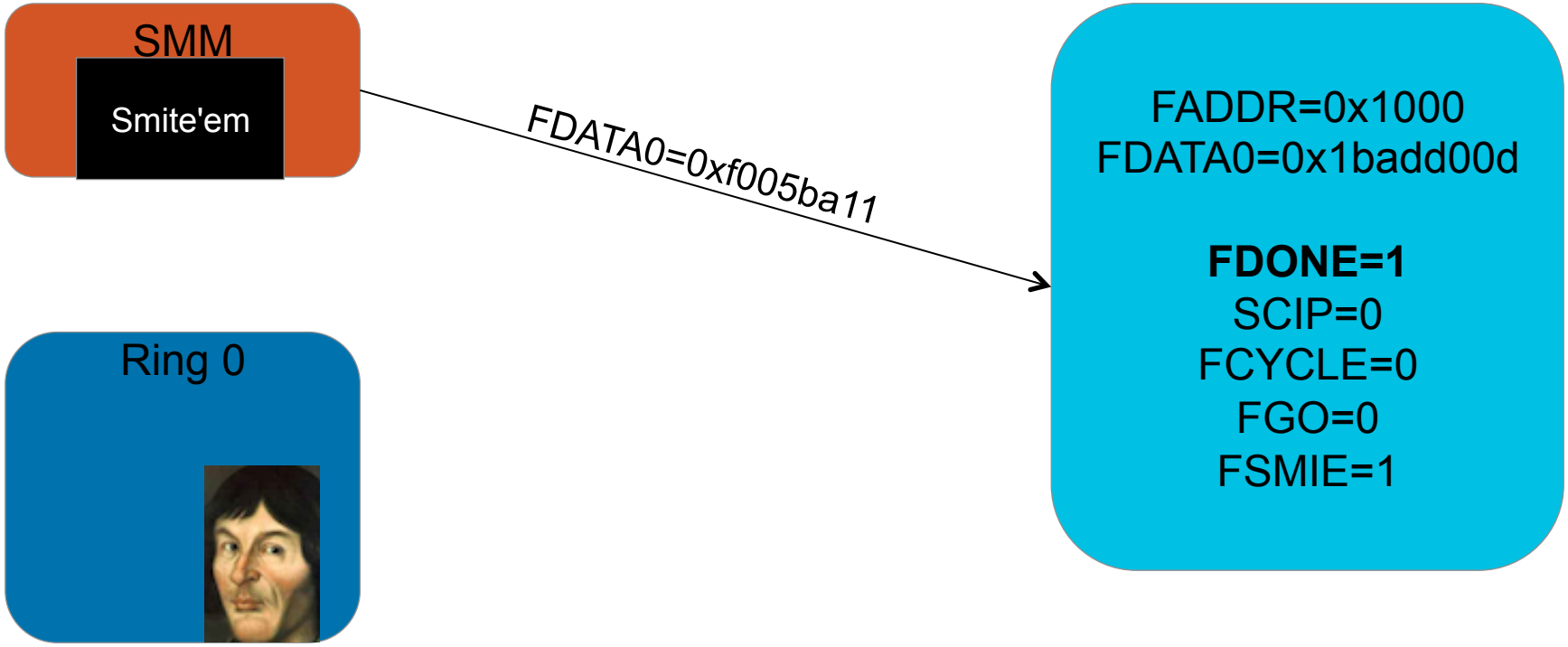


FADDR=0x1000
FDATA0=0x1badd00d

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

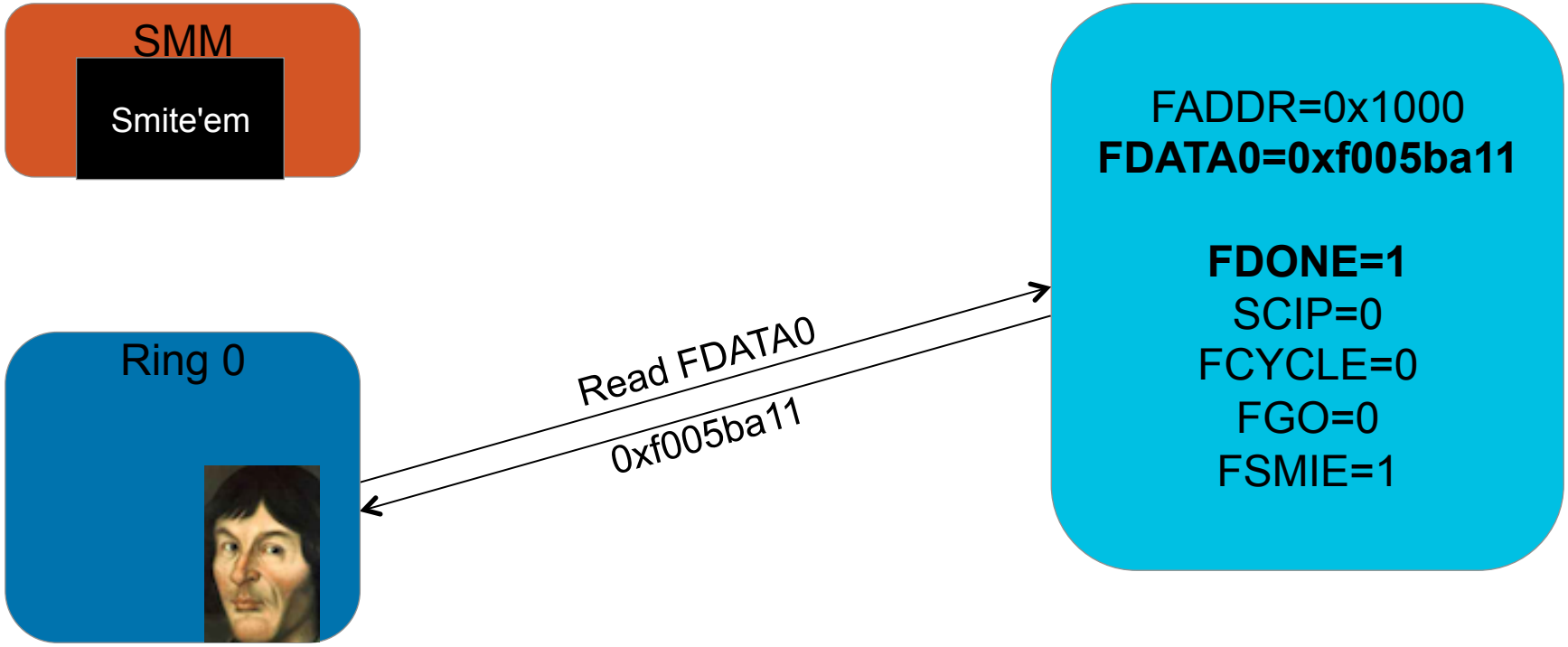
- **Once the cycle is done, and the data is available for reading, if the FSMIE = 1, an SMI is triggered, giving Smite'em the first look**

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**

Reading the flash chip in the presence of Smite'em



- **Copernicus 1 (or any other flash reading software) will be mislead**

What you don't know can bite you

- **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**
- **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 (aka high level overview)

- **Let's assume that Smite'em wants to pick the most generic, lowest-effort way to avoid detection (i.e. doesn't want to use VMX until absolutely necessary)**
- **Smite'em recruits an Avatar**
  - Could be kernel-based code or a DMA device and independent of CPU
- **Avatar polls SPI cycle registers to detect if an SPI cycle is in progress**
- **Upon detecting an SPI cycle in progress, the Avatar triggers an SMI**
- **Smite'em in SMRAM replaces data read from flash before Copernicus can read it**

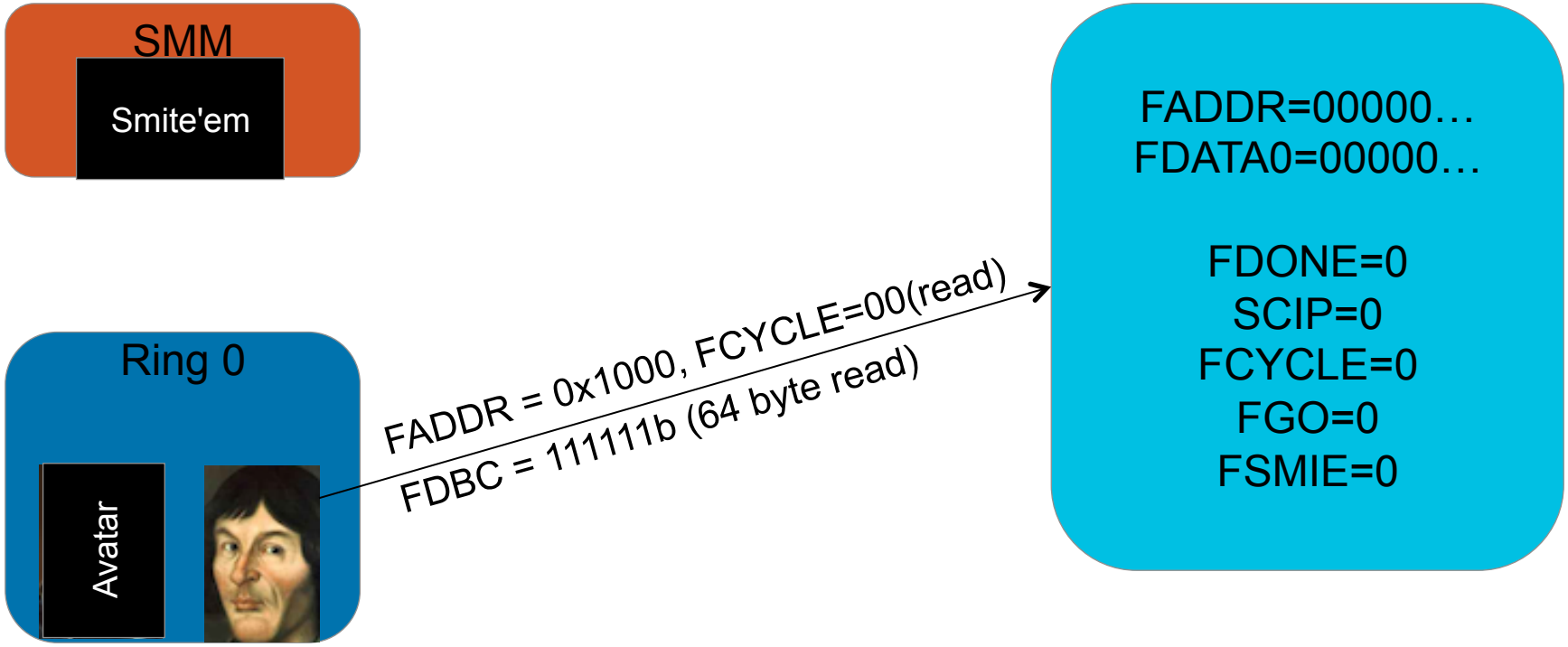
Smite'em Operation 1

- **Agent polls the SPI Cycle in Progress bit in the HSFC register**
  - Prefereably independent of the CPU (located on external device (PCI or otherwise)
- **When it detects an SPI cycle in progress (H/W automatically sets this bit), it triggers an SMI**
- **System transitions to SMM; Copernicus and all other processes are temporarily suspended**
  - MitM occurs in SMM to remove a race-condition where Copernicus reads the data before Smite'em can forge it

Smite'em Operation 2

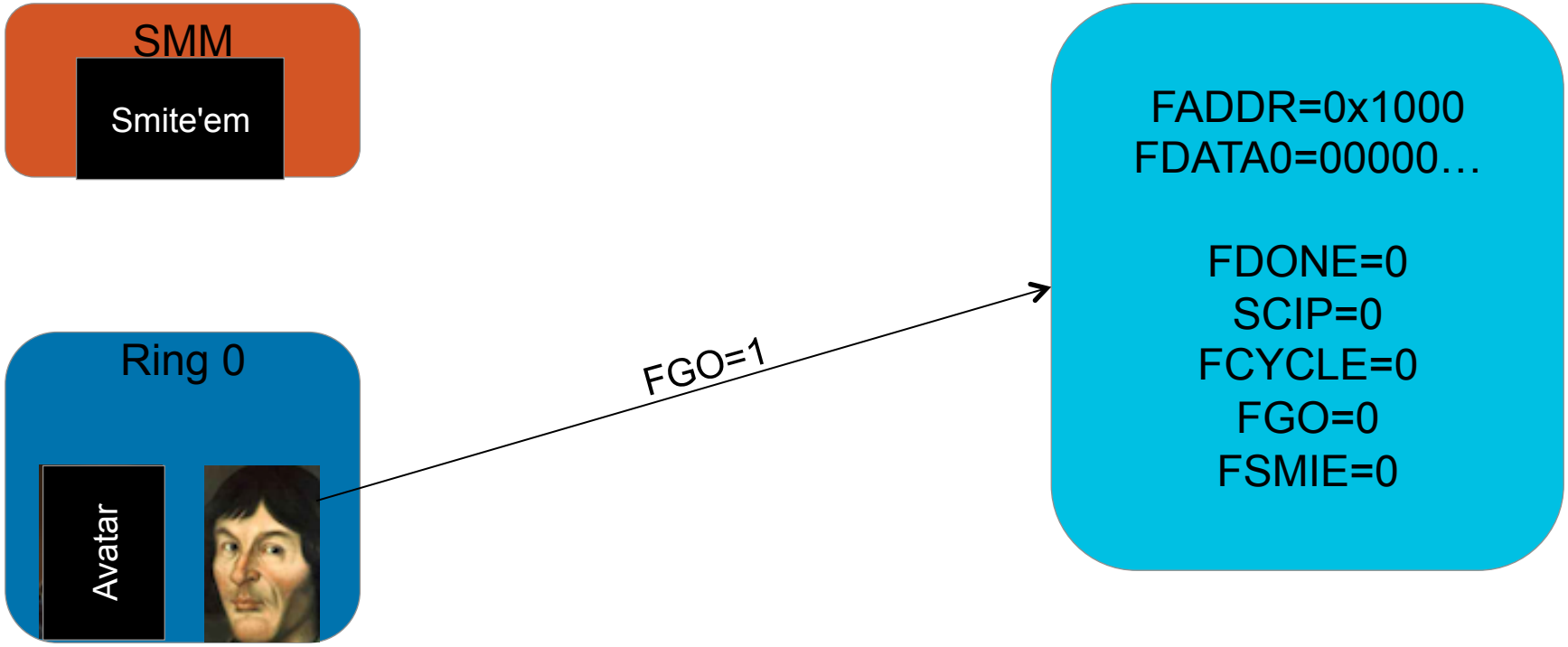
- **Waits for the SPI cycle to complete (SPI will complete independent of CPU)**
- **Compares the SPI range in FADDR register with the ranges in SPI it needs to forge**
  - FADDR address + # specified bits in HSFC equates to a range
  - Could just assume 64 bytes for simplicity
- **If the ranges overlap, it writes the forged bytes to the FDATA registers**
  - Either pick and choose which FDATA registers or assume 64 byte SPI cycles and overwrite them all

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**

Reading the flash chip in the presence of Smite'em

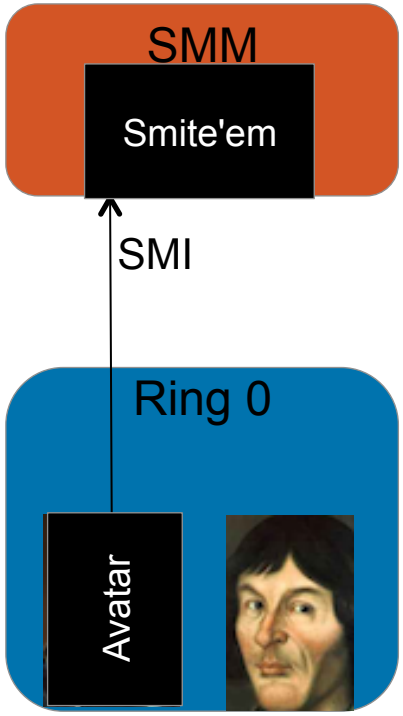


- **Then says go**

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

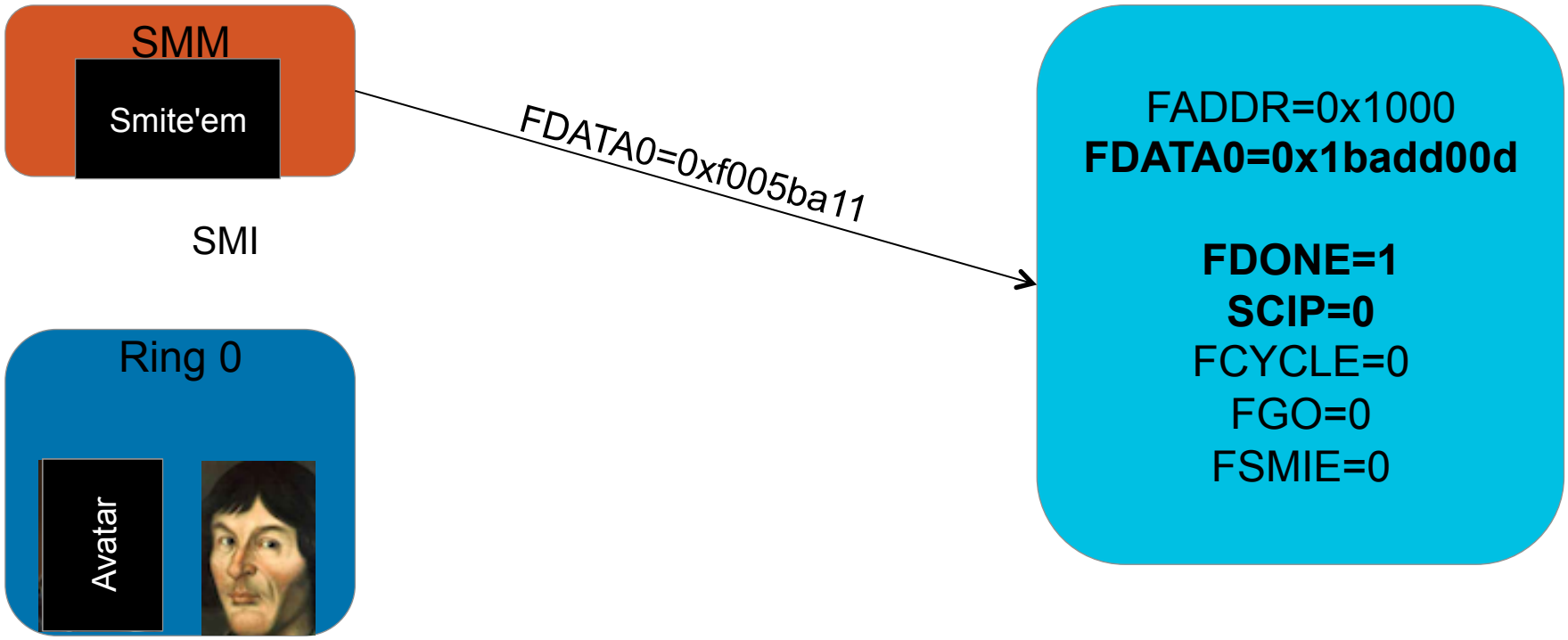


FADDR=0x1000
FDATA0=0x1badd00d

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**

Reading the flash chip in the presence of Smite'em



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

How can we defeat Smite'em?

- **We could utilize our Checkmate[19] timing-based attestation system[cite] within our Copernicus kernel driver, and incorporate SMI disabling or FSMIE disabling into the self-check as a new "untampered execution environment" check**
- **But we saw an opportunity to take a more direct path, and simultaneously get smart on some other trusted computing tech**

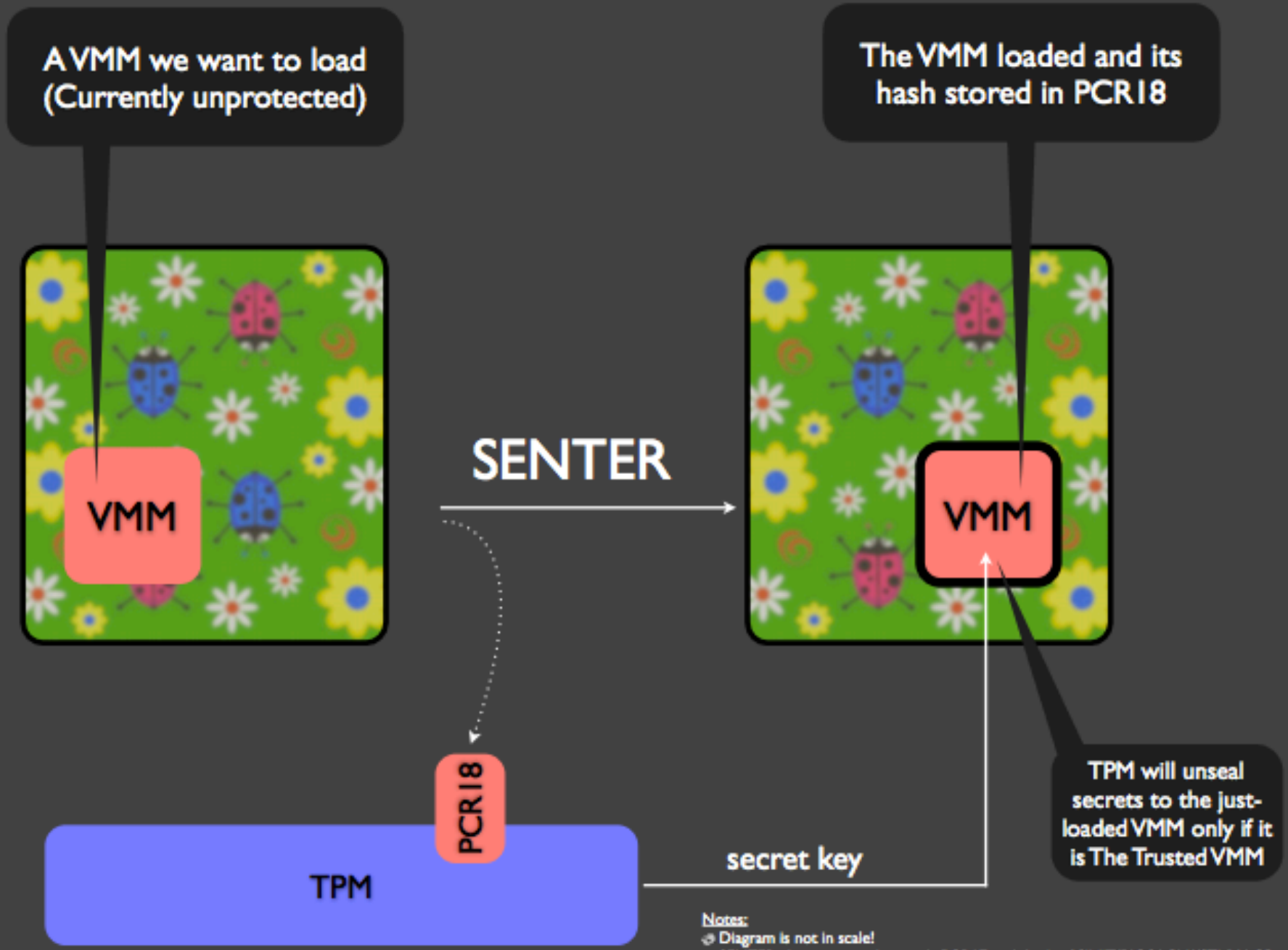
- **Smite'em lives in SMM, let's disable SMIs**
- **But its not sufficient to just disable them from an OS driver, because an attacker could just nop out our code to do so**

- **A side effect of Intel TXT is that it disables SMIs**
- **So lets learn about Intel Trusted Execution Technology (TXT)**
  - Called "Safer Mode Extensions" (SMX) in the Intel manuals

Intel Trusted Execution Technology (TXT)

- **Dynamic Root of Trust for Measurement**
- **A means to provide "late launch" trust**
  - You had a presumed-compromised system, you start TXT, and you're left in a state you setup and that you can trust

From http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20slides.pdf



How does it work?

- New Intel instruction “GETSEC”
- It’s sort of like CPUID in that it’s a single instruction that does different things based on what the value is in the EAX register at the time that it’s called
- EAX = 0; GETSEC[CAPABILITIES] = report the capabilities
- EAX = 1; GETSEC[ENTERACCS] = run authenticated code (AC)
- EAX = 2; GETSEC[EXITAC] = stop running AC
- EAX = 3; GETSEC[*SENTER*] = Start a Measured Launch Environment (MLE) – this is the main one we care about, and the source of the title of this talk
- EAX = 4; GETSEC[SEXIT] = exit MLE
- EAX = 5; GETSEC[PARAMETERS] = reports supported AC info
- EAX = 6; GETSEC[SMCTRL] = *turn on SMIs*
- EAX = 7; GETSEC[WAKEUP] = wake up sleeping processors

We're only interested in a subset

- **We have to use GETSEC[CAPABILITIES] and GETSEC[PARAMETERS] just for sanity checking purposes**
- **We mainly care about SENTER and SEXIT to start and stop our MLE**
- **We're *NOT* going to use SMCTRL or WAKEUP**
  - The whole point here is to freeze SMM code in place

But…what about ITL's attacks on TXT?!?!

- "I thought they broke TXT six ways from Sunday?!?!"
- "Doesn't this mean no one can ever trust TXT for anything ever again?!"
- No
- It just means you need to utilize TXT with awareness of the attacks
- Lets review their TXT-relevant attacks

Feb. 2009 - Attacking Intel Trusted Execution Technology – Wojtczuk & Rutkowska [5]

- **Had an at-the-time-undisclosed vulnerability to get into SMM**
- **Found that SMRAM is not measured as part of the MLE launch**
- **Once defender gets into their MLE, they're encouraged to issue GETSEC[SMCTRL] to enable SMIs as soon as possible**
  - Recall that SMM will often handle performance-critical things like motherboard fan control, so ideally you don't want to go leaving it off for a long time
- **Therefore once SMIs are enabled, and the first one is fired, the attacker regains control, within the context of the MLE**
  - They can then potentially subvert a hypervisor/OS that's being launched with tboot

Does [5] directly affect us?

- **No**
- **We're already using TXT under the assumption that we're dealing with compromised SMM**
- ***Therefore our MLE doesn't reenable SMIs until we're done with everything security-critical***
  - This is only an option for us since we're just popping up into TXT land and then back out as soon as we check a few things

Dec. 2009 - Another Way to Circumvent Intel® Trusted Execution Technology – Wojtczuk, Rutkowska, Tereshkin [6]

- One of the jobs of the SINIT modules is to sanity check that the MLE's memory is protected from DMA attacks
- However there was a bug where it read a 64 bit field as 32 bits
- The 64 bit address was what was actually protected, and the attacker just had to make sure the bottom 32 bits were the same as the MLE 32 bits, and the sanity check would pass

Does [6] directly affect us?

- **No**
- **Intel released patched SINIT modules for all their chipsets [cite/link]**
- **We use the latest patched one**
- **But also because we don't use the VT-d protection, we use the TXT special "DMA Protected Region", which as Intel says is**
  - This is only an option for us because we're loading a small amount of code. If we were trying to launch/protect a hypervisor, we would have to use VT-d

Dec. 2011 - Exploring new lands on Intel CPUs (SINIT code execution hijacking) - Wojtczuk & Rutkowska [7]

- A critical component of TXT is the use of the ACMs.
- The ACM which is used during an SENTER is the “SINIT” ACM binary blob that you have to map into memory and give the base address to SENTER in register EBX
- SINIT sanity checks that the environment is correctly setup and conducive to
- But at the end of the day it’s still just signed x86 code!
- x86 code that does parsing!
- x86 code that does parsing that ITL found a buffer overflow in!
- :O

Dec. 2011 - Exploring new lands on Intel CPUs (SINIT code execution hijacking) - Wojtczuk & Rutkowska [7]

- **The last, and arguably the best, of all of ITL's attacks ever**
- **The buffer overflow occurs when SINIT is parsing the DMA Remapping (DMAR) ACPI (Advanced Configuration and Power Interface) table, which is set up by BIOS**
  - In this way there is an unfortunate dependancy of the DRTM on the SRTM
  - We've shown the flawed nature of current SRTMs in our BIOS Chronomancy work [18]

Does [7] affect us?

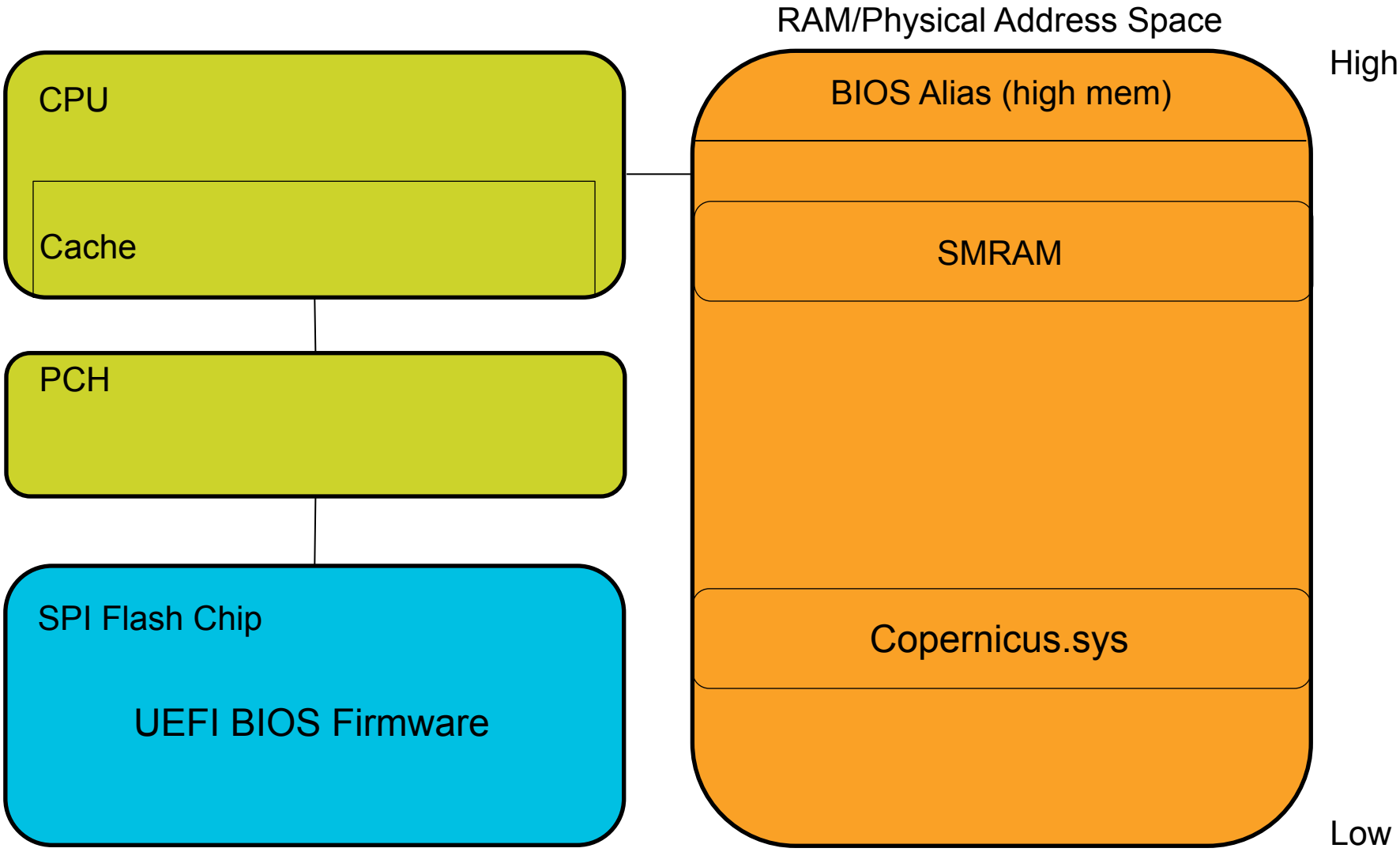
- No
- Intel released patched SINIT modules for all their chipsets [cite/link]
- We use the latest patched one
- BUT…

Purging the sin in SINIT

- **All we can do is pray there are no more bugs in the SINIT code**
  - Or evaluate it ourselves… but Intel says that's not allowed in the EULA…"yay trusted computing" :-/
  - An SINIT module is maybe around 5k instructions based on its size. Hopefully there won't be *that* many more errors ;)
    - Compare this to our timing-based[18][19] code's root of trust, which is about 60 instructions per block, 8 variant blocks, and open source
- **Obviously just hoping that there are no more bugs and the attacker can't get in is anathema to our stated goal with our timing-based attestation work, that we want to assume the attacker is at the same privilege level as us**
- **But we'll go with it for now as a risk we have no choice but to accept in order to play with this technology**

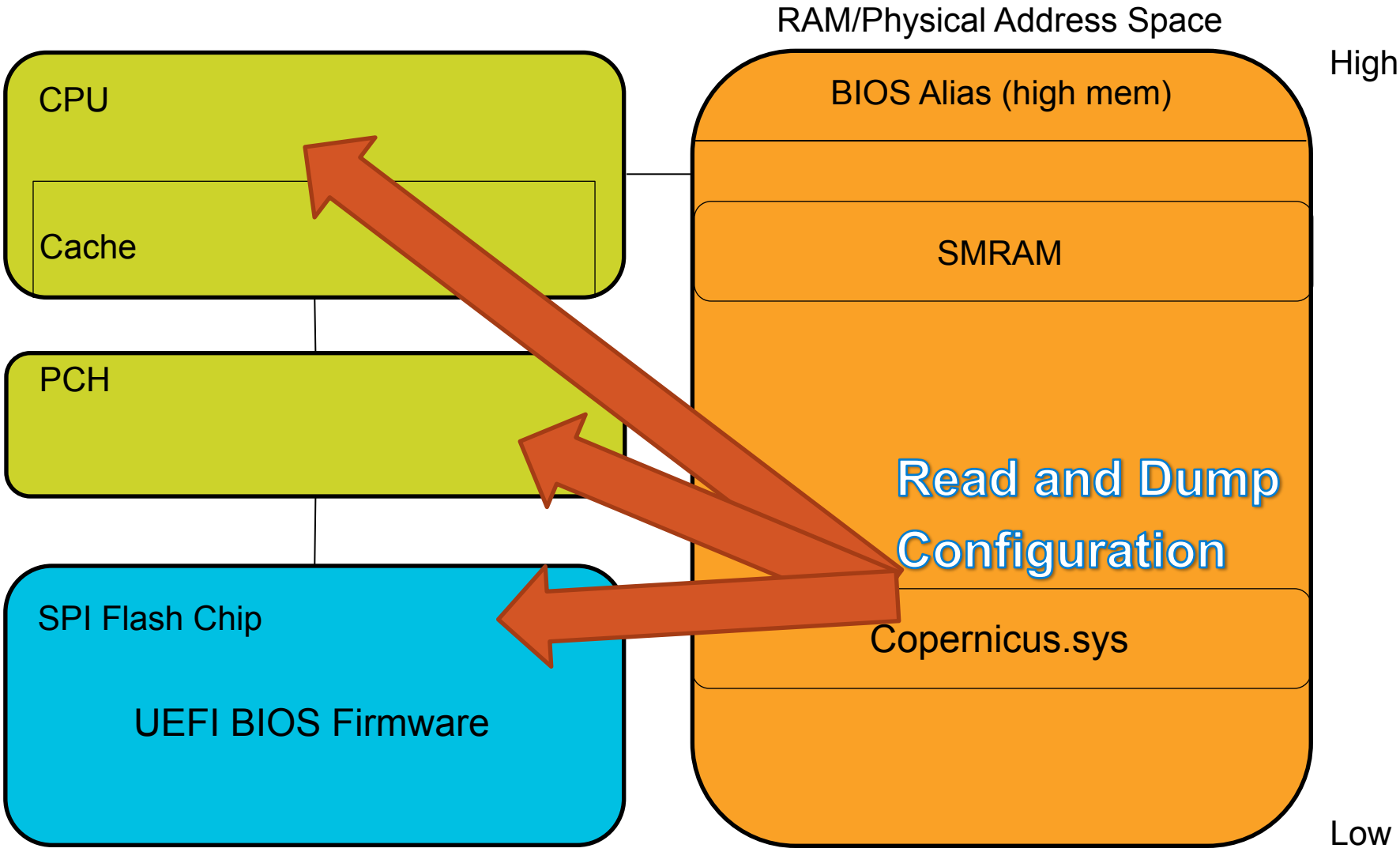
Lets build this thing!

Copernicus 1 Architecture

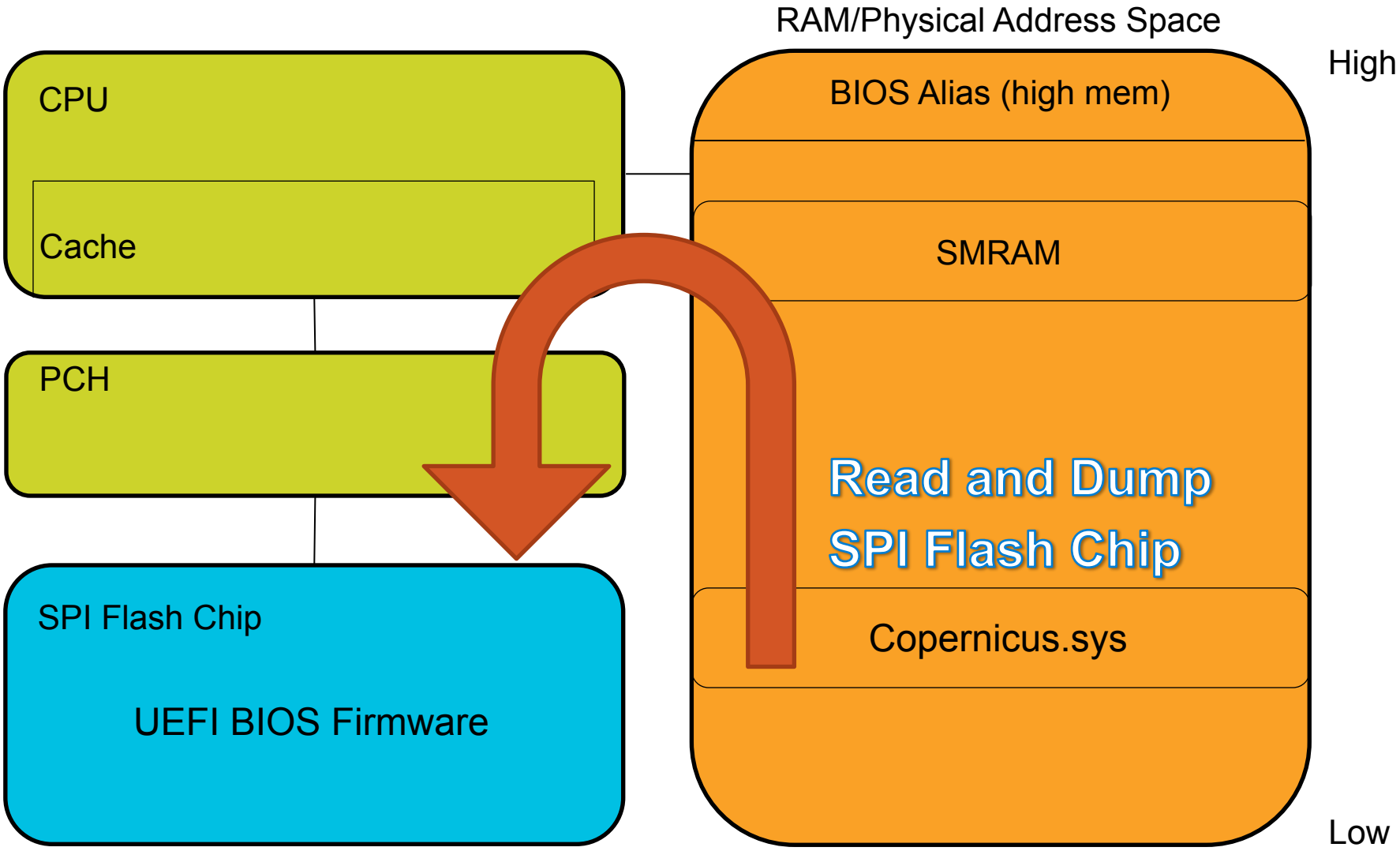


MITRE

Copernicus 1 Architecture

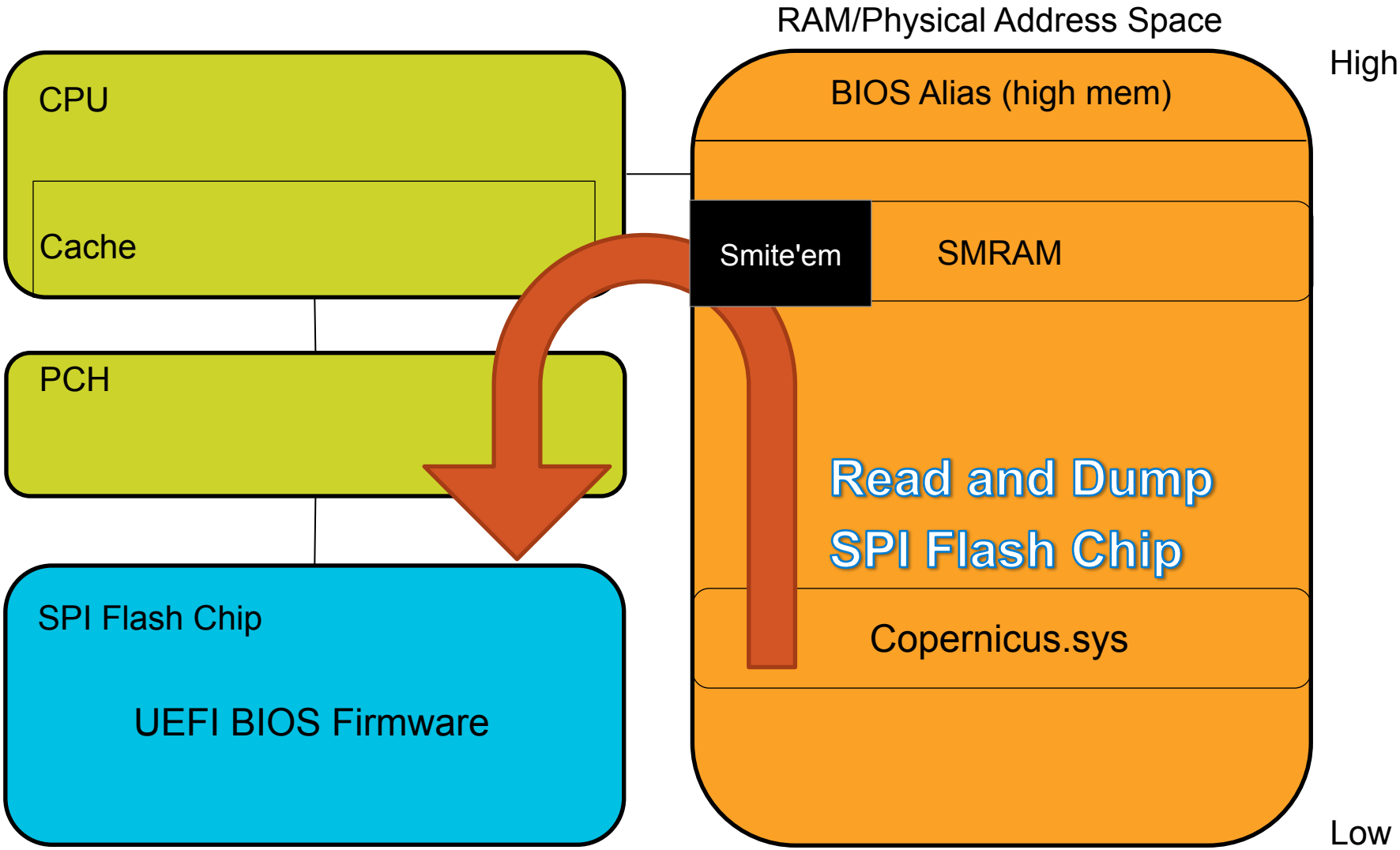


Copernicus 1 Architecture

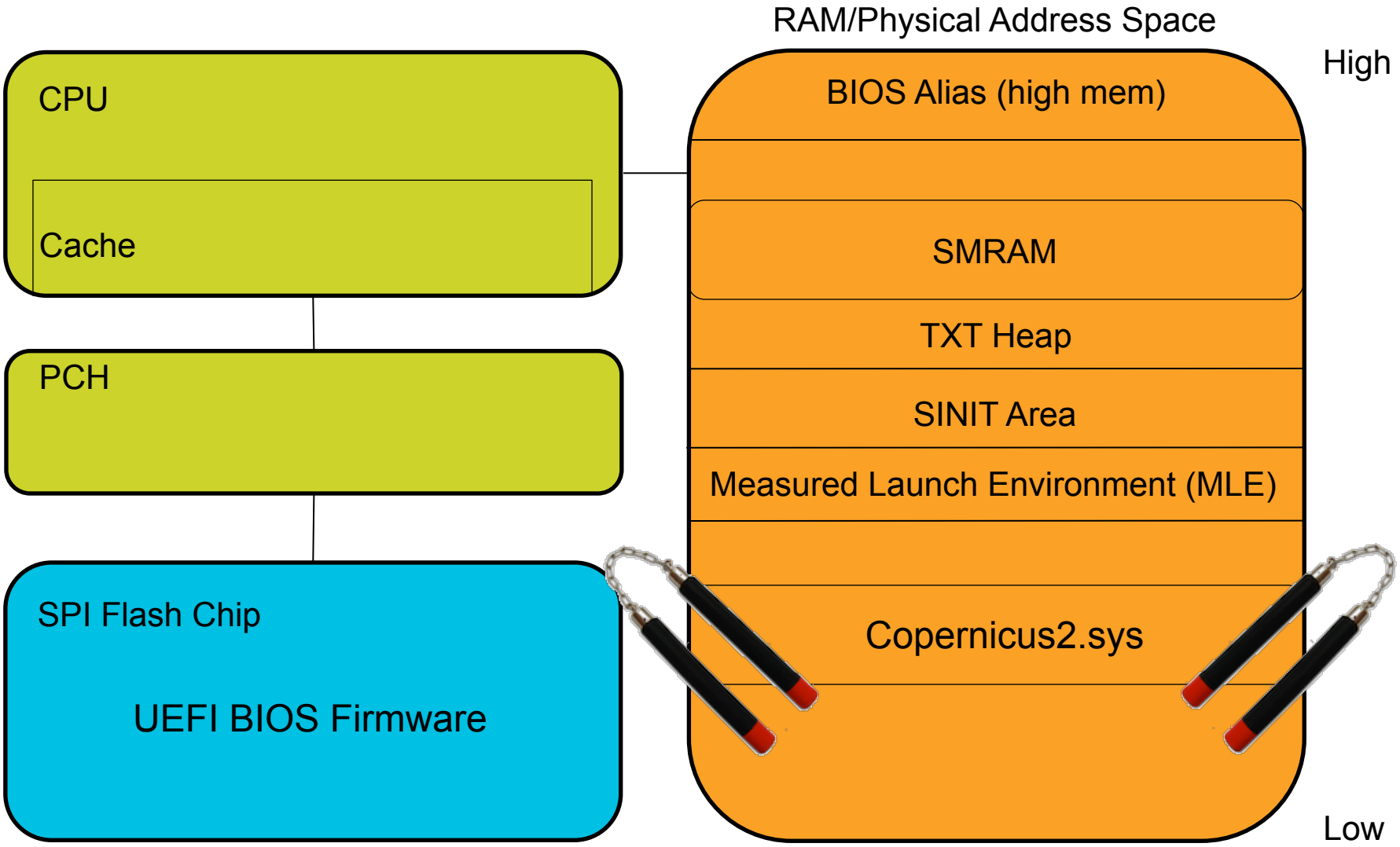


MITRE

Smite'em Attacks!



Copernicus 2 Architecture

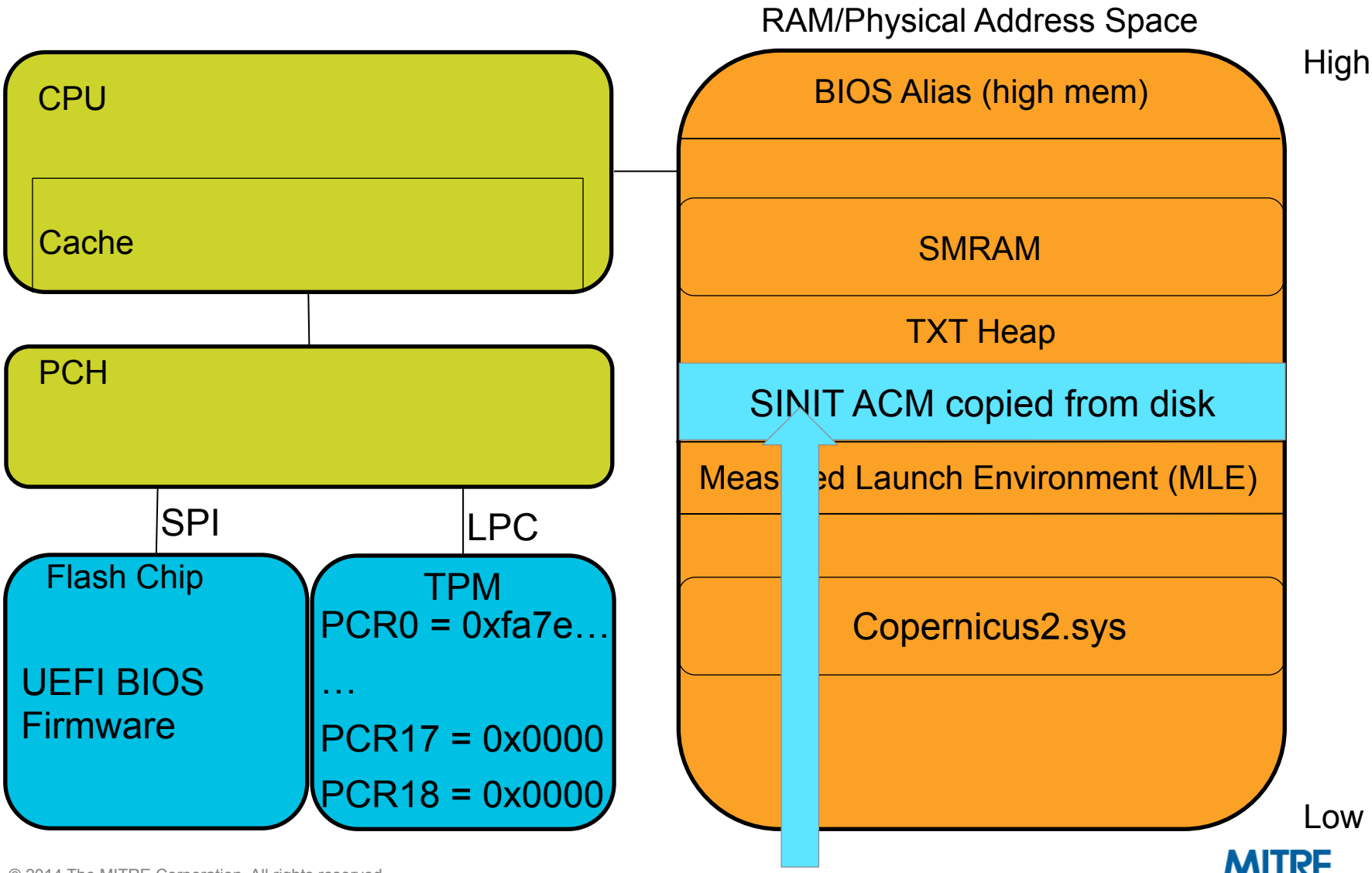


MITRE

Overall behavior

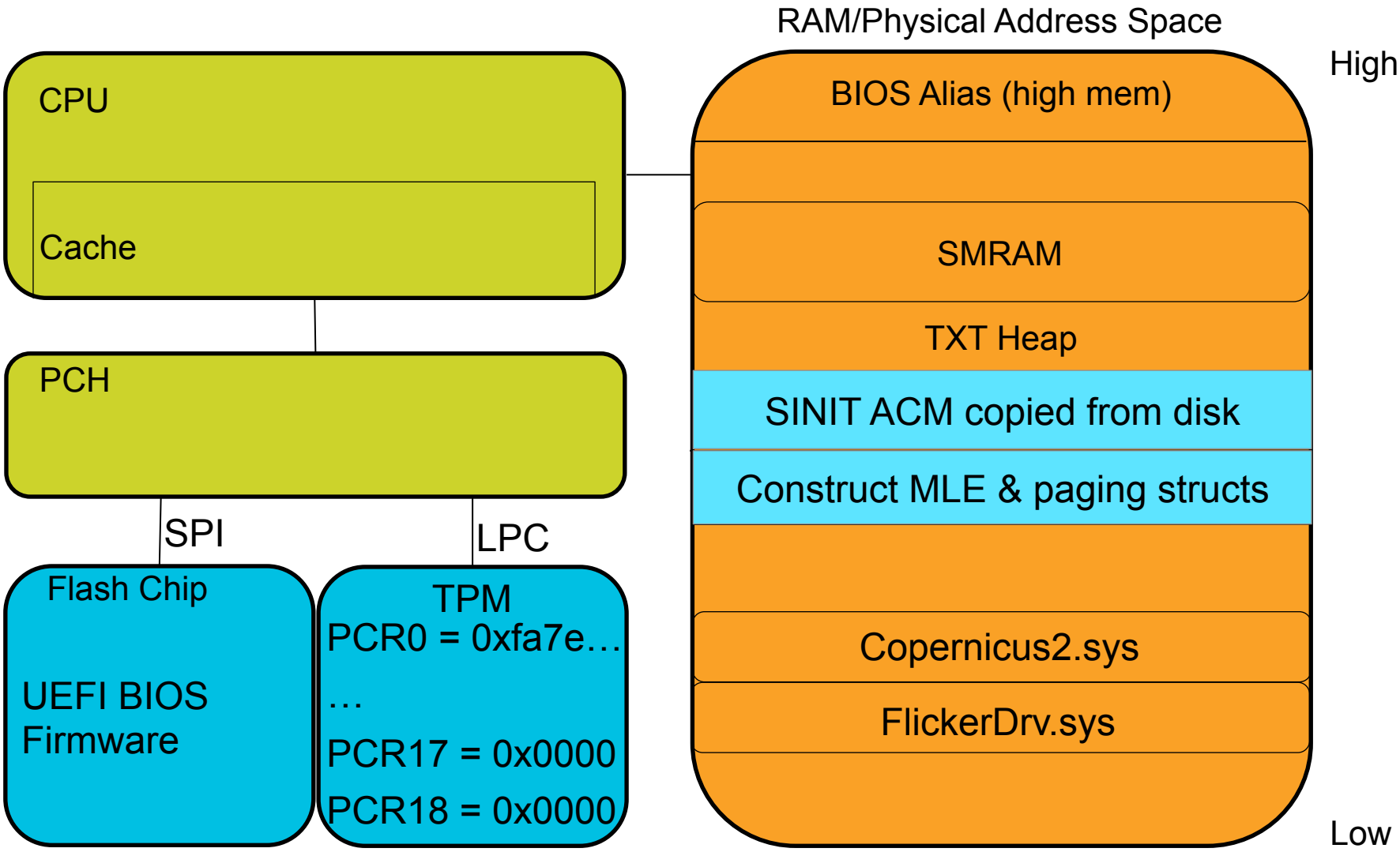
- **Initial actions:**
  - Provision TPM key for later signature verification
  - Load Flicker driver, pass MLE code to Flicker, tell it to start
- **MLE actions:**
  - Read config info, place text in buffer, extend buffer into PCR 18
  - Read BIOS 0x10000 at a time, place into buffer, extend buffer into PCR 18
  - SEXIT
- **Actions upon resume:**
  - Perform equivalent config and BIOS reads from copernicus2.sys, write to disk. Also dump TXT heap for reconstructing PCRs
  - Get TPM Quote of PCR 17, 18, 19, verify signature, write to disk

Copernicus 2 Architecture



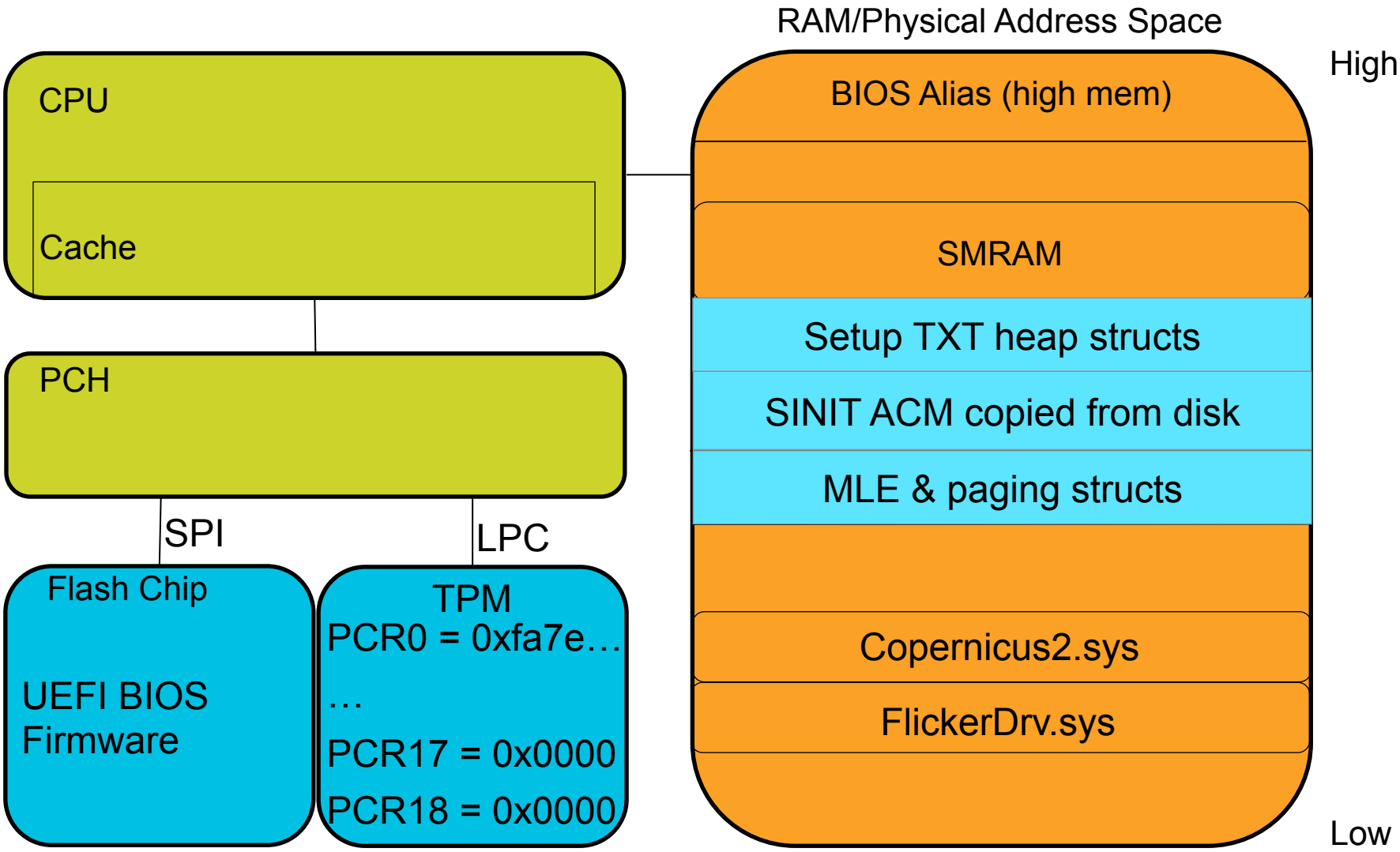
MITRE

Copernicus 2 Architecture



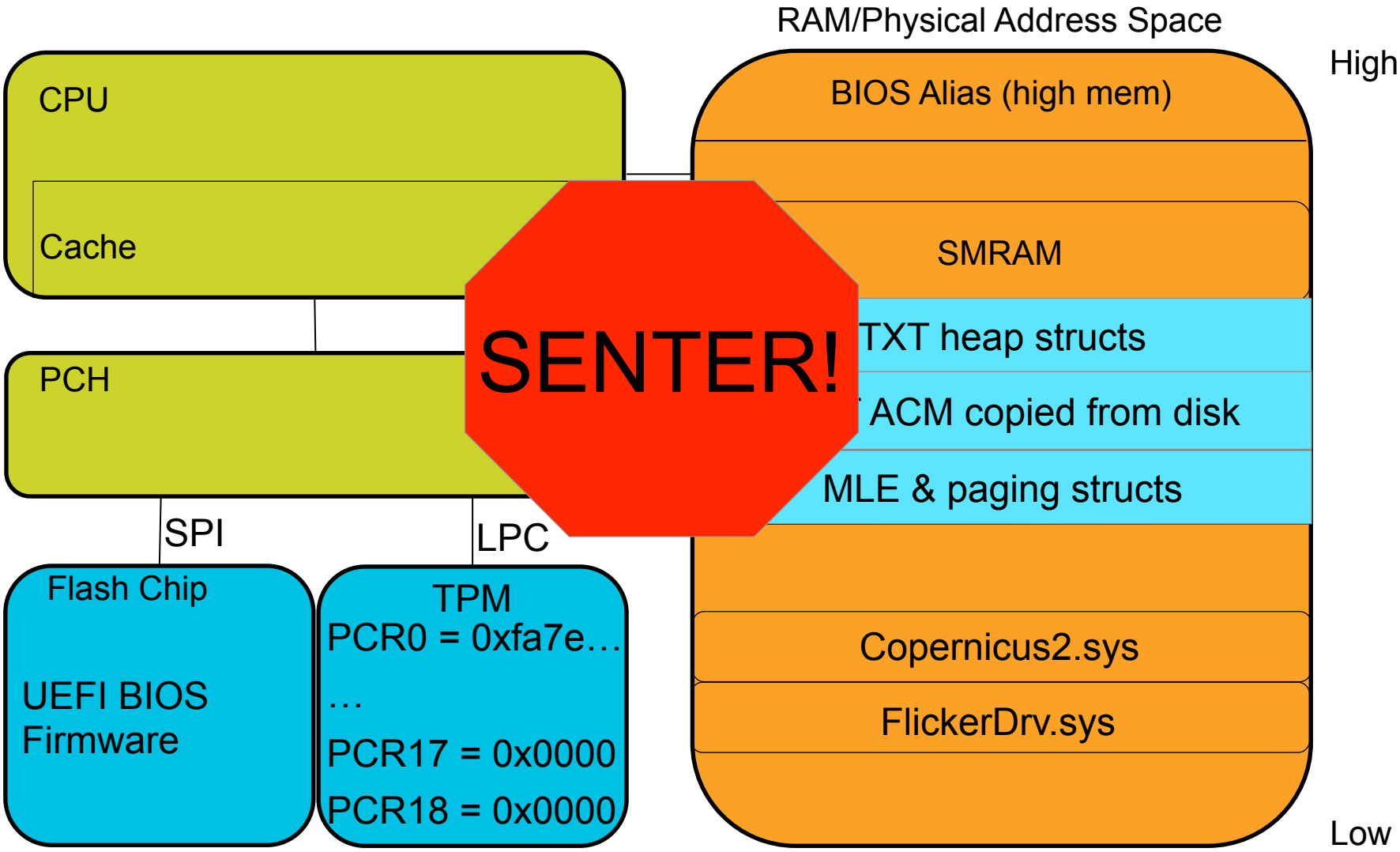
MITRE

Copernicus 2 Architecture



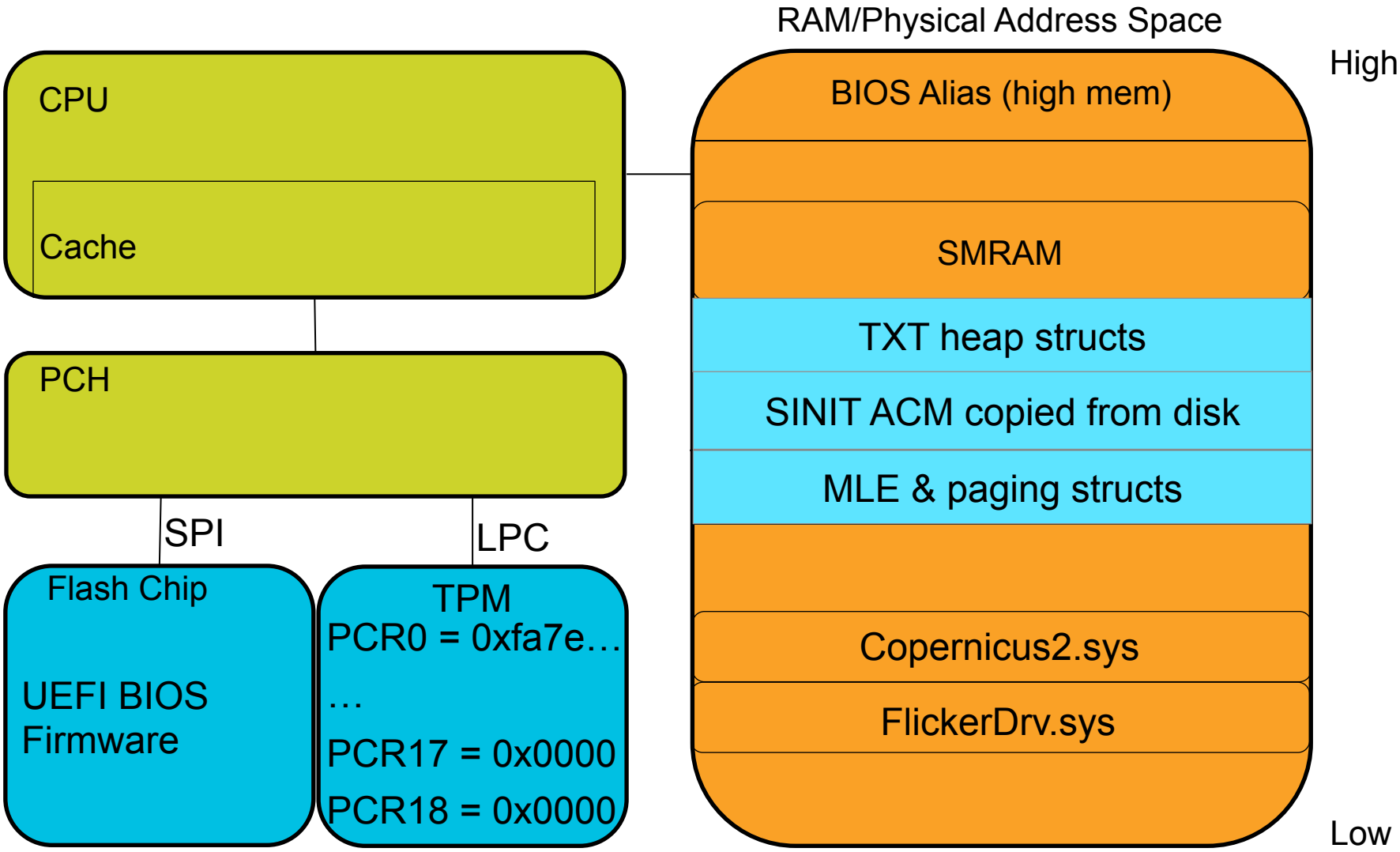
MITRE

Copernicus 2 Architecture



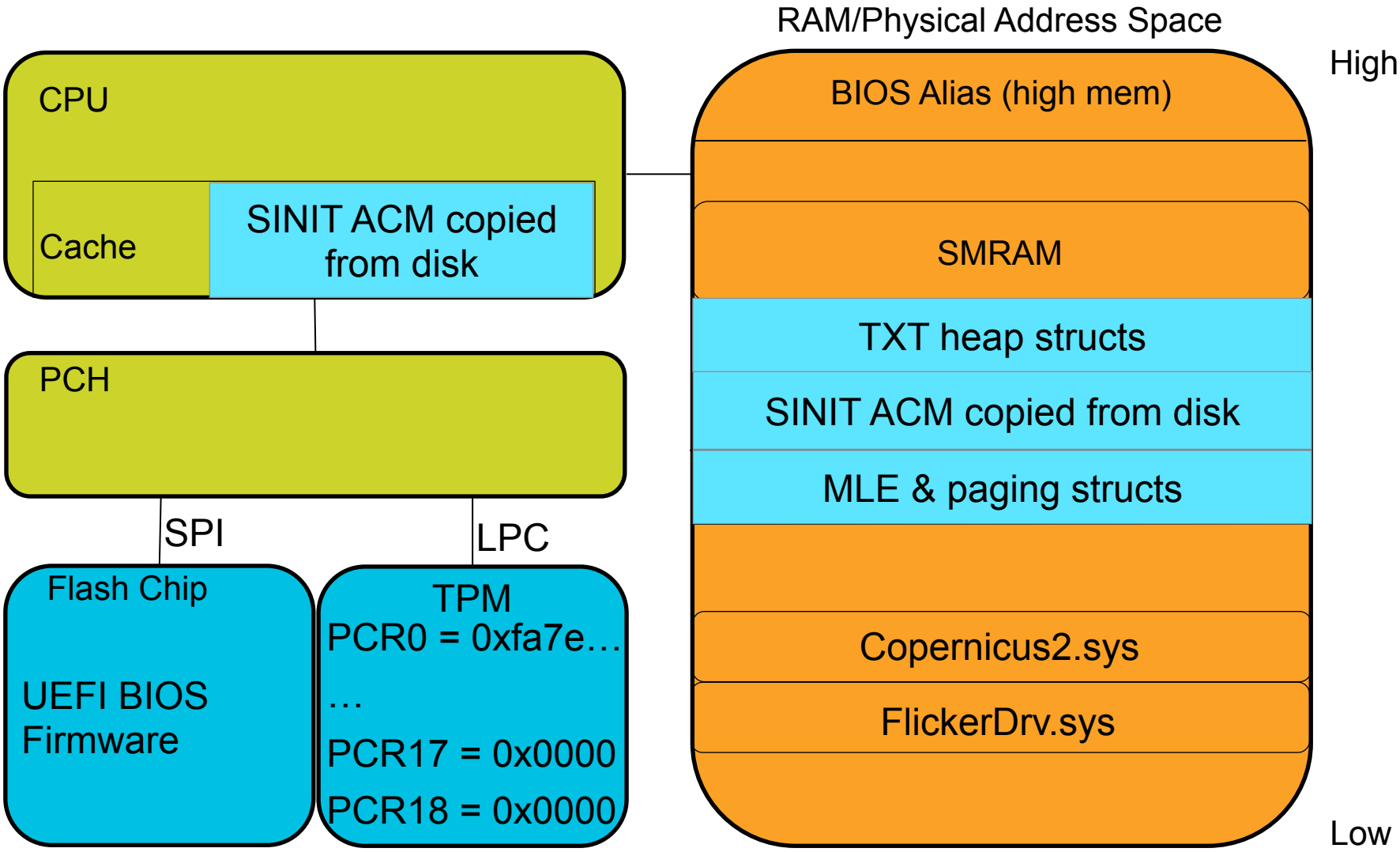
MITRE

Copernicus 2 Architecture



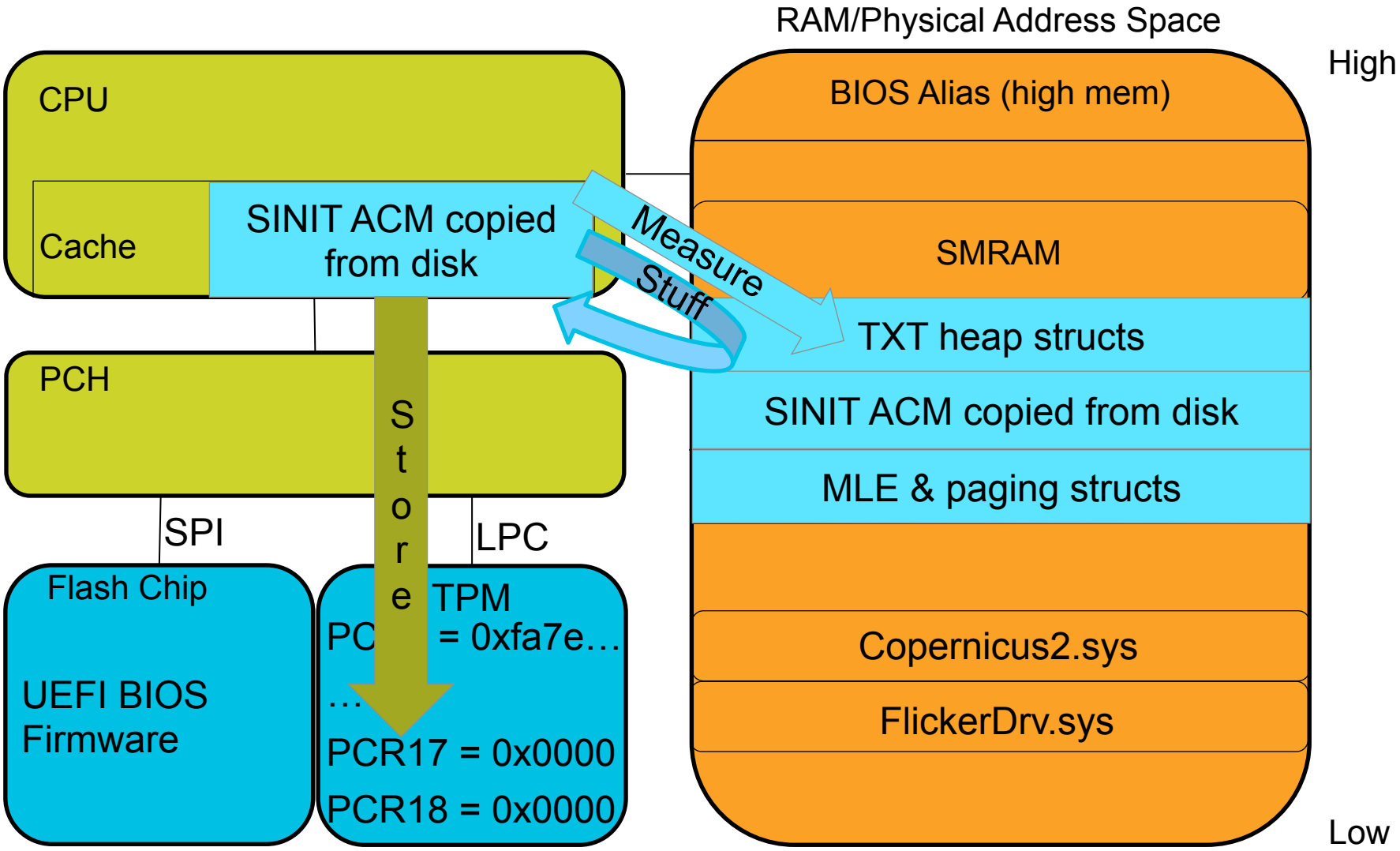
MITRE

Copernicus 2 Architecture



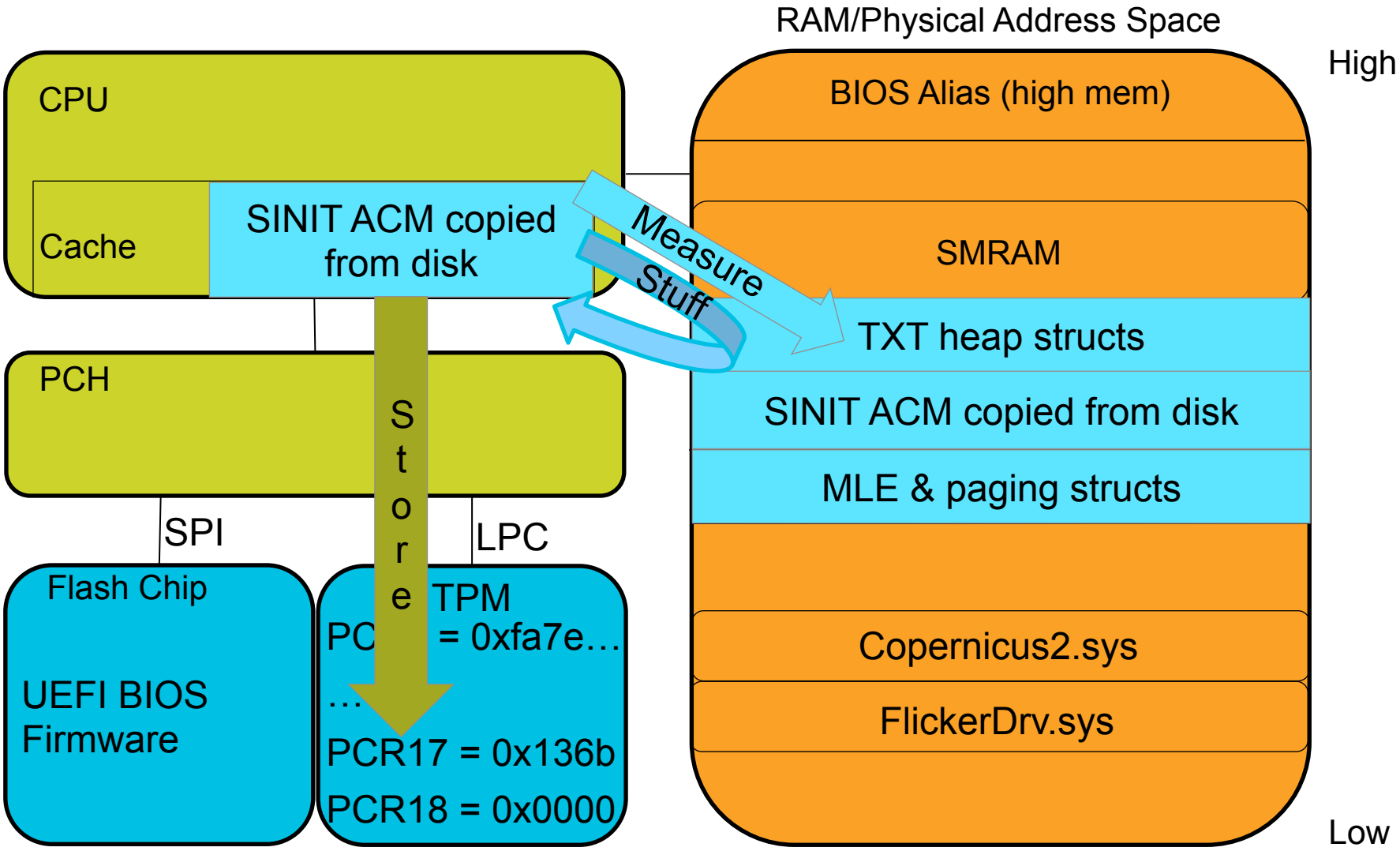
MITRE

Copernicus 2 Architecture



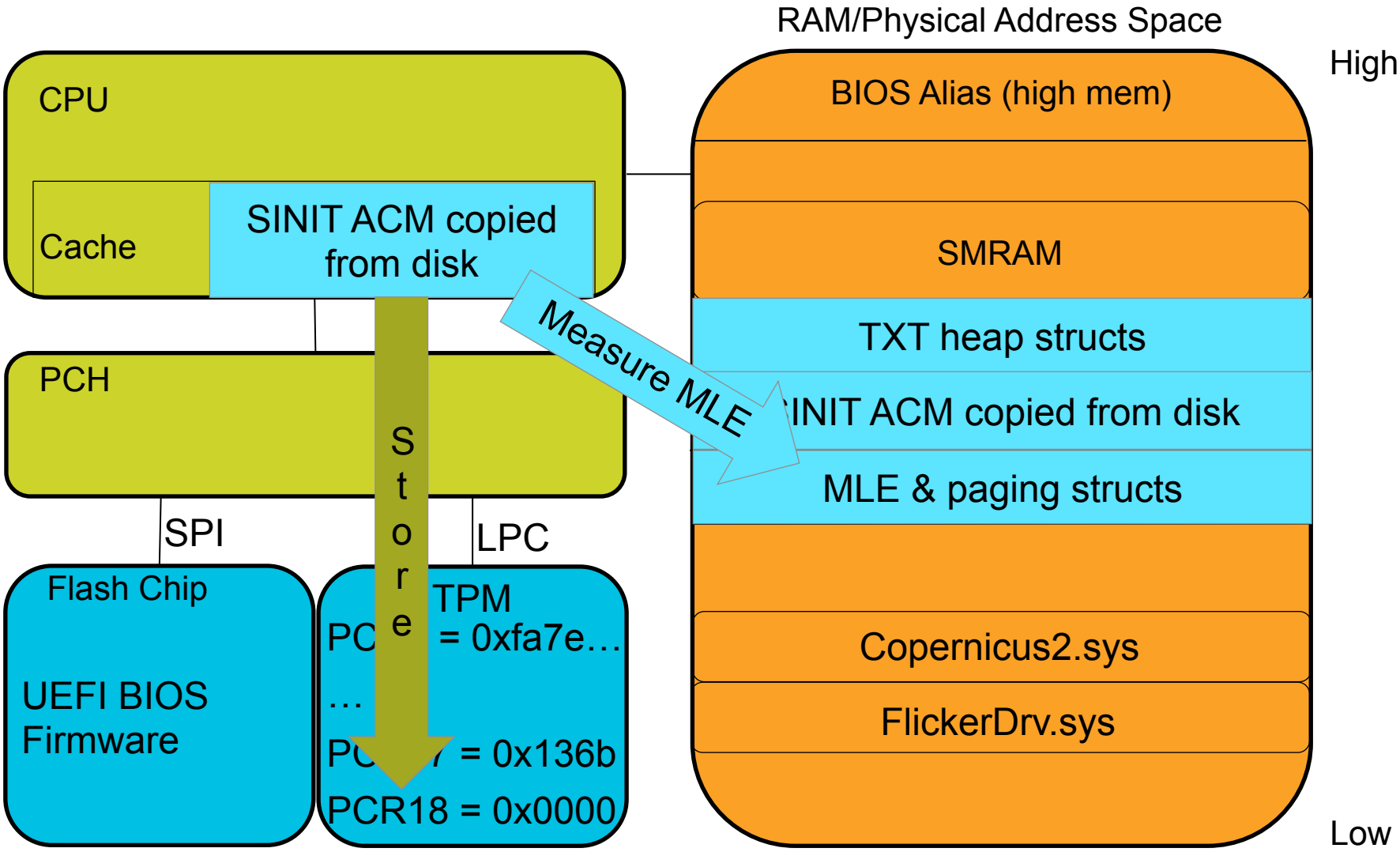
MITRE

Copernicus 2 Architecture



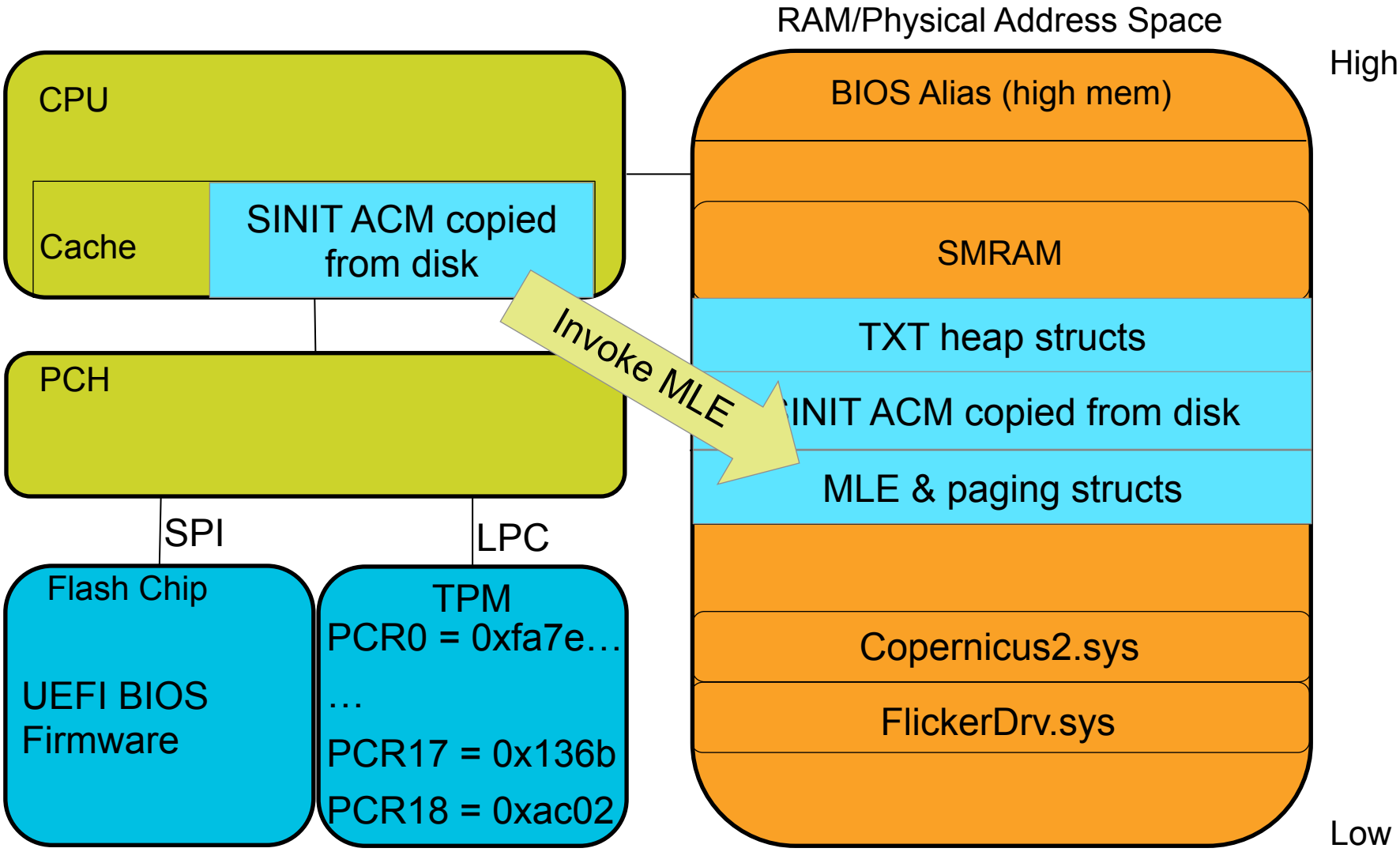
MITRE

Copernicus 2 Architecture



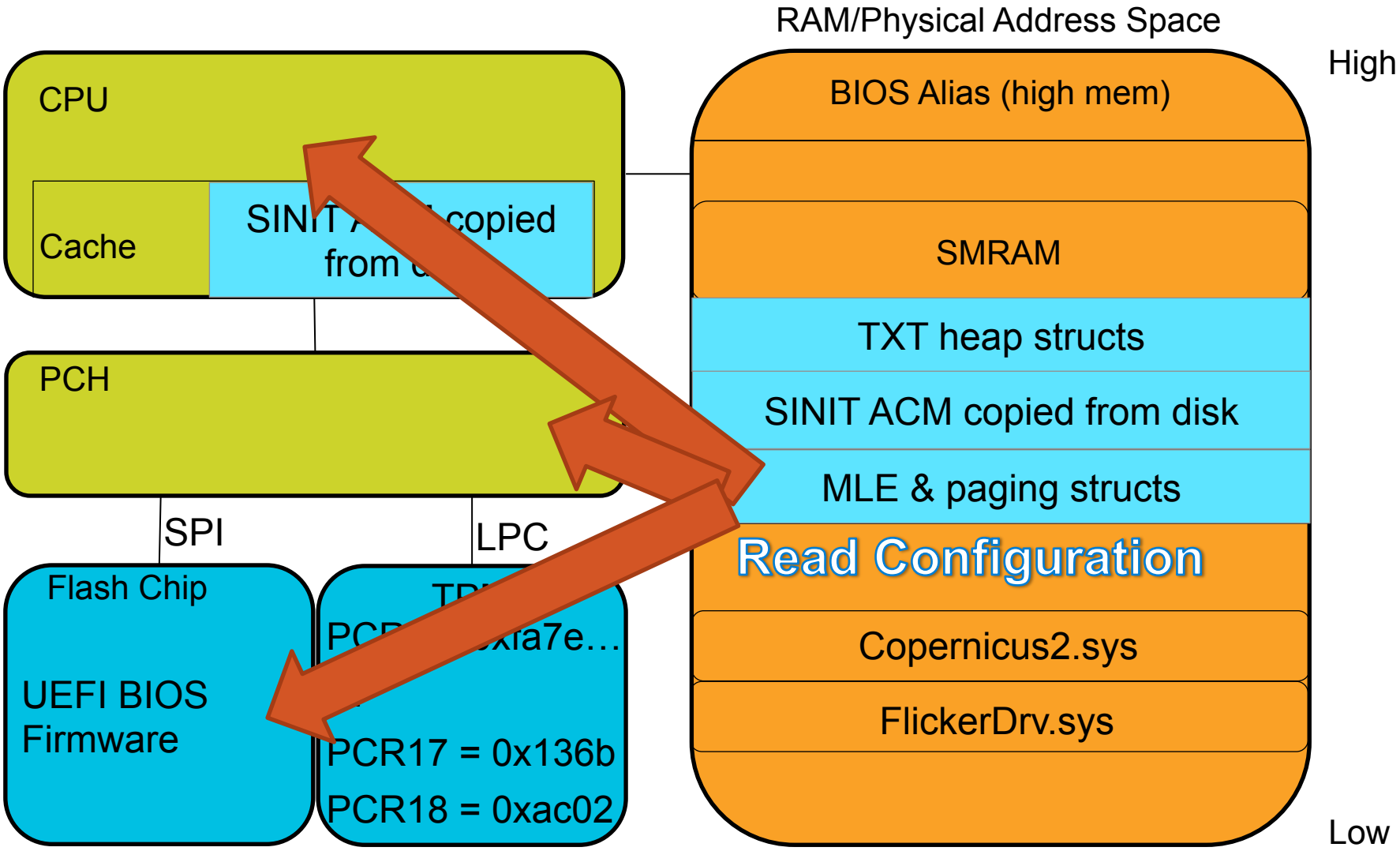
MITRE

Copernicus 2 Architecture



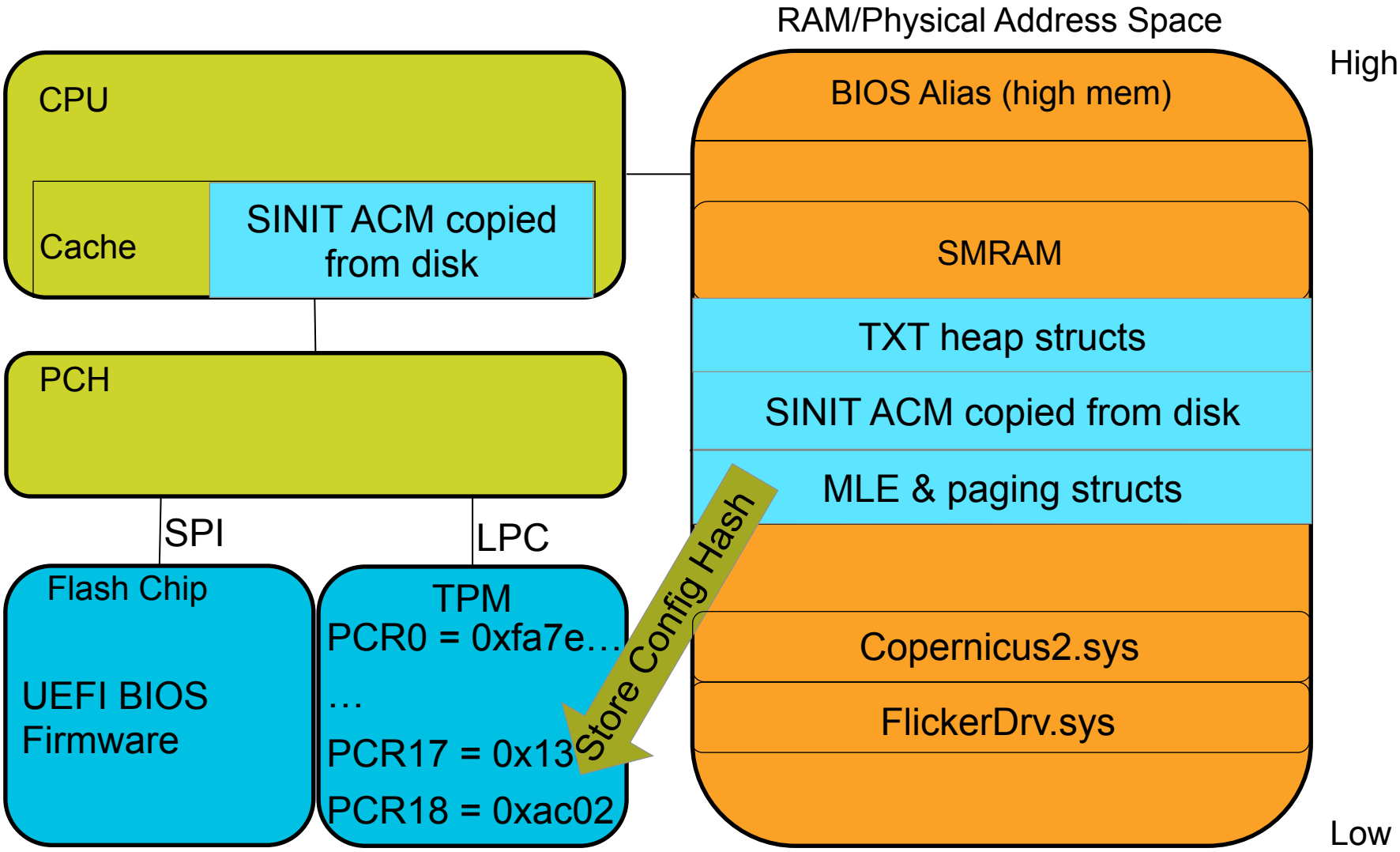
MITRE

Copernicus 2 Architecture



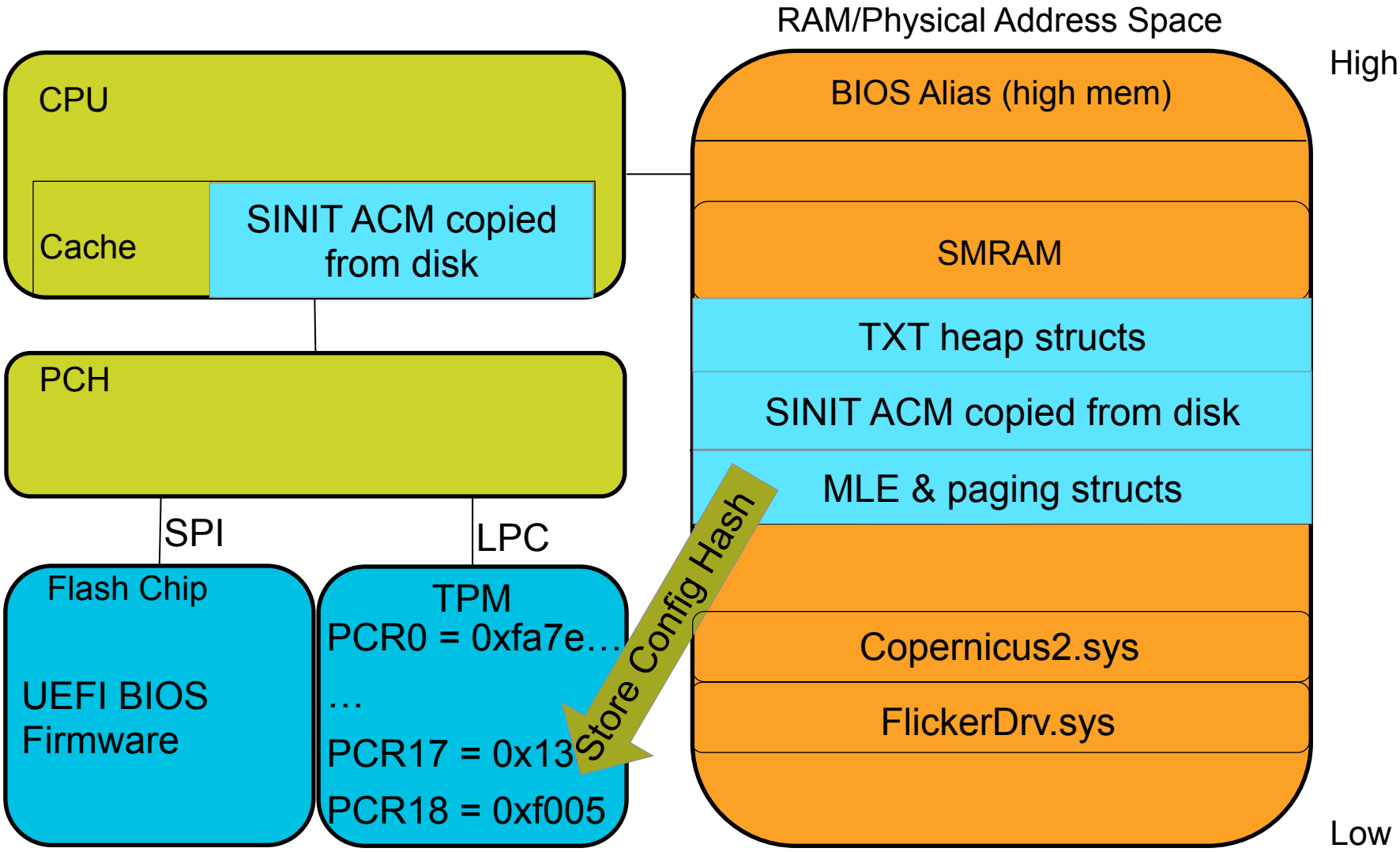
MITRE

Copernicus 2 Architecture



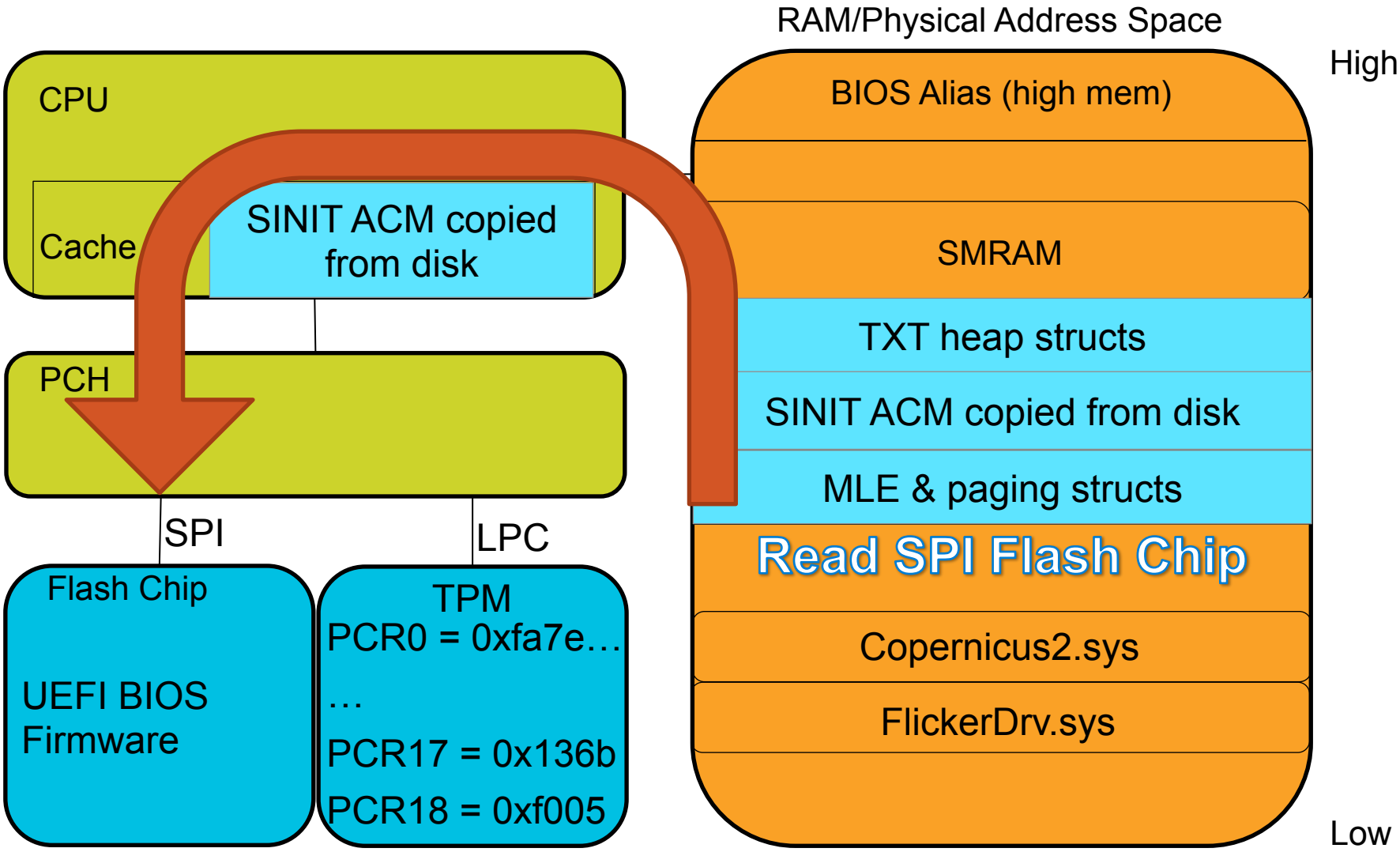
MITRE

Copernicus 2 Architecture



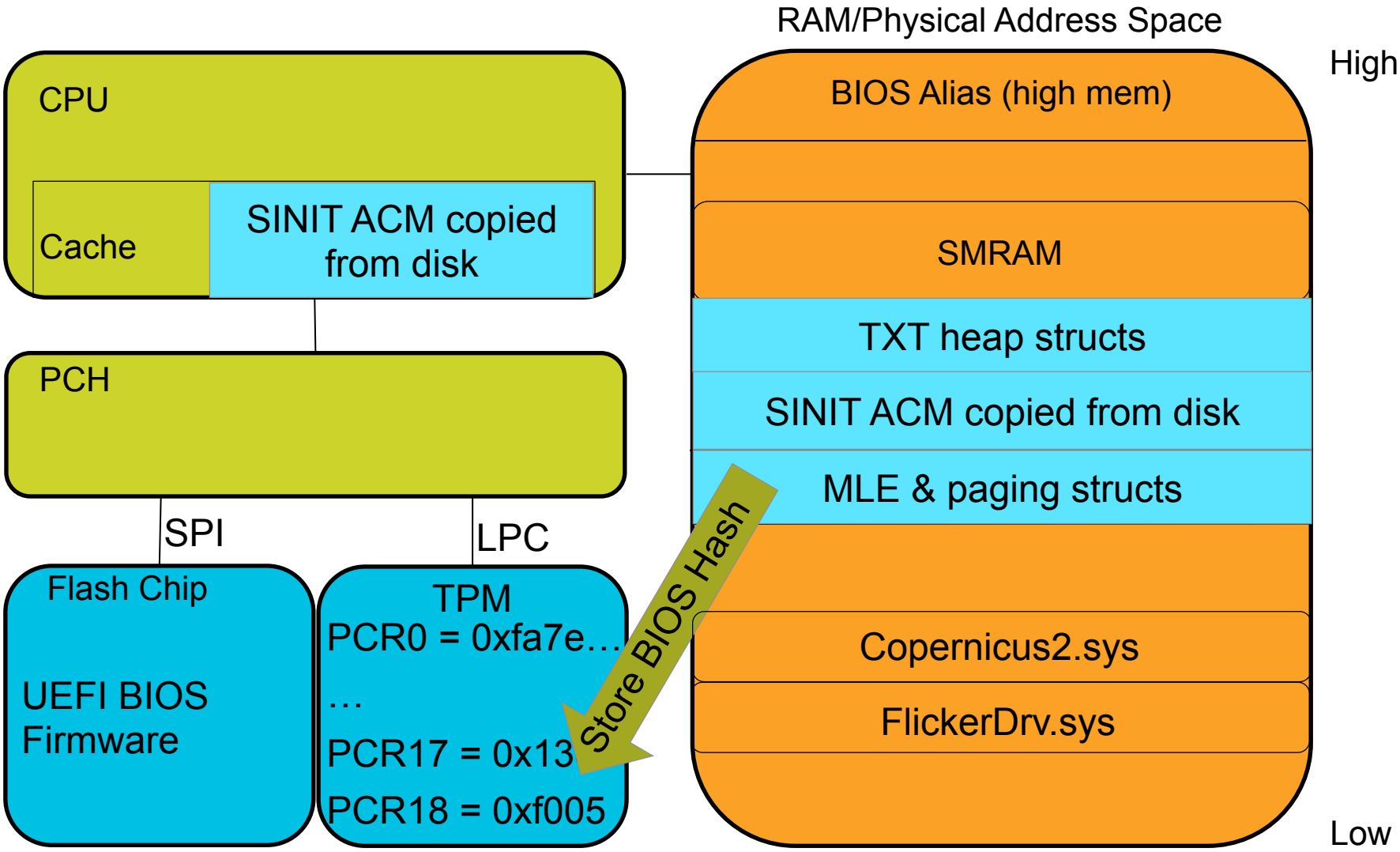
MITRE

Copernicus 2 Architecture



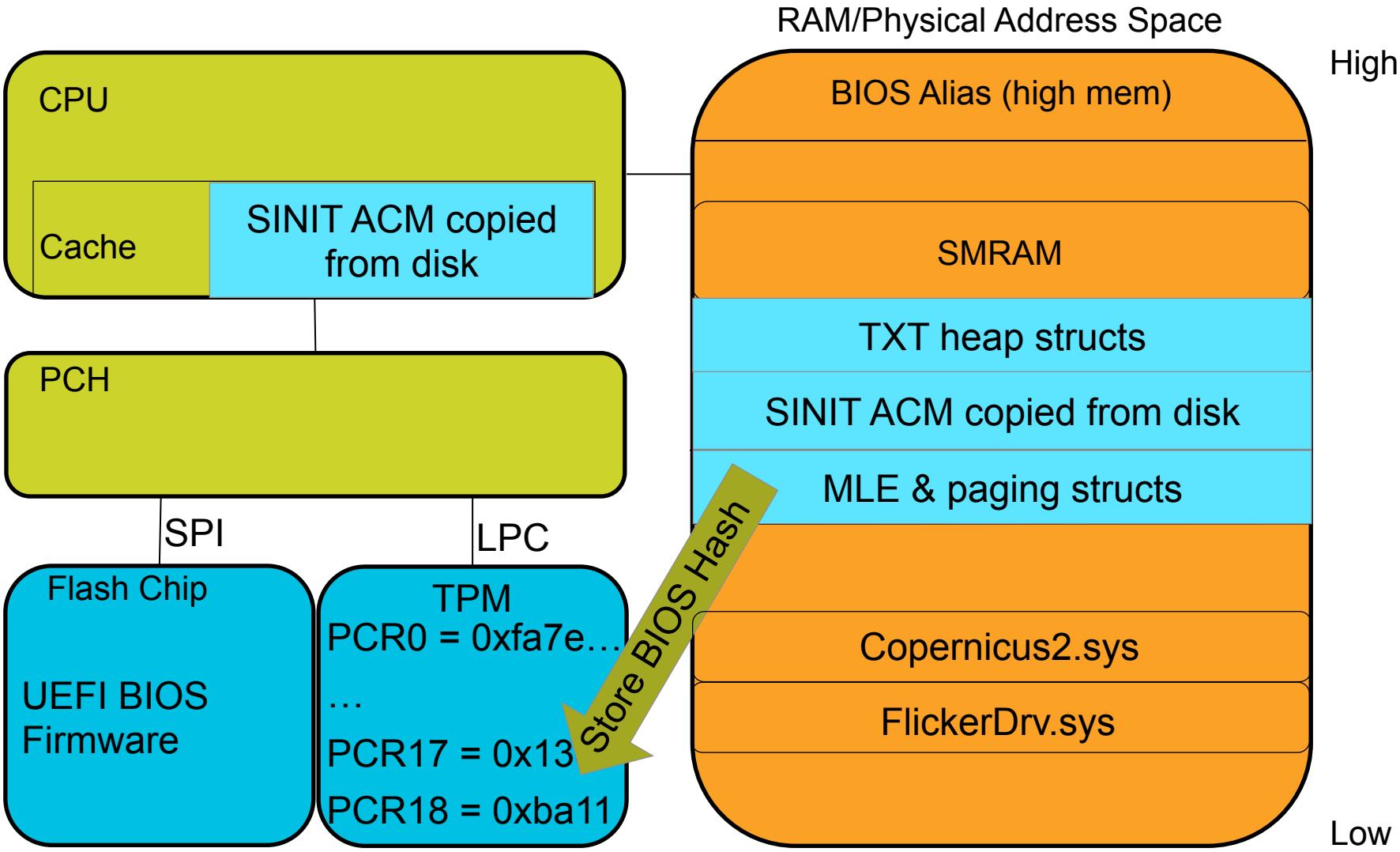
MITRE

Copernicus 2 Architecture



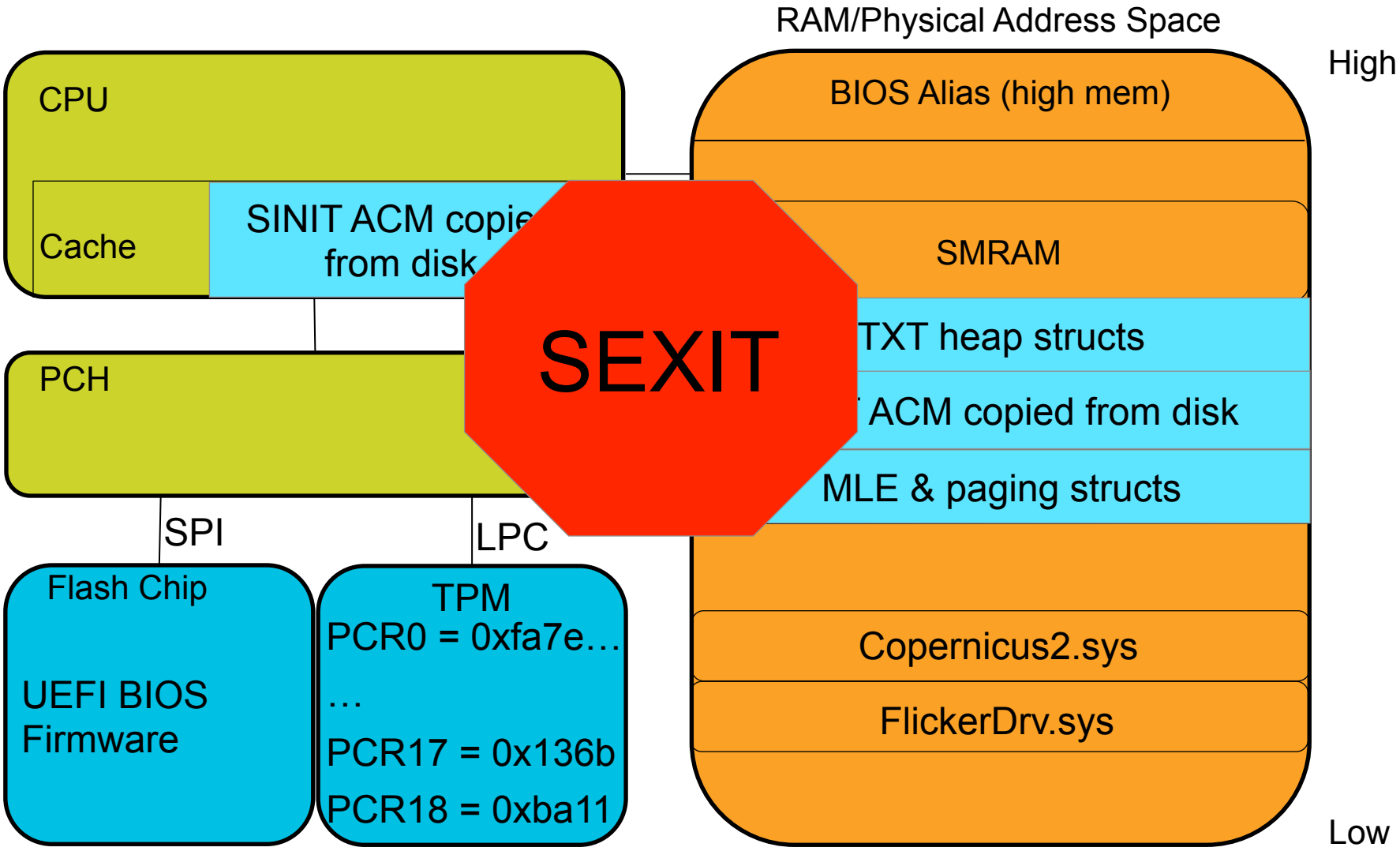
MITRE

Copernicus 2 Architecture



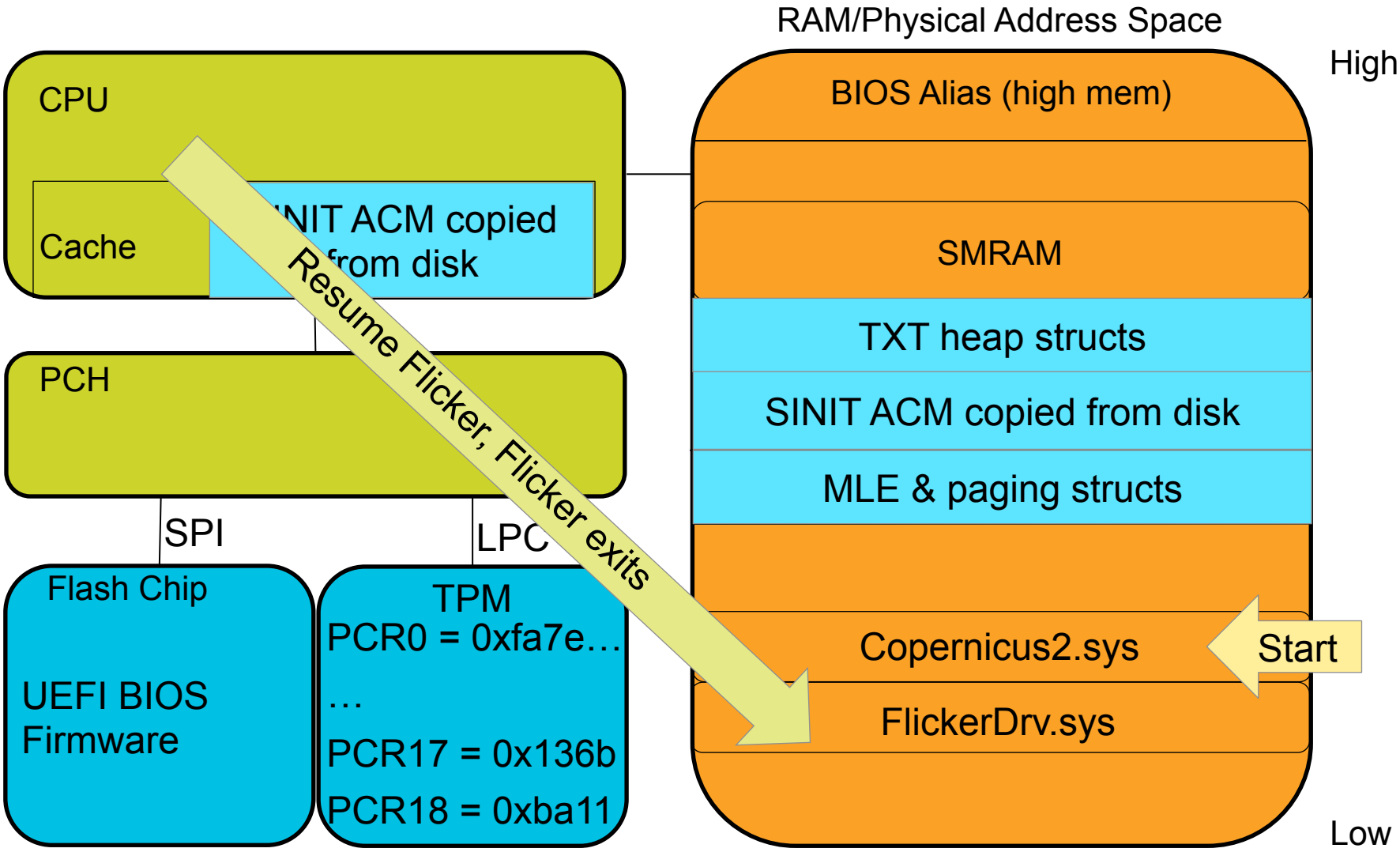
MITRE

Copernicus 2 Architecture



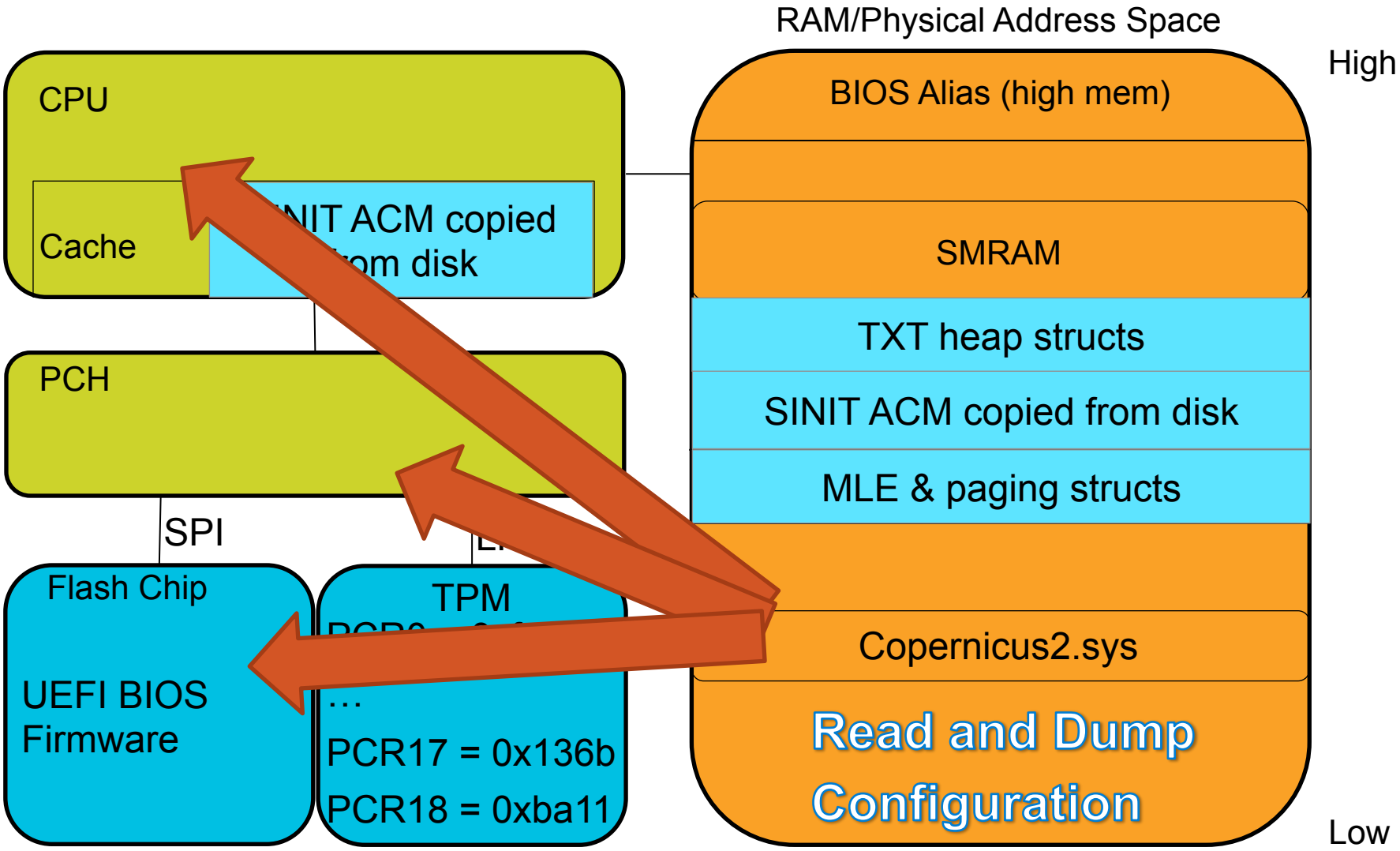
MITRE

Copernicus 2 Architecture



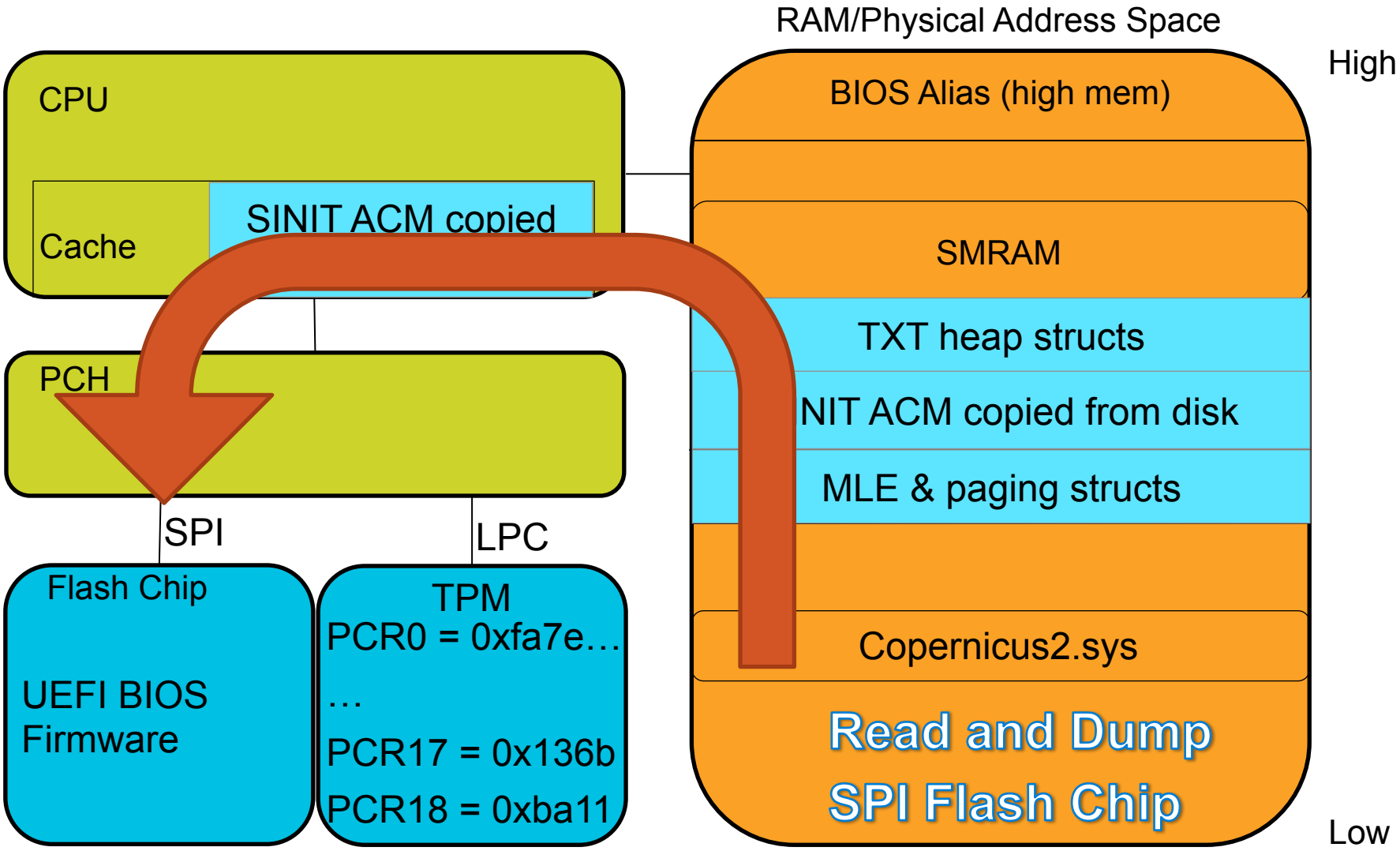
MITRE

Copernicus 2 Architecture



MITRE

Copernicus 2 Architecture



MITRE

Done!

- **We don't need to actually write the data to disk from within the MLE. We just need to collect it and hash it into the TPM PCRs**
  - This is good for cross-OS support, and for performance, that we're not sitting in the MLE with SMIs disabled for extra time
- **As for the extra steps and effort included by making Copernicus 2 "Trusted"…This is the whole reason we chose to release Copernicus 1 as a "best effort" system to start with.**
- **Trusted Computing is HARD yo!**
  - Put another way, anyone who's not going to this level of effort is probably feeding you bogus results
- **"Other evaluators of TXT have made the same comment, 'Oh, this is complex.' The first question back to them has been, 'what should we remove?' The answer has always been 'I do not see anything you can remove.'"**
  - David Grawrock, Intel, "Dynamics of a Trusted Computing System: A Building Blocks Approach", Chapter 11

Verifying measurements

- **Validate signature on TPM PCR 17 & 18 Quote**
- **Confirm PCR18 =**
- **SHA1(SHA1(SHA1(SHA1(SHA1(0_{20} | MleHash) | config) | BIOS)**
  - Can slightly differ depending on the TXT version, e.g. it could actually be using SHA256 in the MleHash
- **Confirm PCR17 is derived from the fields and values given in the "PCR 17" section (1.9.1.1 in the June 2013 TXT sw dev guide)**

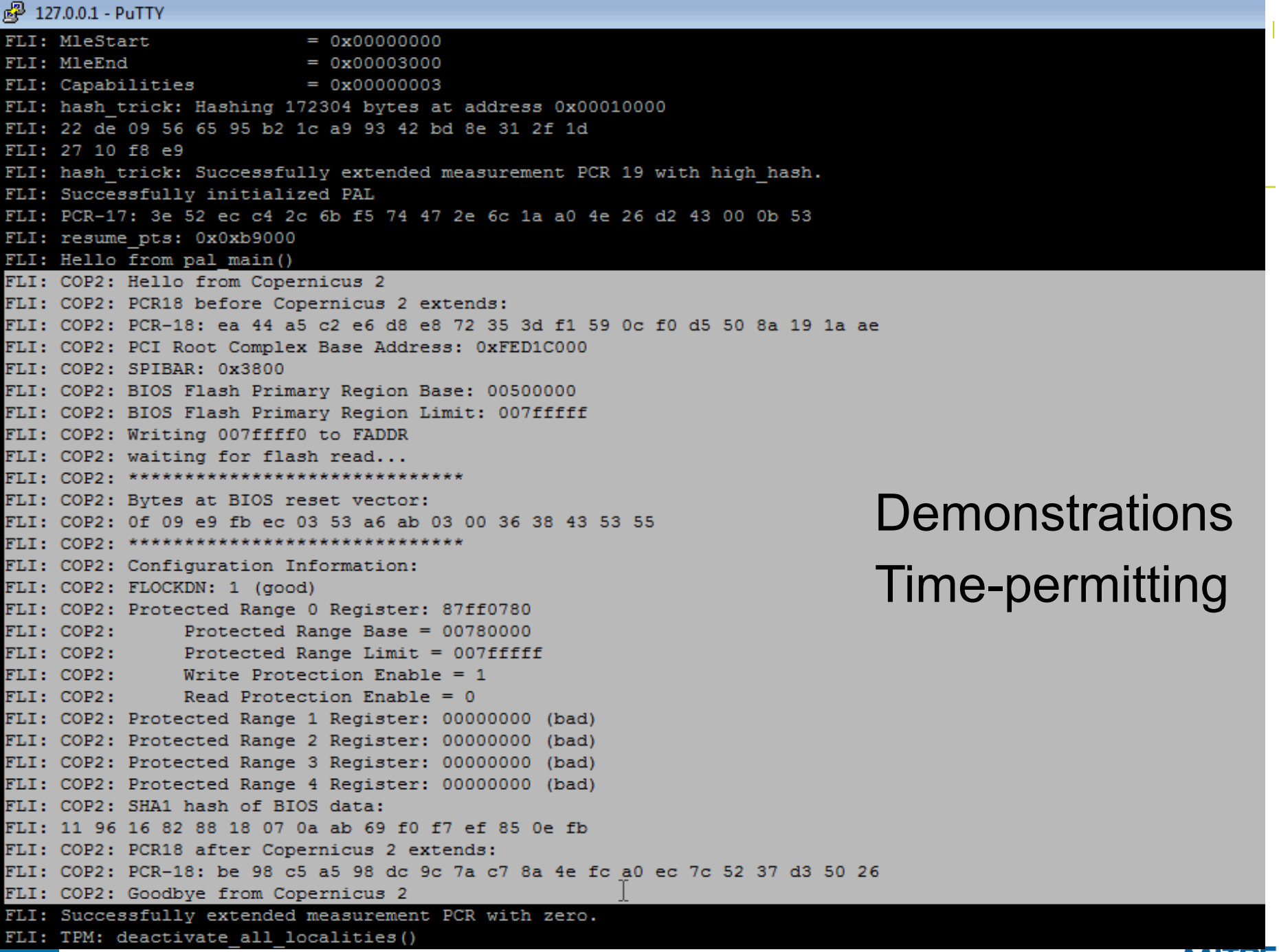
- **(Note: verification should generally be done on some other presumed-trusted platform, such as a server that has no purpose other than verification. "If you try to evaluate the trust on the potentially compromised system you're going to have a bad time")**

If everything matches

- **Congrats, you have a genuine measurement…**
- **Now you just need to figure out whether it actually contains malice or not ;)**
- **John has started offering a BIOS analysis class to help people understand what a reported difference in a BIOS actually means**
  - Too bad you already missed it at CSW
  - In the meantime you've probably got a lot of studying to do before you could take it (e.g. paging, port IO, static RE, IDA, etc)
  - Better head over to http://OpenSecurityTraining.info

Conclusions

- **There exists the potential for an attacker who controls SMM to perform a Man in the Middle attack on SPI reads and writes**
- **We have implemented such an attack**
  - Smite'em the Stealthy
- **The Great Hero Copernicus subsequently went into the belly of the beast and slew it with the power of TXT.**
- **Anyone giving you a BIOS measurement and *not* using TXT is untrustworthy**
  - "Copernicus 2 tech" – ask for it by name :)
  - We're licensing it to companies doing firmware integrity checks
  - We're releasing a binary-only version if you want to try it out:
  - http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/playing-hide-and-seek-with-bios-implants
- **If you don't have TXT support, or if your vendor messed up your TXT support, you're out of luck and you stay vulnerable!**
  - Smite'em's children live on in low end (non-TXT) machines



Demonstrations
Time-permitting

FAQ, Questions?

- **How do I get access to Cop 1 or 2 src code**
  - Cop 1 is available free, but we will want to receive the BIOS data you collect with it as an aid to our future research
  - Cop 2 is available for licensing
  - Contact Xeno – copernicus@mitre.org

- **If I don't have TXT or can't turn it on everywhere is there still any value in Copernicus 1?**
  - Yes, attackers probably weren't expecting Cop1, and thus it may catch ones who don't implement Smite'em functionality

- **What about Vendor X? Can I trust their measurements/**
  - Probably not. We're talking with some vendors about incorporation, but none of them have done it yet.

References

- [1] Attacking Intel BIOS – Alexander Tereshkin & Rafal Wojtczuk – Jul. 2009 http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf
- [2] TPM PC Client Specification - Feb. 2013 http://www.trustedcomputinggroup.org/developers/pc_client/specifications/
- [3] Evil Maid Just Got Angrier: Why Full-Disk Encryption With TPM is Insecure on Many Systems – Yuriy Bulygin – Mar. 2013 http://cansecwest.com/slides/2013/Evil%20Maid%20Just%20Got%20Angrier.pdf
- [4] A Tale of One Software Bypass of Windows 8 Secure Boot – Yuriy Bulygin – Jul. 2013 http://blackhat.com/us-13/briefings.html#Bulygin
- [5] Attacking Intel Trusted Execution Technology - Rafal Wojtczuk and Joanna Rutkowska – Feb. 2009 http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf
- [6] Another Way to Circumvent Intel® Trusted Execution Technology - Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin – Dec. 2009 http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf
- [7] Exploring new lands on Intel CPUs (SINIT code execution hijacking) - Rafal Wojtczuk and Joanna Rutkowska – Dec. 2011 http://www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf
- [7] Meet 'Rakshasa,' The Malware Infection Designed To Be Undetectable And Incurable - http://www.forbes.com/sites/andygreenberg/2012/07/26/meet-rakshasa-the-malware-infection-designed-to-be-undetectable-and-incurable/

References 2

- [8] Implementing and Detecting an ACPI BIOS Rootkit – Heasman, Feb. 2006 http://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf
- [9] Implementing and Detecting a PCI Rookit – Heasman, Feb. 2007 http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf
- [10] Using CPU System Management Mode to Circumvent Operating System Security Functions - Duflot et al., Mar. 2006 http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lti/cansecwest2006-duflot-paper.pdf
- [11] Getting into the SMRAM:SMM Reloaded – Duflot et. Al, Mar. 2009 http://cansecwest.com/csw09/csw09-duflot.pdf
- [12] Attacking SMM Memory via Intel® CPU Cache Poisoning – Wojtczuk & Rutkowska, Mar. 2009 http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf
- [13] Defeating Signed BIOS Enforcement – Kallenberg et al., Sept. 2013 – URL not yet available, email us for slides
- [14] Mebromi: The first BIOS rootkit in the wild – Giuliani, Sept. 2011 http://www.webroot.com/blog/2011/09/13/mebromi-the-first-bios-rootkit-in-the-wild/

References 3

- [15] Persistent BIOS Infection – Sacco & Ortega, Mar. 2009 http://cansecwest.com/csw09/csw09-sacco-ortega.pdf
- [16] Deactivate the Rootkit – Ortega & Sacco, Jul. 2009 http://www.blackhat.com/presentations/bh-usa-09/ORTEGA/BHUSA09-Ortega-DeactivateRootkit-PAPER.pdf
- [17] Sticky Fingers & KBC Custom Shop – Gazet, Jun. 2011 http://esec-lab.sogeti.com/dotclear/public/publications/11-recon-stickyfingers_slides.pdf
- [18] BIOS Chronomancy: Fixing the Core Root of Trust for Measurement – Butterworth et al., May 2013 http://www.nosuchcon.org/talks/D2_01_Butterworth_BIOS_Chronomancy.pdf
- [19] New Results for Timing-based Attestation – Kovah et al., May 2012 http://www.ieee-security.org/TC/SP2012/papers/4681a239.pdf

  http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Bilby-up.pdf

References 4

- [20] Low Down and Dirty: Anti-forensic Rootkits - Darren Bilby, Oct.2006 http://www.blackhat.com/presentations/bh-jp-06/BH-JP-06-Bilby-up.pdf
- [21] Implementation and Implications of a Stealth Hard-Drive Backdoor – Zaddach et al., Dec. 2013 https://www.ibr.cs.tu-bs.de/users/kurmus/papers/acsac13.pdf
- [22] Hard Disk Hacking – Sprite, Jul. 2013 http://spritesmods.com/?art=hddhack
- [23] Embedded Devices Security and Firmware Reverse Engineering - Zaddach & Costin, Jul. 2013 https://media.blackhat.com/us-13/US-13-Zaddach-Workshop-on-Embedded-Devices-Security-and-Firmware-Reverse-Engineering-WP.pdf
- [24] Can You Still Trust Your Network Card – Duflot et al., Mar. 2010 http://www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf
- [25] Project Maux Mk.II, Arrigo Triulzi, Mar. 2008 http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf

Backup

- I likes the nitty gritty, but decided this is too much of a side-track from the main point of the talk
- Also I wrote this up in the ideal way it's implemented, but Flicker differs slightly so I need to re-work to show how flicker does it

TXT Configuration Registers

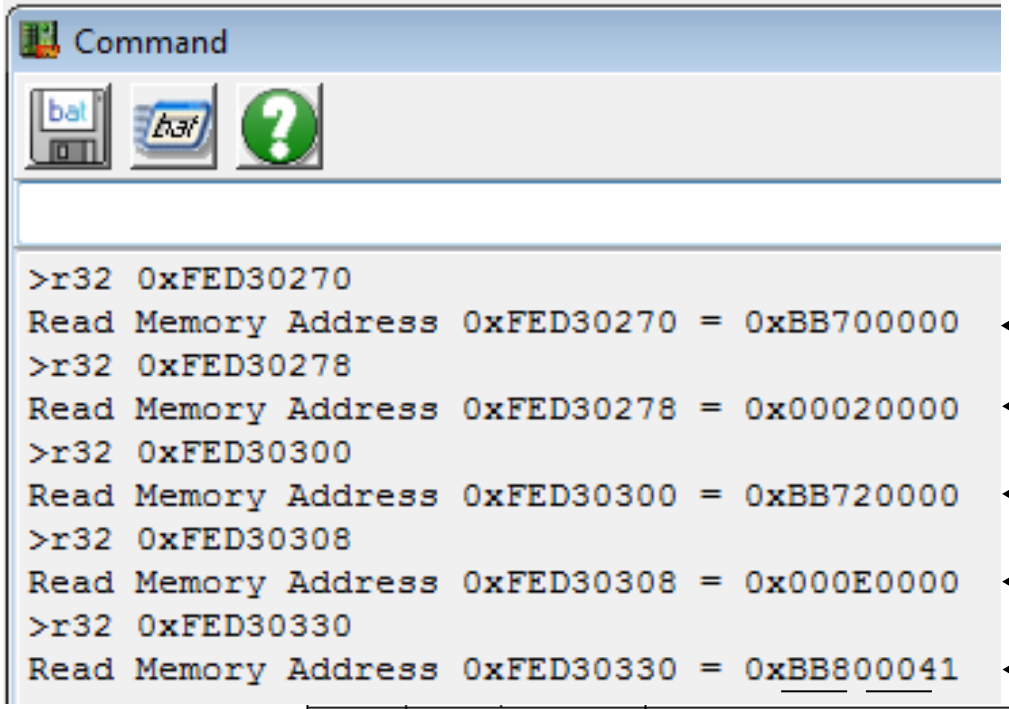
- At fixed physical address 0xFED3000 + offset
- Specified in Appendix B of the TXT developers guide

- TXT.SINIT.BASE = offset 0x270 = physical memory address the BIOS has set aside for the SINIT module to be copied into
- TXT.SINIT.SIZE = offset 0x278 = maximum available memory
- TXT.HEAP.BASE = offset 0x300 = physical memory reserved for use by the MLE, but also used when bootstrapping the MLE
- TXT.HEAP.SIZE = offset 0x308 = maximum available memory
- TXT.DPR = offset 0x330 = definition for the DMA Protected Region. This is actually a separate sort of DMA protection that doesn't have anything to do with Intel VT-d (IOMMU)

- Also has some error status and other important registers which we won't cover right now

Example reading TXT config regs with "Read Write Everything" – link

From a HP Elitebook 2540p



← TXT.SINIT.BASE

← TXT.SINIT.SIZE

← TXT.HEAP.BASE

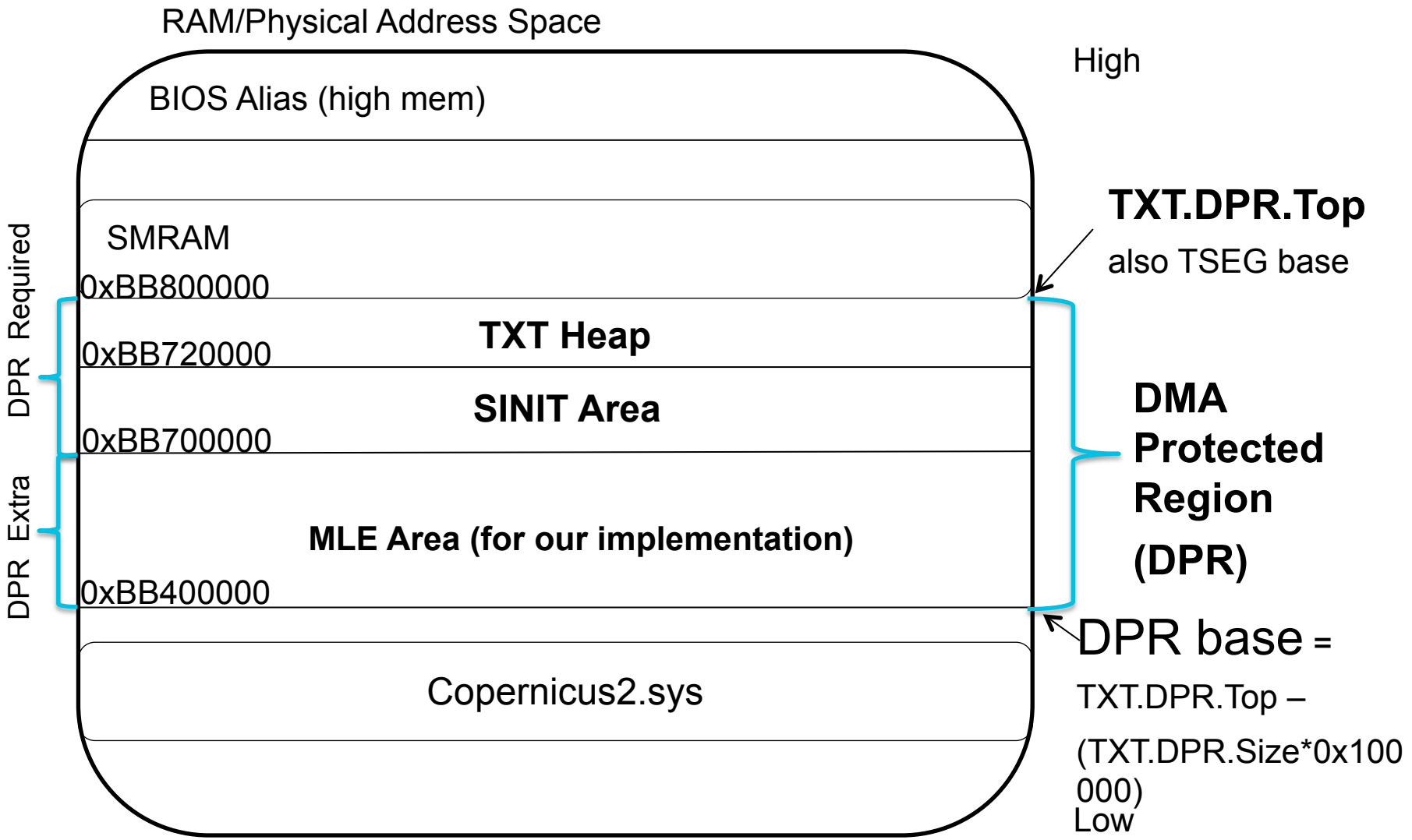
← TXT.HEAP.SIZE

← TXT.DPR

DPR Interpretation (from Intel TXT sw dev guide)

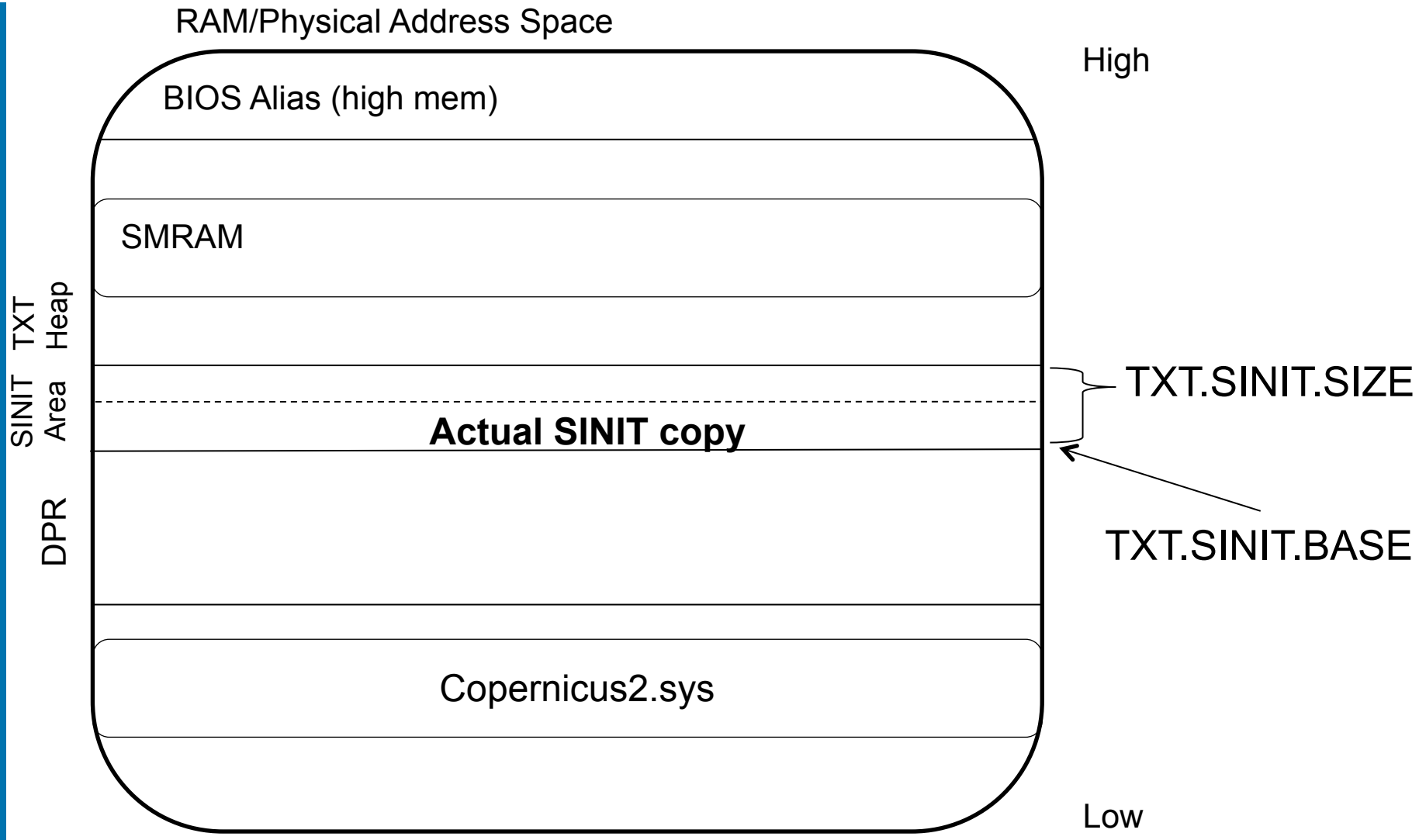
Bits	Field Name	Field Description
0	Lock	Bits 19:0 are locked down in this register when this bit is set.
3:1	Reserved	Reserved
11:4	Size	This is the size of memory, in MB, that will be protected from DMA accesses. A value of 0x00 in this field means no additional memory is protected. The DPR range works independently of any other DMA protections, such as VT-d, and is done post any VT-d translation or TXT checks.
19:12	Reserved	Reserved
31:20	Top	Top address + 1 of DPR. This is the base of TSEG.

Copernicus 2 Architecture: Enhance!



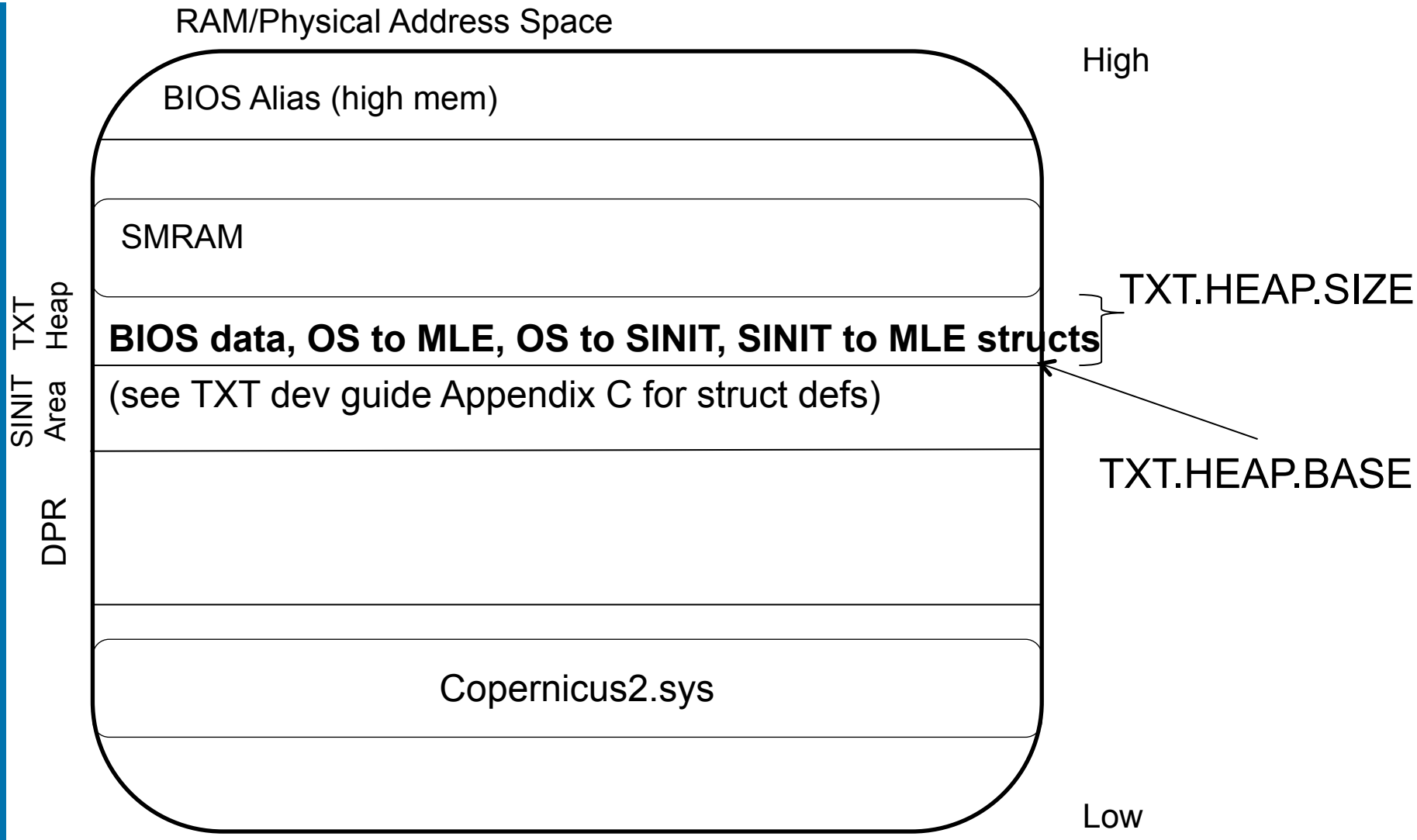
MITRE

Copernicus 2 Architecture: Enhance!



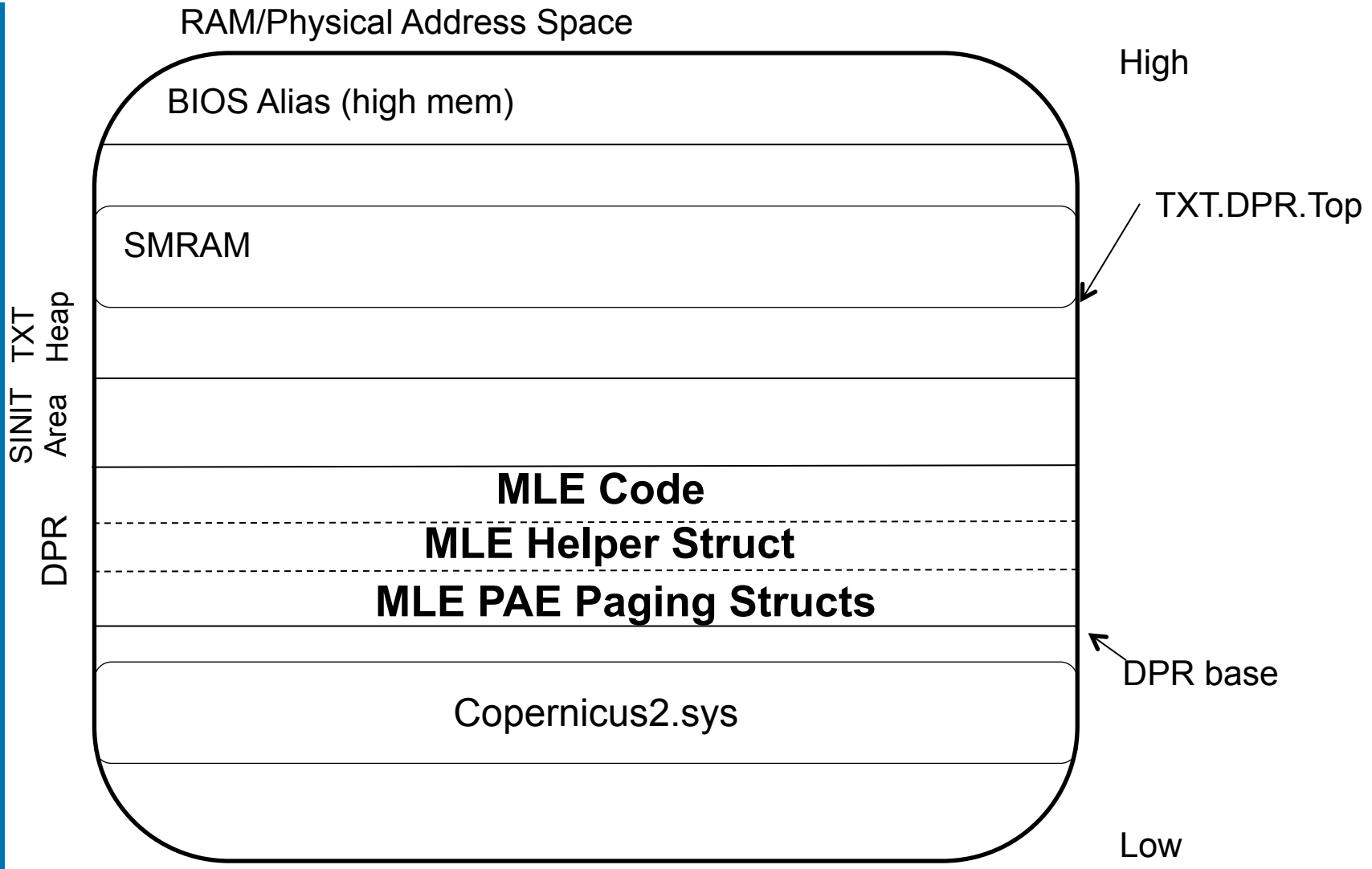
MITRE

Copernicus 2 Architecture: Enhance!



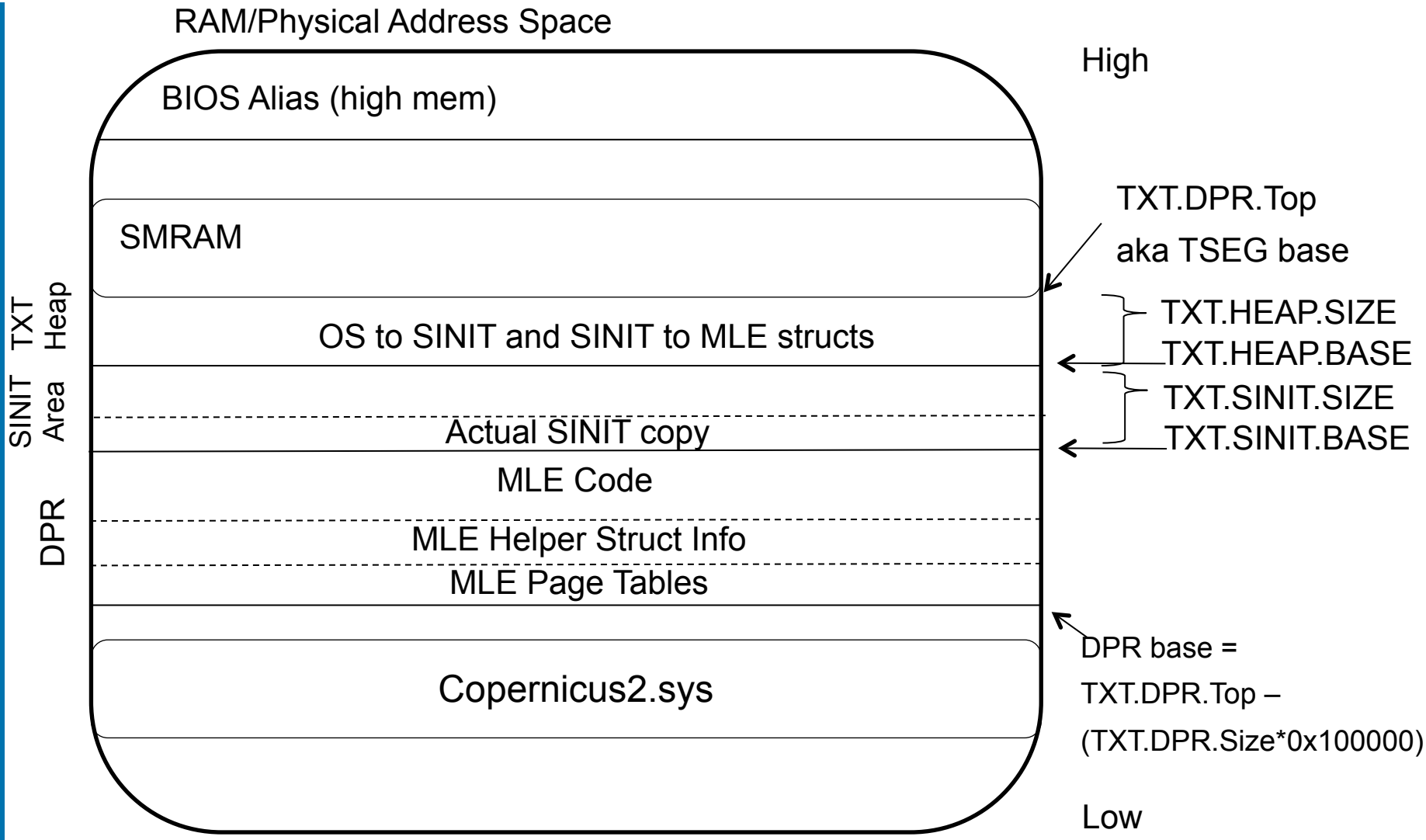
MITRE

Copernicus 2 Architecture: Enhance!



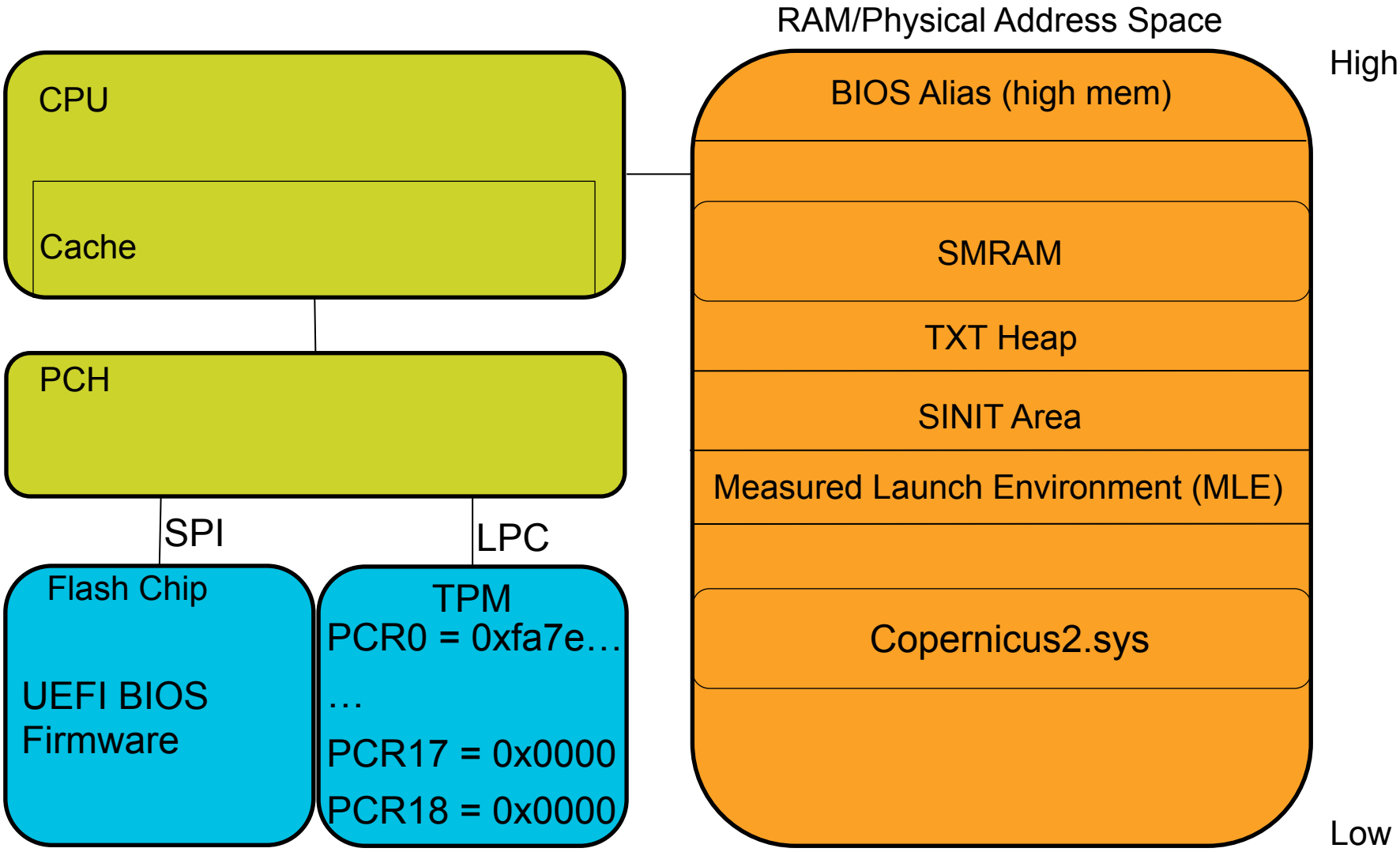
MITRE

Copernicus 2 Architecture: Enhance!



MITRE

Copernicus 2 Architecture



MITRE

Paging?

- **Yes, that's right, you have to roll your own PAE (Physical Address Extensions) paging structures in order to use TXT!**
  - Not canonical 32 bit paging that you learn in school, not 64 bit paging, but PAE 36 bit paging
- **Kind of a tall order for most folks**
  - Part of why TXT has limited uptake
- **But on the plus side you can go out and learn about x86 paging right now by taking Xeno's free "Intermediate x86 class"**
  - http://OpenSecurityTraining.info/Intermediatex86.html