BIOS Chronomancy: Fixing the Static Root of Trust for Measurement

John Butterworth

Corey Kallenberg

Xeno Kovah

MITRE

For internal MITRE use

Introduction

- **Who we are:**
  - Trusted Computing researchers at The MITRE Corporation
- **What MITRE is:**
  - A non-profit company that runs six US Government "Federally Funded Research & Development Centers" (FFRDCs)
  - We also do a lot of standards work such as CVE, CWE, etc

Motivation

- **Why:**
  - Attackers will seek to reach the highest privilege levels and/or the same privilege levels as the defender
  - We believe that access controls can always break down
  - We believe that BIOS Chronomancy is capable of detecting an attacker even when other access controls have broken down
    - Is it perfect? Sadly not yet, but we keep making it better
  - We believe this is a technology worthy of further exploration
  - We hope to inspire others to carry the torch and explore further

Outline

- **How the foundation of trusted measurement is rooted in firmware**
- **We will show that when this trust fails: really (really!) bad things can happen**
  - and we'll prove it
- **We will introduce BIOS Chronomancy, a technology capable of detecting an attacker who has achieved equal privileges**
- **We will show you the results of tests we performed using BIOS Chronomancy running in System Management Mode**

Terminology

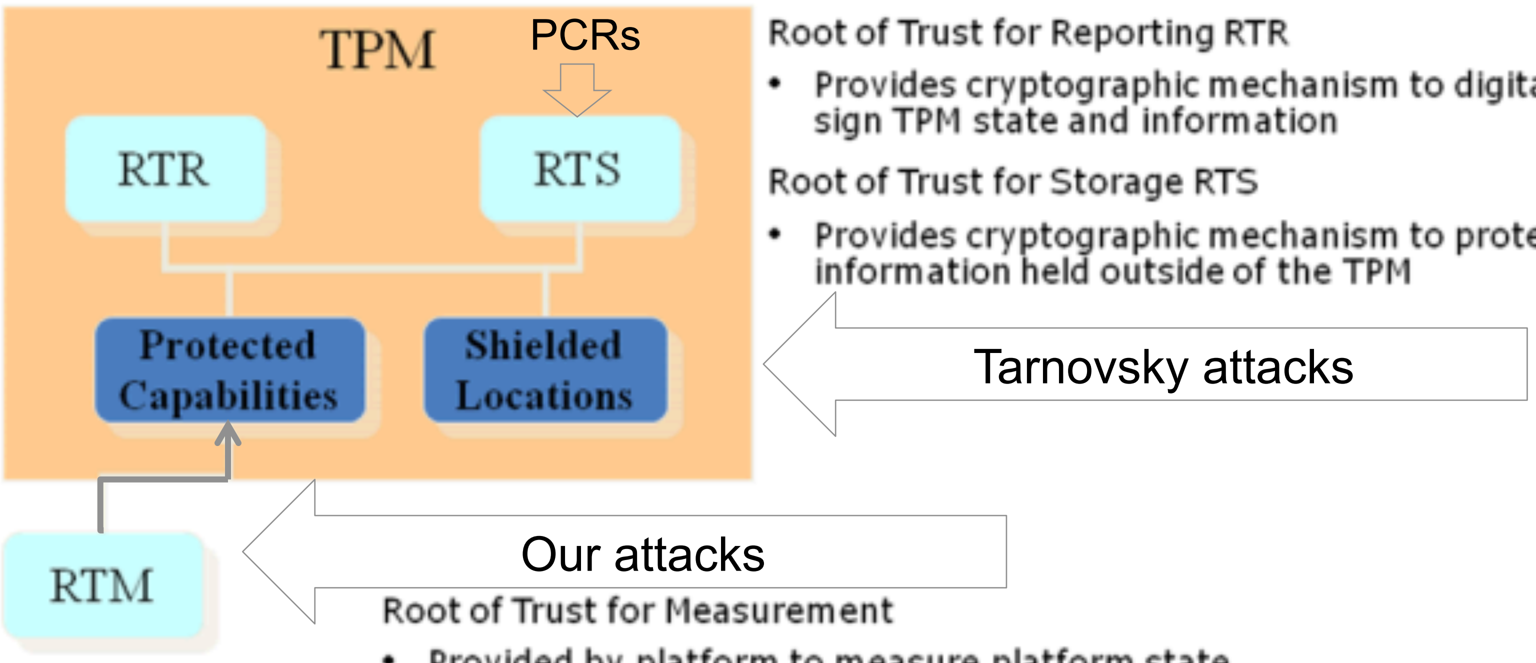
- **Trusted Platform Module (TPM)**
  - Supports secure key generation and secure key storage.
  - Can "seal" keys or data such that they can only be decrypted if the PCR set hasn't changed.
  - Can act as a root of trust for reporting by signing a quote of its current PCR set.
- **Platform Configuration Register (PCR)**
  - Store 20 byte hashes representing measurements of the system.
  - Are reset to $0x00_{20}$ upon reboot.
  - Can only be modified with an "Extend" operation.
  - Extend_PCR0(data): $PCR0_{new} = SHA1(PCR0_{old} \| SHA1(data))$

Terminology Continued

- **Trusted Boot**
  - A TPM supported boot of the system where each component in the boot up process (BIOS, Option ROMs, MBR) are *supposed to be* measured into PCRs before control is passed to them.
- **Static Root of Trust for Measurement (SRTM)**
  - The anchor in the Trusted Boot chain.
  - Responsible for measuring itself and other parts of the BIOS.
  - PCR0 holds the measurement of the SRTM.

All roots of trust are not created equal

Functional TPM Diagram



Root of Trust for Reporting RTR

- Provides cryptographic mechanism to digitally sign TPM state and information

Root of Trust for Storage RTS

- Provides cryptographic mechanism to protect information held outside of the TPM

Root of Trust for Measurement

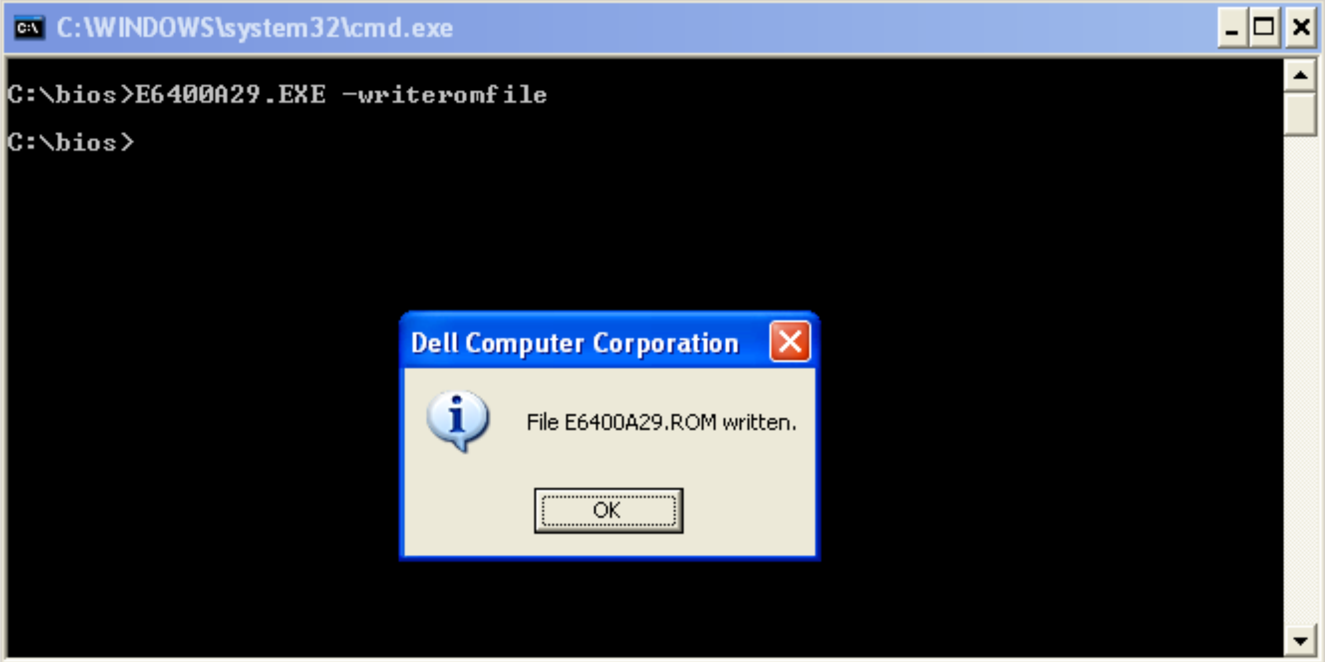
- Provided by platform to measure platform state
- Defined by platform specification

Interaction between RTR and RTS is important TPM capability

Base diagram from http://www.intel.com/content/dam/doc/white-paper/uefi-pi-tcg-firmware-white-paper.pdf

BIOS Acquisition

- **Method 1: Obtain the BIOS ROM from manufacturer**



```
C:\bios>E6400A29.EXE -writeromfile

C:\bios>
```

Dell Computer Corporation

File E6400A29.ROM written.

- **Dependent on manufacturer**
  - May not provide straight-forward method to obtain the actual ROM image
  - Dell, for example, no longer provides this handy feature.

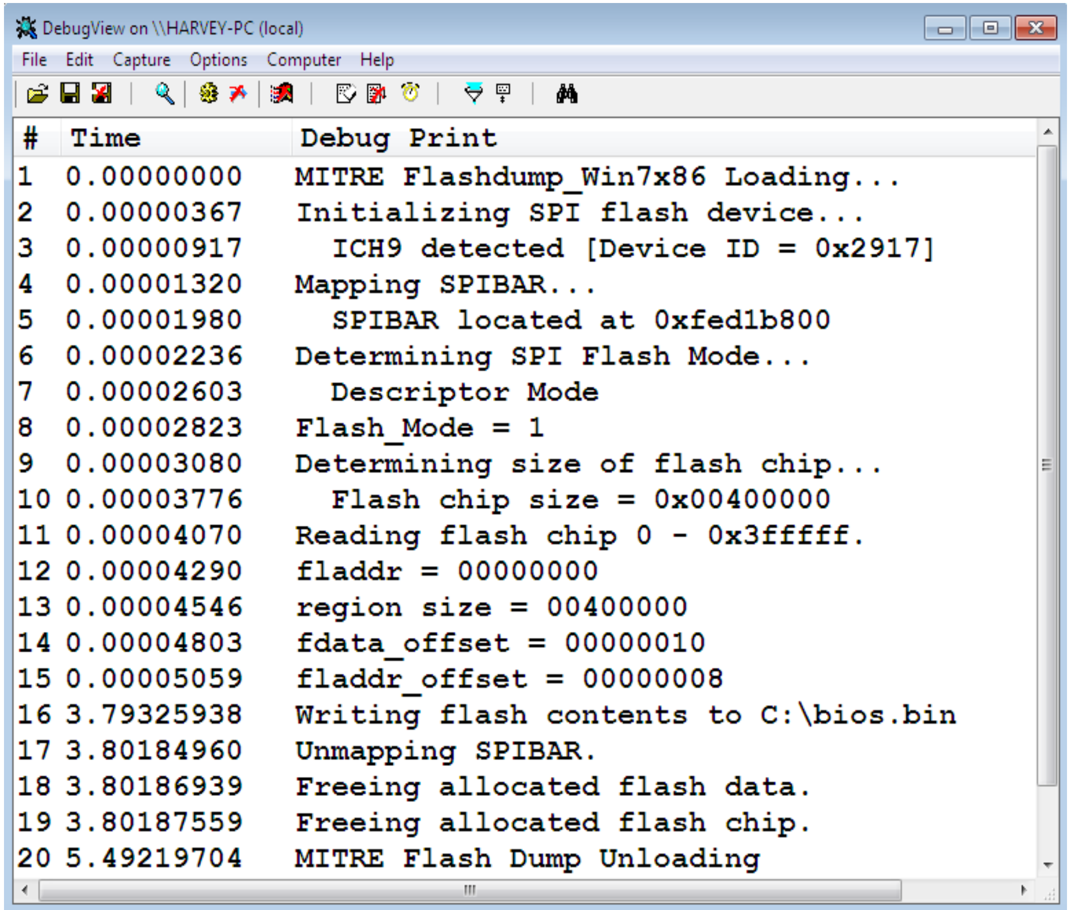
BIOS Acquisition

- Method 2: Read it from the BIOS chip using software

- Write your own if you want to learn the architecture very well

- Time consuming (but fun and educational)

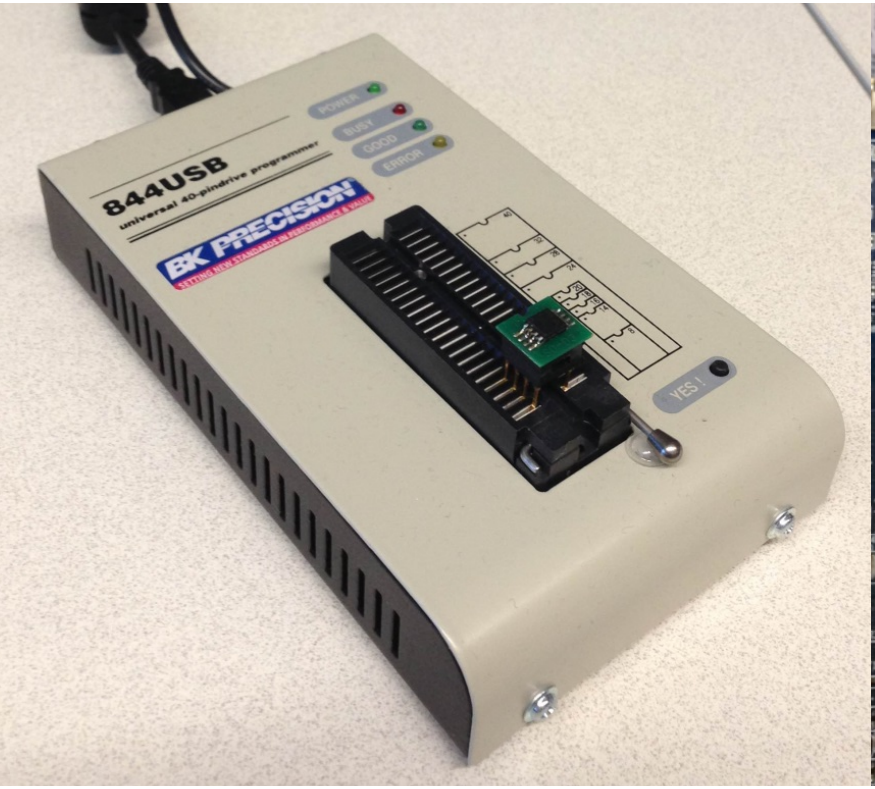
- Linux app with iopl() also works well, better for testing

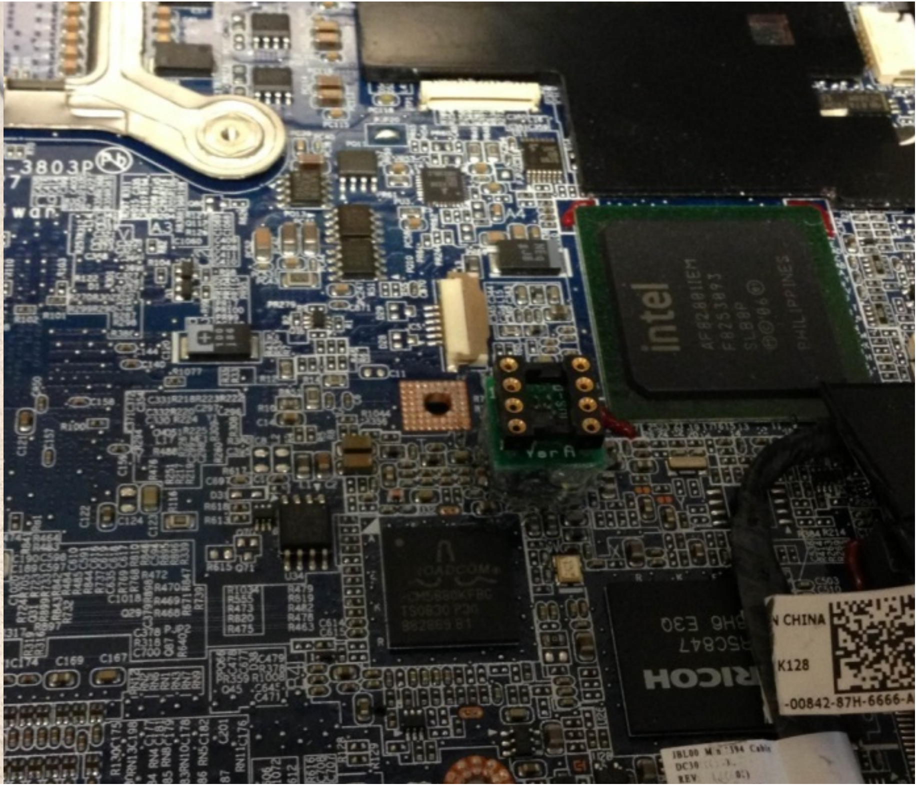

DebugView on \\HARVEY-PC (local)
File Edit Capture Options Computer Help

```
#  Time         Debug Print
1  0.00000000   MITRE Flashdump_Win7x86 Loading...
2  0.00000367   Initializing SPI flash device...
3  0.00000917     ICH9 detected [Device ID = 0x2917]
4  0.00001320   Mapping SPIBAR...
5  0.00001980     SPIBAR located at 0xfed1b800
6  0.00002236   Determining SPI Flash Mode...
7  0.00002603     Descriptor Mode
8  0.00002823   Flash_Mode = 1
9  0.00003080   Determining size of flash chip...
10 0.00003776     Flash chip size = 0x00400000
11 0.00004070   Reading flash chip 0 - 0x3fffff.
12 0.00004290   fladdr = 00000000
13 0.00004546   region size = 00400000
14 0.00004803   fdata_offset = 00000010
15 0.00005059   fladdr_offset = 00000008
16 3.79325938   Writing flash contents to C:\bios.bin
17 3.80184960   Unmapping SPIBAR.
18 3.80186939   Freeing allocated flash data.
19 3.80187559   Freeing allocated flash chip.
20 5.49219704   MITRE Flash Dump Unloading
```

BIOS Acquisition

- Method 3: Read it from the BIOS chip using hardware


POWER
BUSY
GOOD
ERROR
844USB
universal 40-pindrive programmer
BK PRECISION
SETTING NEW STANDARDS IN PERFORMANCE & VALUE
YES !

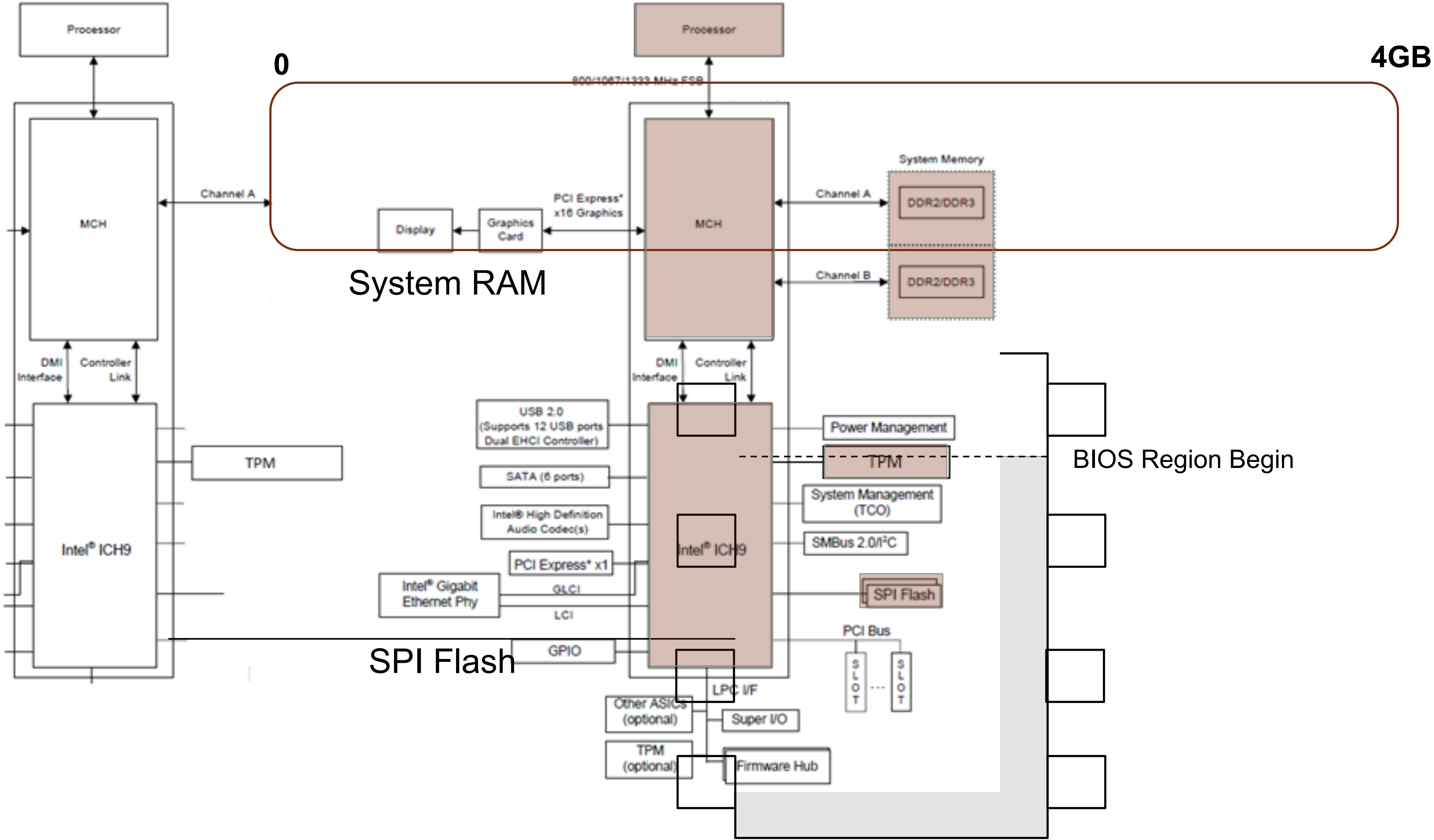


- Turned out to actually be a requirement …
- Not necessarily easy to get at the BIOS chip

BIOS Analysis: Arium CPU Debugger FTW!*

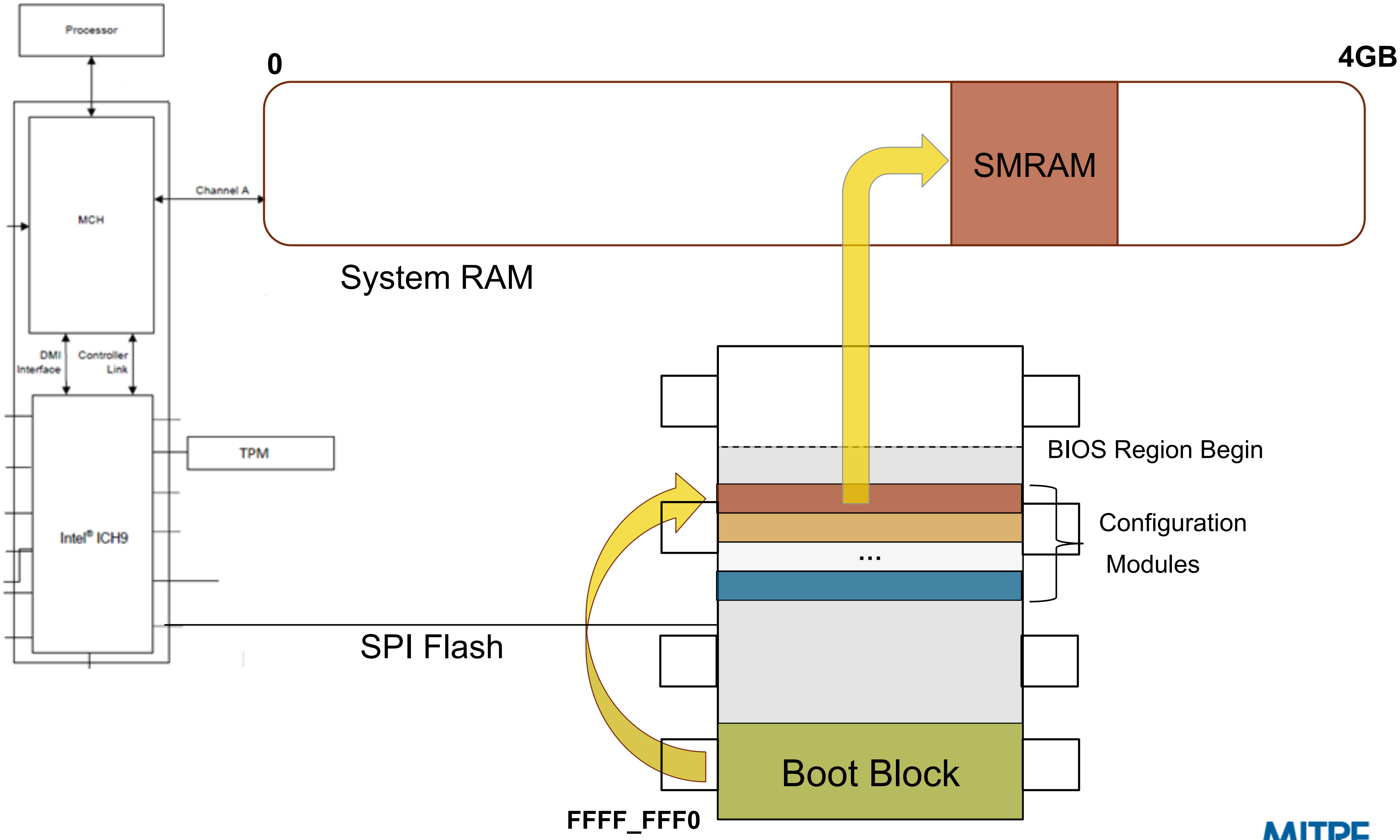
*Some [dis]assembly required.

Q35 Express Chipset

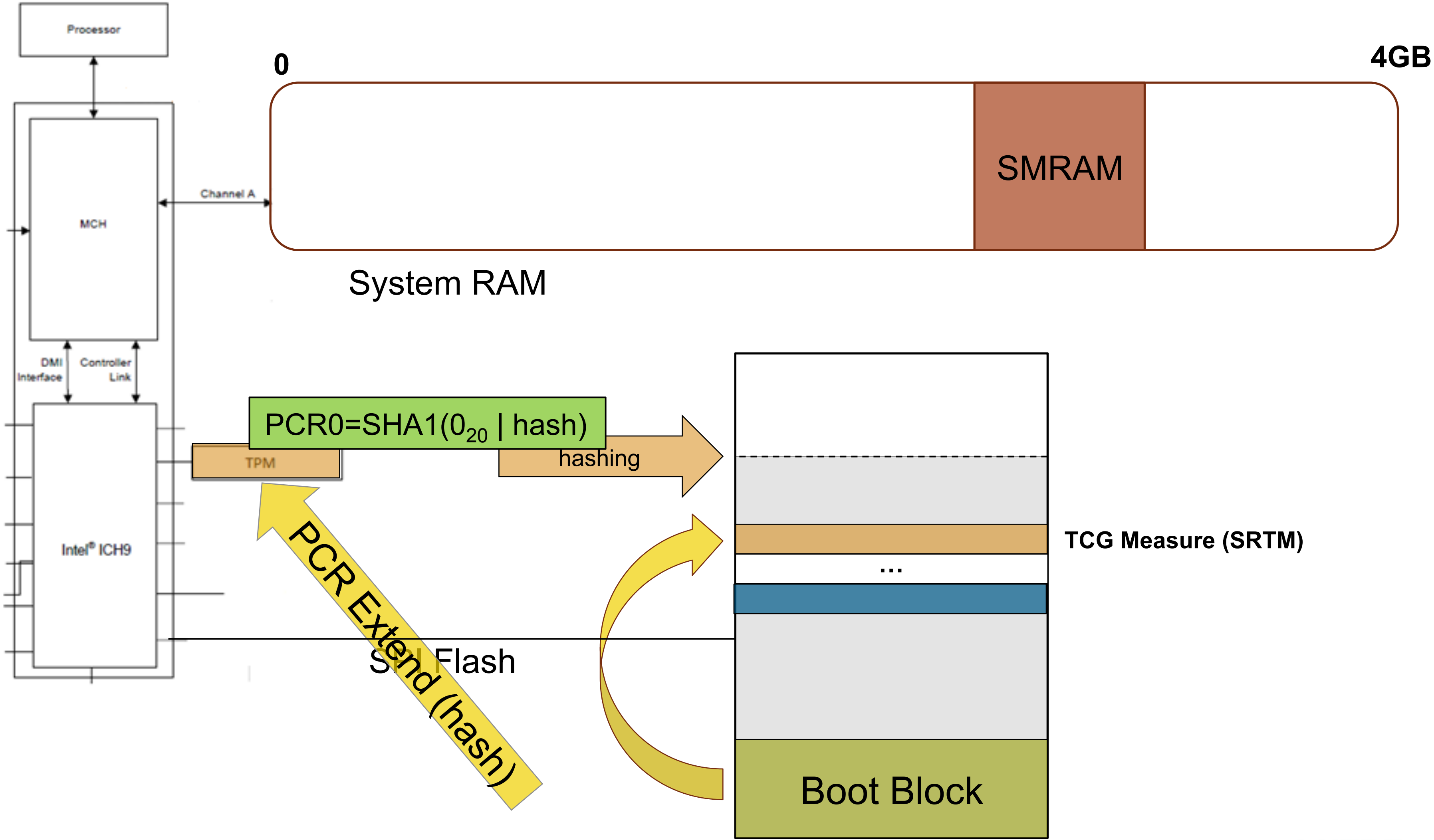


www.intel.com/.../**datasheet/io-controller-hub-9-datasheet**.pdf

Normal E6400 boot sequence 1



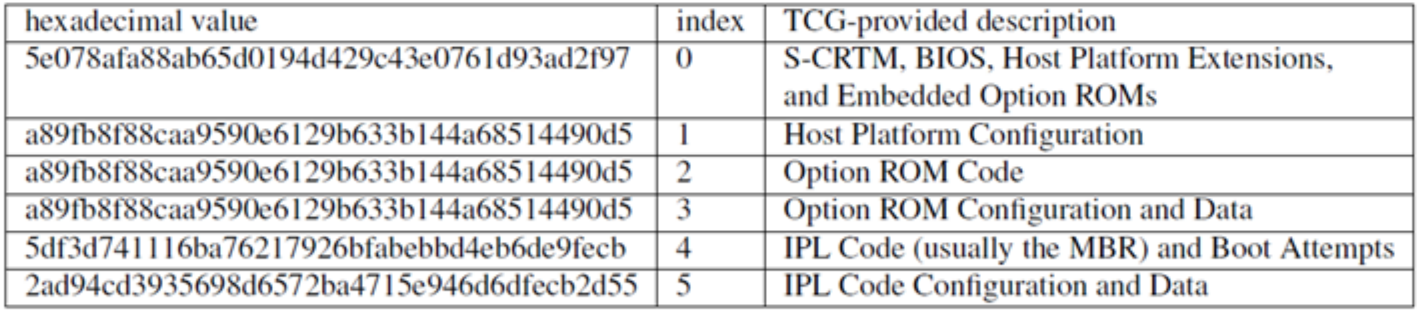
Normal E6400 boot sequence 2



The Problems with PCRs

- **Opaqueness**
  - No golden set of PCRs is provided by the OEM.
  - No description of what is actually being measured and incorporated into the PCR values.[1]
  - Homogeneous systems can have different PCR values.[2]
  - Duplicate PCR values are unexpected…

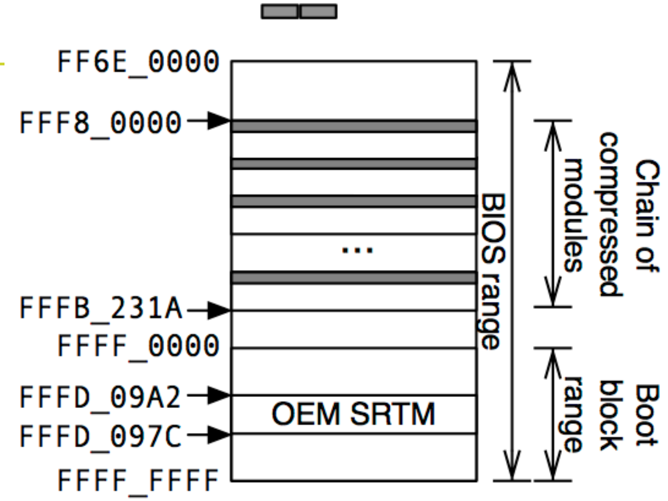
- **Example E6400 PCR Set**



hexadecimal value	index	TCG-provided description
5e078afa88ab65d0194d429c43e0761d93ad2f97	0	S-CRTM, BIOS, Host Platform Extensions, and Embedded Option ROMs
a89fb8f88caa9590e6129b633b144a68514490d5	1	Host Platform Configuration
a89fb8f88caa9590e6129b633b144a68514490d5	2	Option ROM Code
a89fb8f88caa9590e6129b633b144a68514490d5	3	Option ROM Configuration and Data
5df3d741116ba76217926bfabebbd4eb6de9fecb	4	IPL Code (usually the MBR) and Boot Attempts
2ad94cd3935698d6572ba4715e946d6dfecb2d55	5	IPL Code Configuration and Data

1. The TCG specification gives vague guidelines on what should be incorporated into individual PCR values, and many decisions are left to the vendor.
2. Based on our own observation of PCR values across various systems.

E6400 PCR0 (SRTM) Measurement



- **PCR0 should contain a measurement of the SRTM and other parts of the BIOS.**
- **In the above diagram, the dark areas represent what the E6400 actually incorporates into the PCR0 measurement.**
- **Only 0xA90 of the total 0x1A0000 bytes in the BIOS range are incorporated, including:**
  - The first 64 bytes of the 42 compressed modules.
  - Two 8 byte slices at 0xDF4513C0 and 0xDF4513C8.
  - The SRTM is not incorporated at all.

***BIOS Base is located at FFE6_0000**

Implications of the weak PCRs

- **We can modify the majority of the E6400 BIOS without changing any of the PCR values.**
- **Yuriy Bulygin made a similar discovery at CanSecWest 2013 regarding his ASUS P8P67.**
- **But what if we want to modify *any* part of the BIOS with no limits?**
  - Like the splash-screen, or the code that instantiates SMRAM?

Forging the PCRs

- **We can arbitrarily modify *any* part of the BIOS while still maintaining the expected PCR set if we do the following:**

  1. Record the expected hashes that the SRTM calculates and forwards to the TPM for the PCR_Extend operation(s).
  2. Modify the BIOS to prevent the legitimate SRTM from being called.
  3. Insert *your own SRTM* which simply replays the aforementioned “expected” hashes to the TPM.

- **This method maintains a valid PCR set even if the SRTM incorporates the *entire* BIOS into the measurement.**

BIOS Modification: Access Controls

- **Access Controls**
  - Registers which can prevent writes to the BIOS flash*
  - Signed Firmware Updates (per NIST 800-155)
- **Latitude E6400 BIOS revisions:**
  - A29 did not protect the flash from direct writes to the firmware flash from privileged applications
    - A30 and higher do ☺
  - A29 did not provide an option to require Signed Updates (released prior to NIST 800-155)
    - A30 and higher do, as well as Dell's newer systems ☺
- **However, even Access Controls can fail or be bypassed:**
  - In 2009 ITL showed that firmware signing can be bypassed in their Attacking Intel BIOS presentation.
  - And so have we. We are currently working with Dell to resolve the vulnerability.

***A detailed discussion about these architectural controls is beyond the scope of this presentation.**

Firmware Rootkit Types

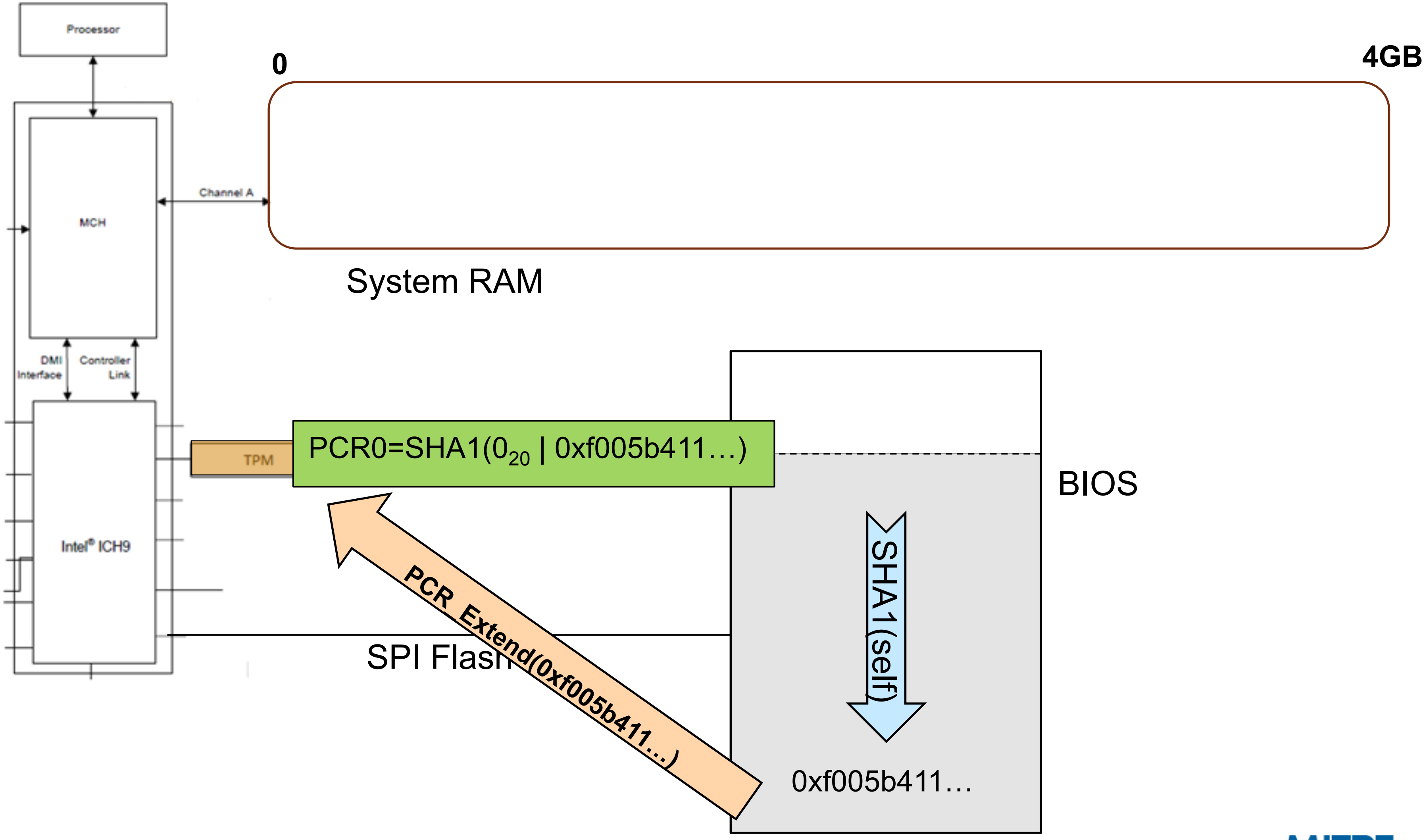
- **All firmware malware is resident on the NVRAM firmware and is therefore persistent**

- **Naïve**
  - That which can be detected by simply observing PCRs

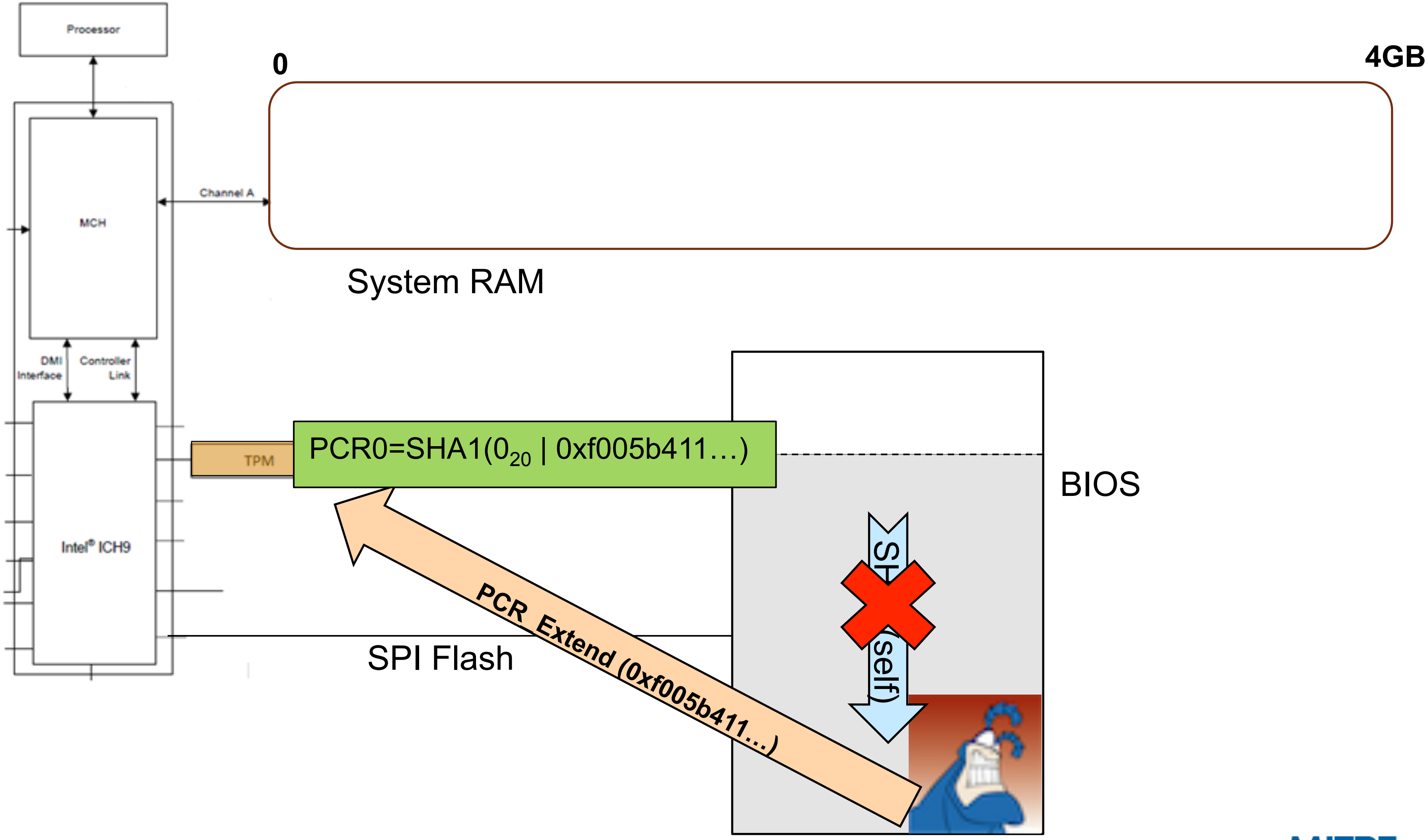
- **Tick**
  - Embeds itself in the firmware
  - Evades detection by forging PCRs
  - Once in place, can modify any other portion of the BIOS (even injecting itself into SMM)

- **Flea (to be discussed shortly)**

Normal BIOS PCR0 Measurement



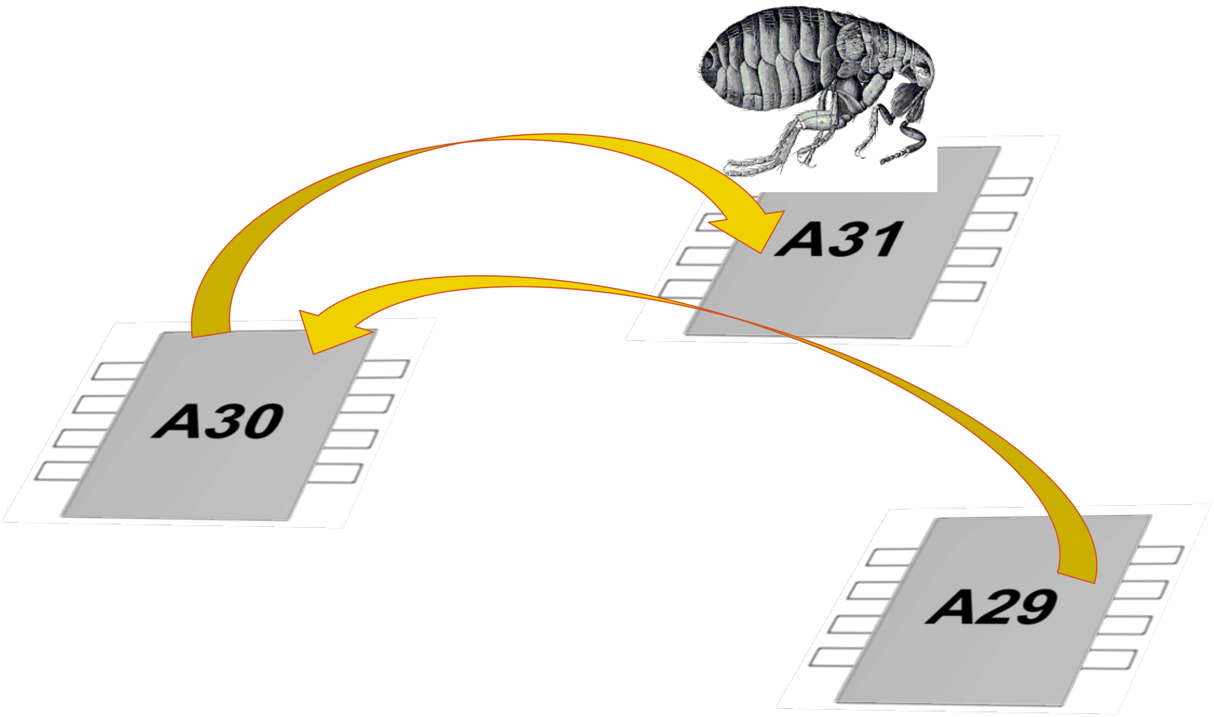
PCR0 Measurement with a Tick



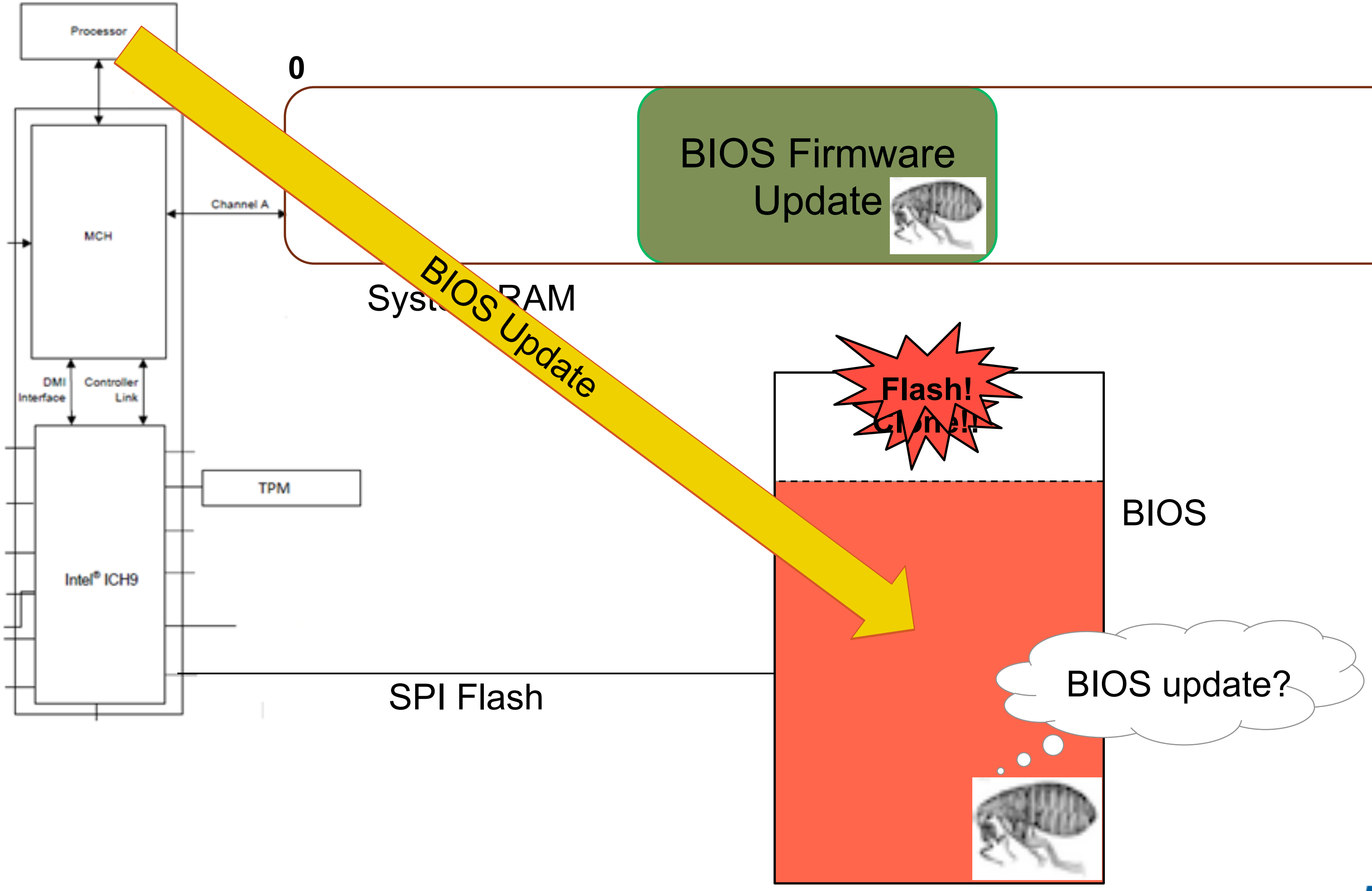
Tick Demo Video

The Flea

- **All the same stealth capabilities of the Tick**
- **Achieves persistence beyond BIOS re-flashes**
  - “Jumps” from one BIOS revision to another



The Flea


Processor
MCH
Channel A
DMI Interface
Controller Link
Intel® ICH9
TPM
0
4GB
BIOS Firmware Update
System RAM
BIOS Update
Flash! Clone!!
BIOS
BIOS update?
SPI Flash

Flea Demo Video

Countermeasure: Timing-based attestation

- **The fundamental premise:**
  - "Build your software so that if it's code is modified, it runs slower."
- **We coined "timing-based" because it is a superset of the "software-based" techniques, but using hardware (e.g. TPM) for timing measurement**
- **Meant to replace CRTM, but not reimplement entire SRTM**
- **Assumptions:**
  - Attacker has complete control of execution environment before self-checking begins (i.e. same privilege as defender)
  - Self-checksuming code is time-optimal for a given microarchitecture
  - There are no free execution slots where an attacker can insert a "free" instruction and suffer no timing slowdown
- **There is a decade of work in this area, we can't do the many many nuances justices. A timeline of related work here:**
  - http://bit.ly/11xEmlV

Components of all self-checks

- **Nonce/PseudoRandom Number(PRN)**
  - Decrease likelihood of precomputation due to storage constraints, and prevent replay (here only with online SMM-based challenges, not the boot)
- **Read own data**
  - Incorporated into checksum so if it changes the checksum changes
- **Read own instruction and data pointers**
  - Indicates where in memory the code itself is executing
- **Do all the above in millions of loop iterations**
  - So that ideally an instruction or two worth of conditional checks per loop iteration leads to millions of extra instructions in the overall runtime

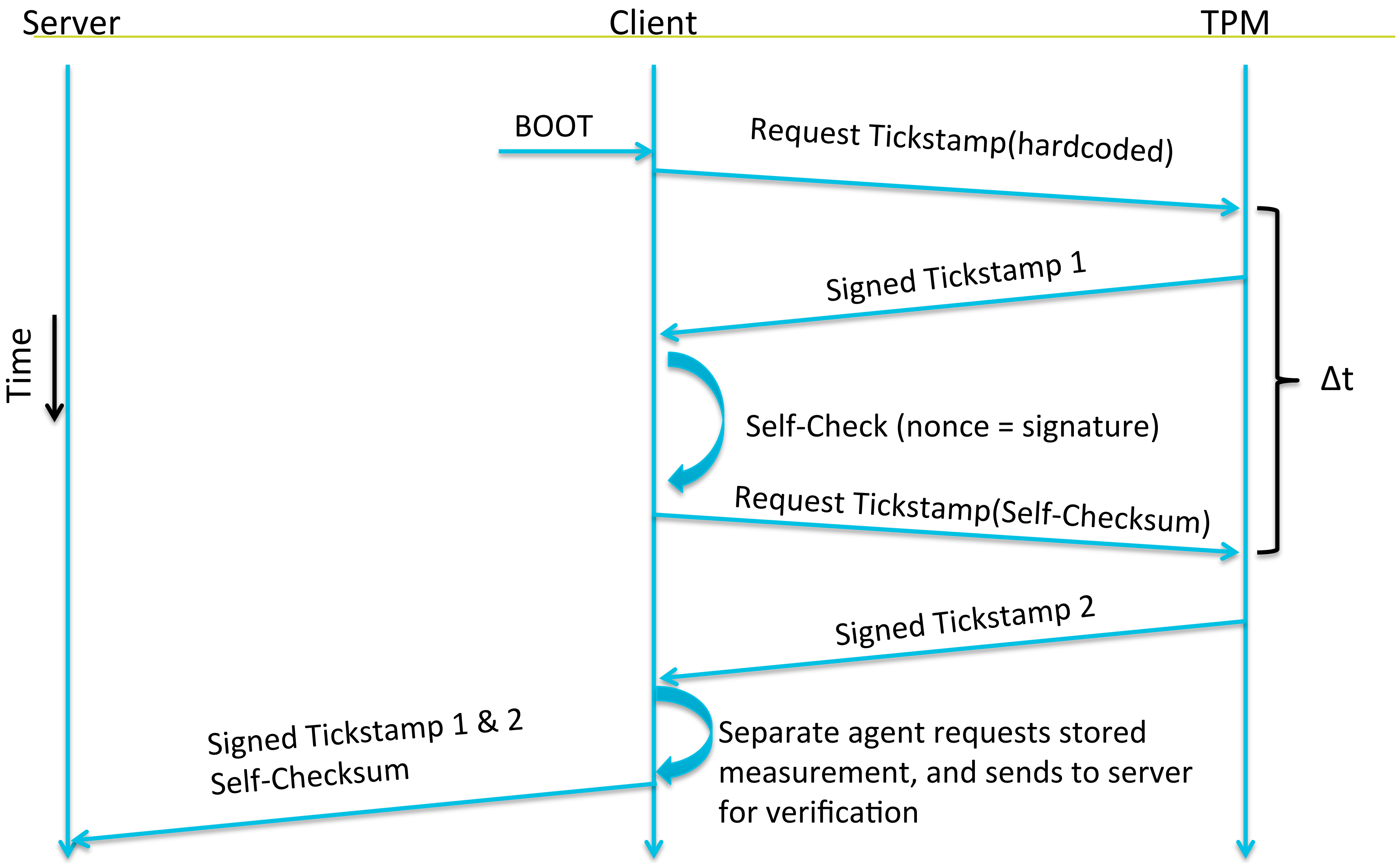
Simplified Selfcheck()

```
Selfcheck(checksum, nonce, codeStart, codeEnd, codeSize) {
       while (iteration < 2500000)
       {
              checksum[0] += nonce;
              checksum[1] ^= DP;
              checksum[2] += *DP;
              checksum[4] ^= EIP;
              mix(checksum);
              nonce += (nonce*nonce) | 5;
              DP = codeStart + (nonce % codeSize);
              iteration++;
       }
}
```

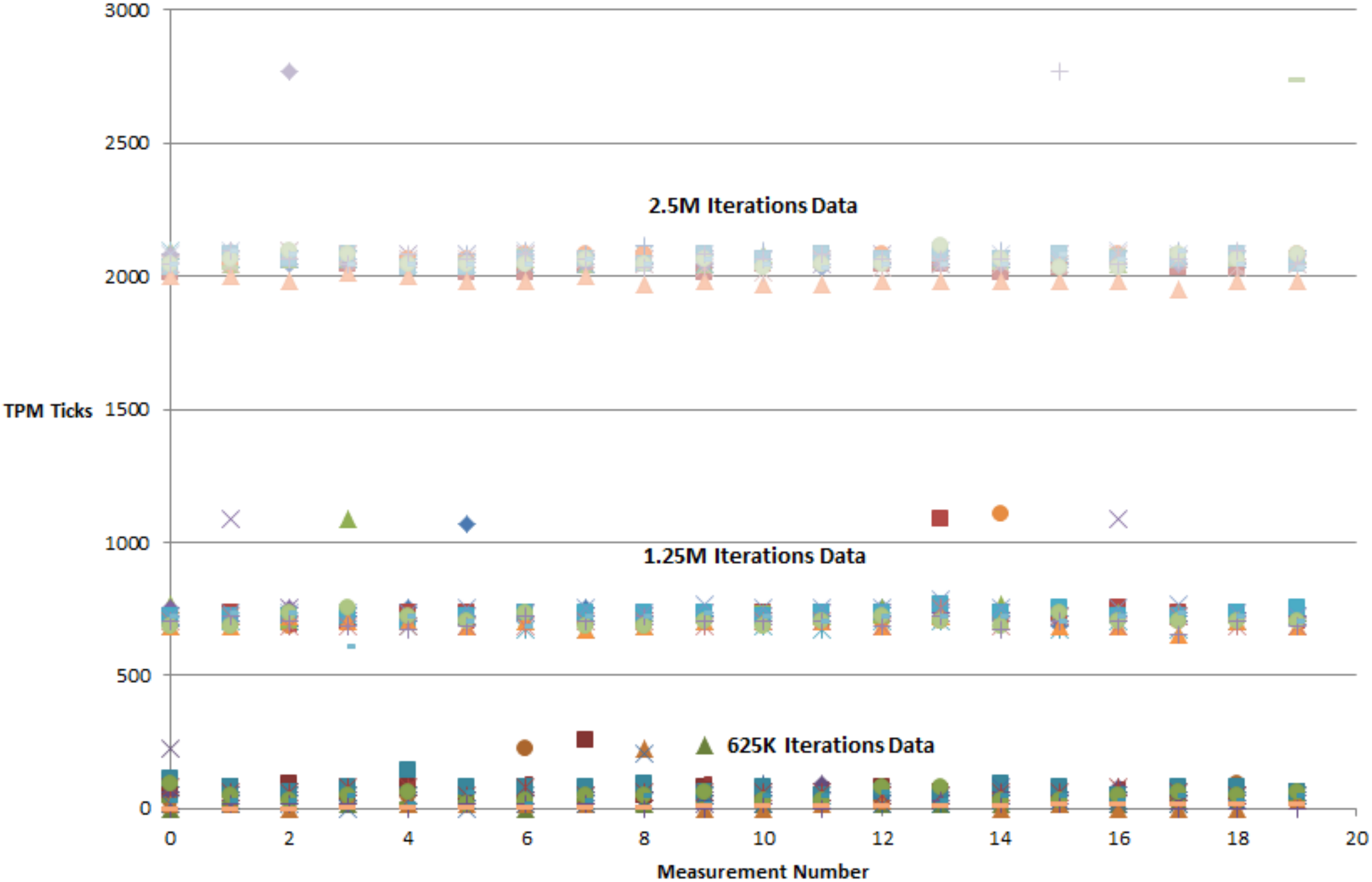
Simplified Selfcheck() Forgery

```
Selfcheck_forge(checksum, nonce, codeStart, codeEnd, codeSize) {
        while (iteration < 2500000)
        {
                checksum[0] += nonce;
                checksum[1] ^= DP;
                if (DP == myHookLocation)
                        checksum[2] += copyOfGoodBytes;
                else
                        checksum[2] += *DP;
                checksum[2] += *DP;
                checksum[4] ^= EIP;
                mix(checksum);
                nonce += (nonce*nonce) | 5;
                DP = codeStart + (nonce % codeSize);
                iteration++;
        }
}
```

Trusted Platform Module (TPM) Timing Implementation (BIOS Boot-Time)

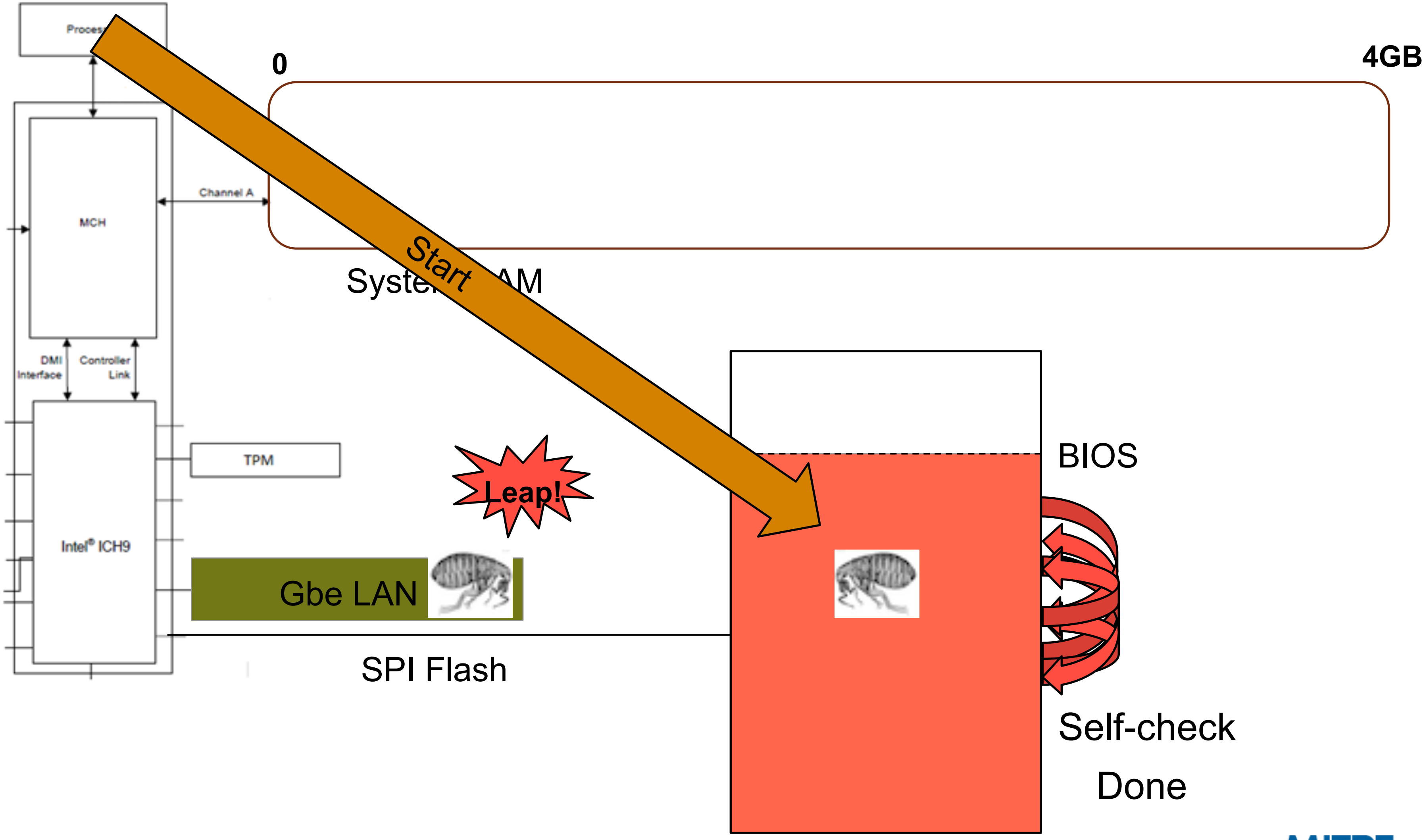

Server
Client
TPM
Time
BOOT
Request Tickstamp(hardcoded)
Signed Tickstamp 1
Self-Check (nonce = signature)
Request Tickstamp(Self-Checksum)
Δt
Signed Tickstamp 2
Signed Tickstamp 1 & 2
Self-Checksum
Separate agent requests stored measurement, and sends to server for verification

BIOS Chronomancy – attacker overhead vs. clean measurement 18 E6400s, 20 measurements, 3 different loop iteration counts



Takeaway: If you only do 625k iterations, occasionally the attacker wins. With 1.25M or more the attacker doesn't even get close.

Is BC perfect? NOPE!



Conclusion

- **There is a paper from CMU named VIPER specifically on attesting peripheral firmware. We will play with malicious peripherals & TOCTOU attacks this coming year.**
- **Trusted Computing implementations always need independent review. It's ironic that they're overwhelmingly closed source & proprietary. (Even academics don't usually post their code for review![1]) We don't want to get a 2.5% measurement and be lead to believe it's a 100% measurement.**
- **As long as the SRTM is implemented in writable firmware, ticks and fleas will mean that you can't trust your SRTM.**
  - And as ITL has shown, DRTM can depend on SRTM

[1] Our code for our previous self-check is at http://code.google.com/p/timing-attestation

We're working on getting the modified self-check for BC public released too.

Conclusion deux!

- We need more people working in this space!
- You should try your hand at making and breaking timing-schemes.
- It's obviously a very challenging and cool problem!

- Contact us for a private copy of the much more detailed whitepaper (still under submission for an academic conference)
- jbutterworth, ckallenberg, xkovah @ mitre.org

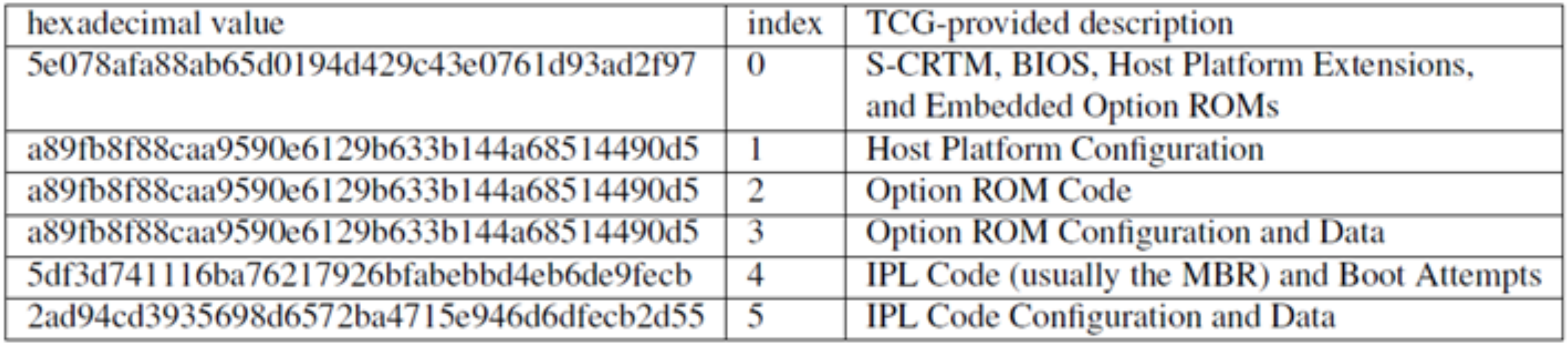
- P.S. To learn more about TPMs go to OpenSecurityTraining.info

References

- [1] Apokrif. Dell bios, how to decompose/mod, 2010. http://forums.mydigitallife.info/threads/12962-Dell-bios-how-to-decompose-mod./page48

Backup slides

E6400 PCR[1-3]



hexadecimal value	index	TCG-provided description
5e078afa88ab65d0194d429c43e0761d93ad2f97	0	S-CRTM, BIOS, Host Platform Extensions, and Embedded Option ROMs
a89fb8f88caa9590e6129b633b144a68514490d5	1	Host Platform Configuration
a89fb8f88caa9590e6129b633b144a68514490d5	2	Option ROM Code
a89fb8f88caa9590e6129b633b144a68514490d5	3	Option ROM Configuration and Data
5df3d741116ba76217926bfabebbd4eb6de9fecb	4	IPL Code (usually the MBR) and Boot Attempts
2ad94cd3935698d6572ba4715e946d6dfecb2d55	5	IPL Code Configuration and Data

- **PCRs 1-3 should contain configuration and option rom measurements.**
- **Interesting because they are duplicate values.**
- **We had also seen this a89fb8f… value on other (non-E6400) systems.**
- **PCR[1..3] = SHA1(0x00$_{20}$ || SHA1(0x00))**

Conditions for TOCTOU

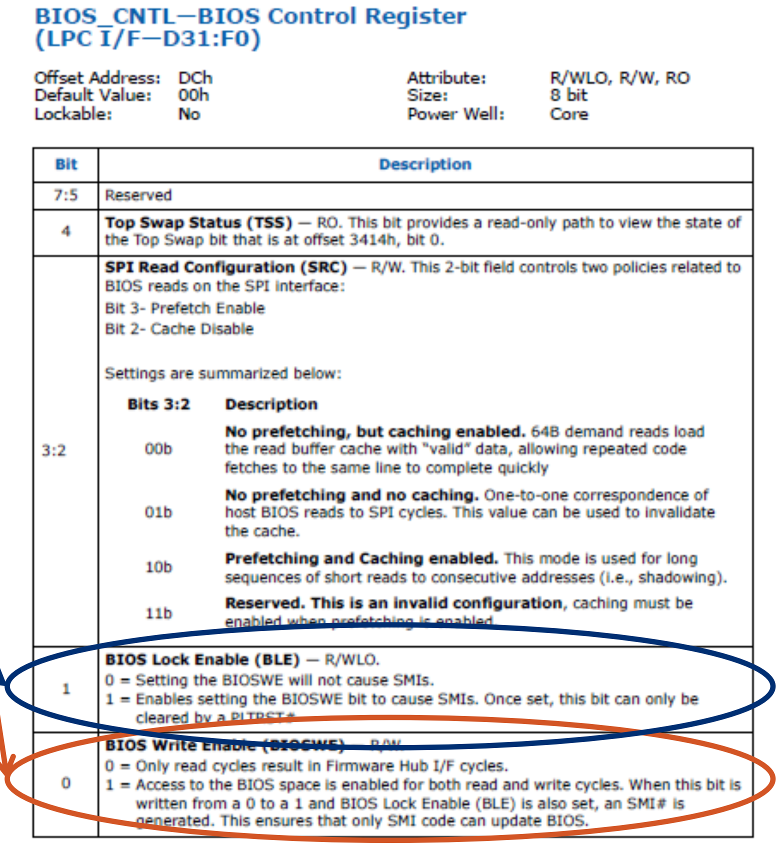
- 1) The attacker must know when the measurement is about to start.
- 2) The attacker must have some un-measured location to hide in for the duration of the measurement.
- 3) The attacker must be able to reinstall as soon as possible after the measurement has finished.

- It turns out a bunch of the example attacks in the literature are TOCTTOU without being explicit about it.
- And it turns out TOCTOU more severely undercuts the technique than prior work had acknowledged

BIOS Modification: Access Controls

BIOSWE can “always” be set to make the flash chip writeable (R/W attributes!)

BLE, however provides SMRAM the final say as to whether or not writes to the flash will be permitted.



BIOS_CNTL—BIOS Control Register (LPC I/F—D31:F0)

Offset Address:	DCh	Attribute:	R/WLO, R/W, RO
Default Value:	00h	Size:	8 bit
Lockable:	No	Power Well:	Core

Bit	Description
7:5	Reserved
4	**Top Swap Status (TSS)** — RO. This bit provides a read-only path to view the state of the Top Swap bit that is at offset 3414h, bit 0.
3:2	**SPI Read Configuration (SRC)** — R/W. This 2-bit field controls two policies related to BIOS reads on the SPI interface: Bit 3- Prefetch Enable Bit 2- Cache Disable Settings are summarized below: **Bits 3:2** **Description** 00b **No prefetching, but caching enabled.** 64B demand reads load the read buffer cache with "valid" data, allowing repeated code fetches to the same line to complete quickly 01b **No prefetching and no caching.** One-to-one correspondence of host BIOS reads to SPI cycles. This value can be used to invalidate the cache. 10b **Prefetching and Caching enabled.** This mode is used for long sequences of short reads to consecutive addresses (i.e., shadowing). 11b **Reserved. This is an invalid configuration**, caching must be enabled when prefetching is enabled.
1	**BIOS Lock Enable (BLE)** — R/WLO. 0 = Setting the BIOSWE will not cause SMIs. 1 = Enables setting the BIOSWE bit to cause SMIs. Once set, this bit can only be cleared by a PLTRST#
0	**BIOS Write Enable (BIOSWE)** — R/W. 0 = Only read cycles result in Firmware Hub I/F cycles. 1 = Access to the BIOS space is enabled for both read and write cycles. When this bit is written from a 0 to a 1 and BIOS Lock Enable (BLE) is also set, an SMI# is generated. This ensures that only SMI code can update BIOS.

E6400 version A29 didn't set BLE, A30 did

Coming Soon: Copernicus – “Question your assumptions”

- **We have a nice standalone tool**
- **It dumps BIOS to file**
- **It checks configuration registers to see if the BIOS/SMM is writable**
- **We’re interested in investigating the prevalence of unlocked flash chips**
- **Contact us**