



BIOS Chronomancy: Fixing the Core Root of Trust for Measurement

**John Butterworth
Corey Kallenberg
Xeno Kovah**

MITRE

Introduction

- **Who we are:**
 - Trusted Computing researchers at The MITRE Corporation
- **What MITRE is:**
 - A not-for-profit company that runs six US Government "Federally Funded Research & Development Centers" (FFRDCs) dedicated to working in the public interest
 - The first .org, !(.mil | .gov | .com | .edu | .net), on the ARPANET
 - Manager for a number of standards such as CVE, CWE, OVAL, CAPEC, STIX, TAXI, etc

Motivation

- **Why should you care about BIOS security?**
 - It's the first code that runs on your CPU
 - Almost no one is integrity checking the BIOS, so it's a great place for multi-year backdoors to reside
 - BIOS overwrite leads to very annoying/time-consuming to recover from bricking of machines (CIH Virus)
 - It's rarified knowledge, so it's cool :)
- **We cared because we wanted to know how trusted computing BIOS security mechanisms work.**
 - What is actually measured to generate Trusted Platform Module (TPM)-stored BIOS measurements?
 - Can an attacker defeat these measurements?
 - How can we build a better root of trust – one that detects an attacker at the same privilege level as the defender?
 - Heresy!? Read-on!

Outline

- **How an attacker could get into BIOS**
- **How the trusted computing technology of the Core Root of Trust for Measurement (CRTM) is rooted in the writable BIOS, and therefore vulnerable to manipulation**
- **BIOS malware (tick, flea) that can subvert TPM-mediated trust**
- **Our defensive strategy – BIOS Chronomancy**
- **Conclusions**

Dell Intro

Rick Martinez – BIOS Security Architect

Dell End User Computing Solutions

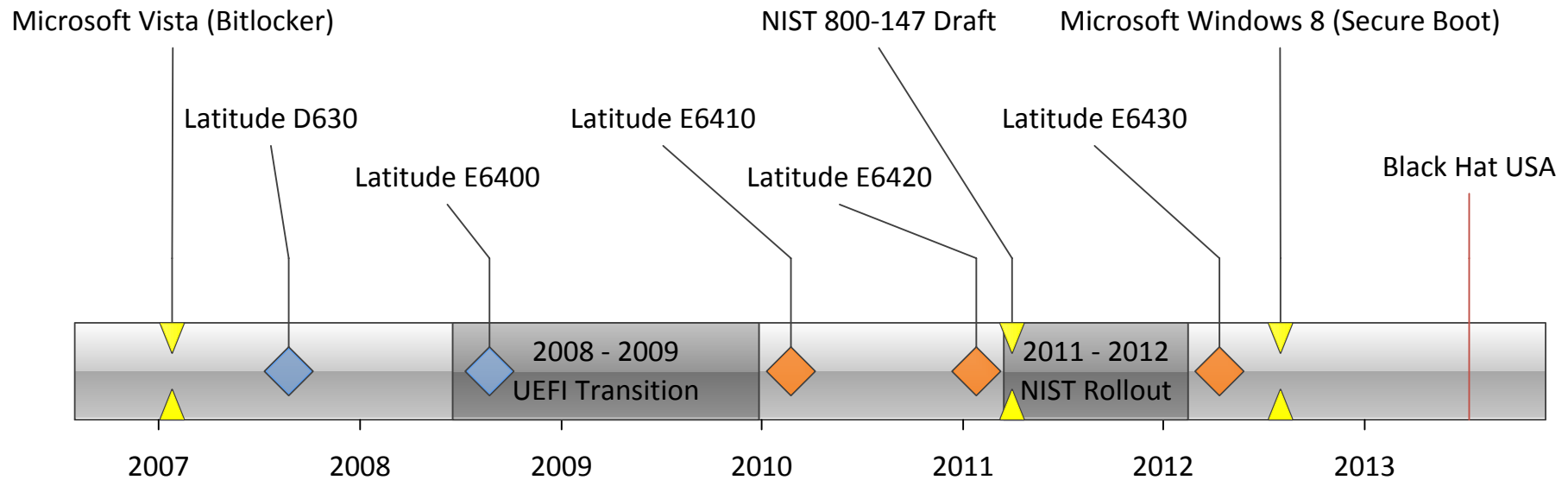
Rick_Martinez@Dell.com

Dell Latitude E6400

- * Platform launch August 2008
- * Legacy BIOS – x86 assembly language
- * TCG implementation for Vista Bitlocker
- * Last generation of legacy BIOS on Latitude

New BIOS available on support.dell.com (A34)

Dell Latitude Timeline



Getting into BIOS

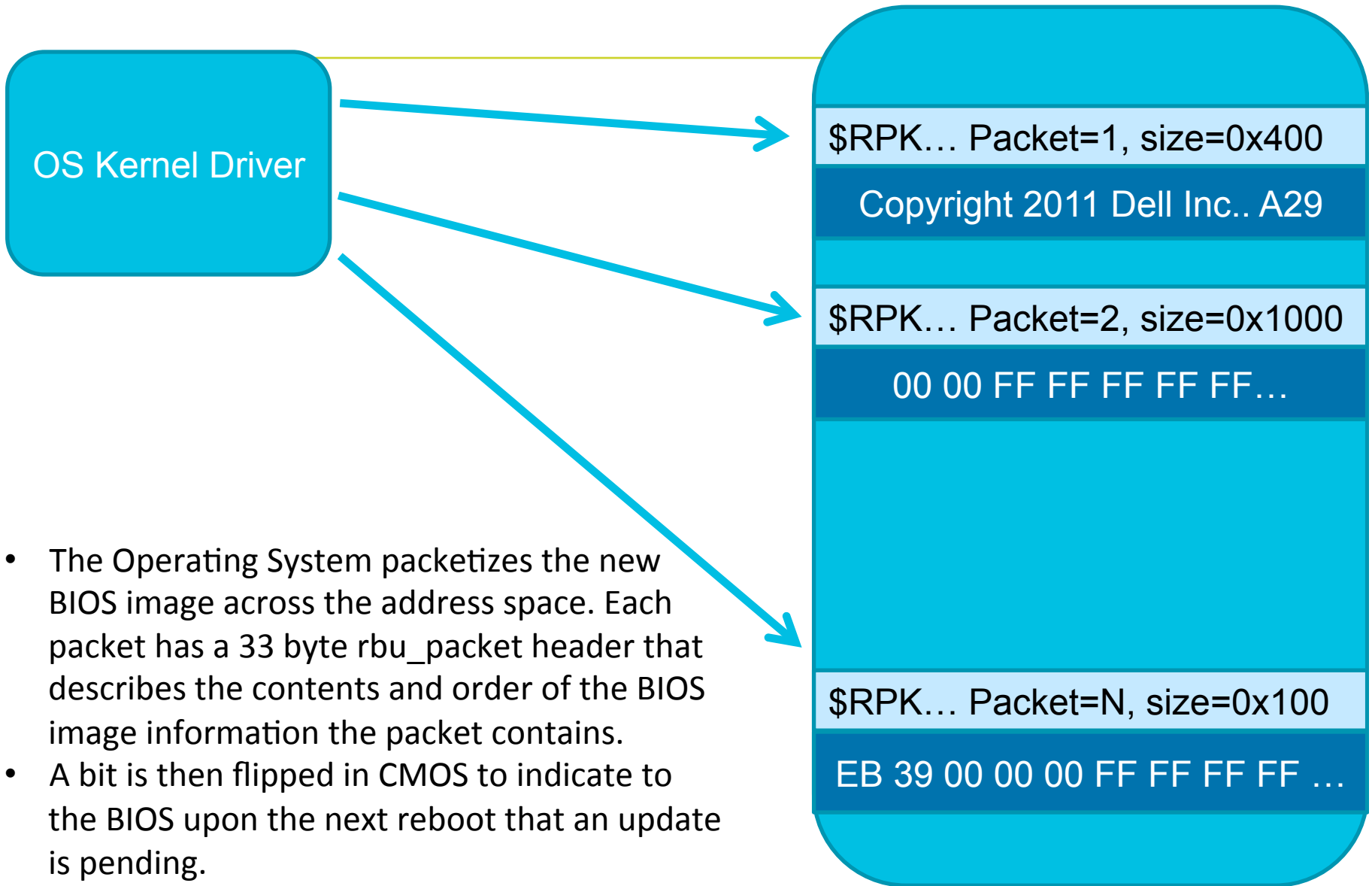
- **Access Controls**
 - There are registers that can prevent writes to the BIOS flash*
 - Signed firmware updates
- **Latitude E6400 BIOS revisions:**
 - A29 did not protect the flash from direct writes to the firmware flash from privileged applications
 - A30 and higher do
 - A29 did not provide an option to require signed updates
 - A30 and higher *have the option*, on newer Dell systems it's required
- **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[1]
 - *Until now*, the only BIOS talk to bypass signed update with an exploit

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

Dell E6400 BIOS Update

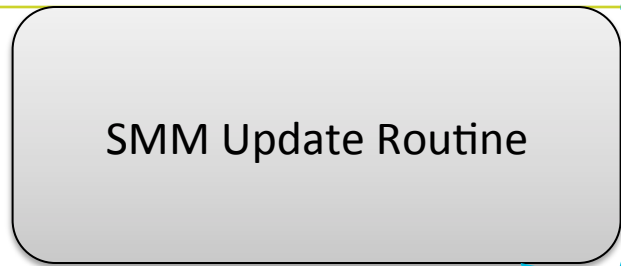
- 1. Firmware update binary (“HDR”) is copied to kernel memory**
 - Default method is to packetize the HDR file into “rbu packets”
 - HDR contains more than just the BIOS update (Keyboard Controller, Management Engine, too)
- 2. A bit in CMOS byte 0x78 is flipped**
- 3. The system is rebooted**
- 4. BIOS sees CMOS bit is flipped and triggers an SMI to execute the SMM BIOS Update routine**

BIOS Update Routine (1 of 2)



- The Operating System packetizes the new BIOS image across the address space. Each packet has a 33 byte rbu_packet header that describes the contents and order of the BIOS image information the packet contains.
- A bit is then flipped in CMOS to indicate to the BIOS upon the next reboot that an update is pending.

BIOS Update Routine (2 of 2)



- Upon reboot, the System Management Mode update routine scans for the individual rbu packets and uses them to reconstruct the complete BIOS image.
- SMM then verifies the reconstructed BIOS image is signed by Dell before writing to the flash chip.

Attacker Objective and Plan

- **Reflash BIOS chip with arbitrary image despite signed BIOS enforcement.**
- **Method: find a memory corruption vulnerability in the parsing of the BIOS update information (RBU packets). This will allow us to seize control of SMM and reflash the BIOS chip at will.**
- **The memory corruption vulnerability must occur before the signature on the bios update image is checked.**
- **SMM parses the 33 byte rbu_packet header that describes metadata about the BIOS update image. This parsing occurs before the signature check.**

Attack Surface

```
struct  rbu_packet
{
    u32 pktId;          // must be '$RPK'
    u16 pktSize;       // size of packet in KB
    u16 reserved1;     //
    u16 hdrSize;       // size of packet header in paragraphs (16 byte chunks)
    u16 reserved2;     //
    u32 pktSetId;      // unique id for packet set, can be anything
    u16 pktNum;        // sequential pkt number (only thing that changes)
    u16 totPkts;       // total number of packets
    u8  pktVer;        // version == 1 for now
    u8  reserved[9];
    u16 pktChksum;     // sum all bytes in pkt must be zero
    u8  pktData;      // Start of packet data.
}
LIBSMBIOS_PACKED_ATTR;
```

Packet Parsing

```
DFF23BCD mov     eax, [esp+arg_0_rbu_packet]
DFF23BD1 mov     eax, [eax]           ; base of RBU_Packet
DFF23BD3 mov     ds:g_foundKPR, 1
DFF23BDC mov     ecx, [eax+0Ch]
DFF23BDF mov     ds:g_pktSetId, ecx ; rbu_packet.pktSetId
DFF23BE5 movzx   edx, word ptr [eax+12h]
DFF23BE9 mov     ds:g_totPkts, edx ; rbu_packet.totPkts
DFF23BEF movzx   ecx, word ptr [eax+8]
DFF23BF3 shl     ecx, 4
DFF23BF6 mov     ds:g_hdrSize, ecx ; rbu_packet.hdrSize
DFF23BFC movzx   eax, word ptr [eax+4]
DFF23C00 shl     eax, 0Ah
DFF23C03 sub     eax, ecx
DFF23C05 dec     edx
DFF23C06 imul  edx, eax
DFF23C09 cmp     edx, 800000h
DFF23C0F mov     ds:g_pktSizeMinusHdrSize, eax ; 0x7fe0
```

- SMM first locates the RBU packet by scanning for an ASCII signature upon page aligned boundaries.
- Once located, members of the RBU packet are stored in an SMM data area for use in later calculations...

Curious GEOR?

```
DFF23C3C mov     eax, esi           ; esi=101000
DFF23C3E cmp     ecx, esi
DFF23C40 pop     esi
DFF23C41 pop     ebx
DFF23C42 jb     short loc_DFF23C54

DFF23C44 loc_DFF23C44:
DFF23C44 mov     dword ptr [eax], 'ROEG'
DFF23C4A add     eax, ds:g_pktSizeMinusHdrSize
DFF23C50 cmp     eax, ecx
DFF23C52 jbe     short loc_DFF23C44

DFF23C54 loc_DFF23C54:
DFF23C54 push    1
DFF23C56 pop     ecx
```

- When reconstructing the BIOS image from the rbu packets, SMM writes an initialization string “GEOR” to the destination memory space where the BIOS image is being reconstructed....

RBU Packet Copied

```

DFF23C78 mov     ecx, ds:g_pktSizeMinusHdrSize
DFF23C7E dec     edi
DFF23C7F imul   edi, ecx
DFF23C82 add     edi, 101000h
DFF23C88 cmp     dword ptr [edi], 'ROEG'
DFF23C8E jz     short loc_DFF23C94

```

```

DFF23C90
DFF23C90 loc_DFF23C90:
DFF23C90 xor     eax, eax
DFF23C92 pop     edi
DFF23C93 retn

```

```

DFF23C94
DFF23C94 loc_DFF23C94:
DFF23C94 mov     edx, ds:g_hdrSize
DFF23C9A push   esi
DFF23C9B shr     edx, 2
DFF23C9E lea   esi, [eax+edx*4]
DFF23CA1 mov     eax, ecx
DFF23CA3 shr     ecx, 2
DFF23CA6 rep movsd                ; buffer overflow
DFF23CA8 mov     ecx, eax

```

- Eventually the portion of the BIOS image described by the RBU packet is copied to the reconstruction location in memory.
- Notice the size parameter (ecx) for the inline memcpy (rep movsd) is derived from attacker data (g_pktSizeMinusHdrSize).

RBU Packet Parsing Vulnerability

```
kpr_ptr = *(_DWORD *)a1;
if ( *(_DWORD *)(*(_DWORD *)a1 + 12) == g_pktSetId
    && *(_WORD *)(kpr_ptr + 16) // pktNum
    && (copy_dest = (void *)(g_pktSizeMinusHdrSize * (*(_WORD *)kpr_ptr + 16) - 1) + 0x101000),
    *(_DWORD *)(g_pktSizeMinusHdrSize * (*(_WORD *)kpr_ptr + 16) - 1) + 0x101000) == 'ROEG' )
{
    copy_src = (const void *)(kpr_ptr + 4 * ((unsigned int)g_hdrSize >> 2));
    memcpy(copy_dest, copy_src, 4 * ((unsigned int)g_pktSizeMinusHdrSize >> 2));
    result = 1;
}
```

- In fact, the copy destination and copy source are also both derived from attacker data read in from the current rbu_packet.
- This is an exploitable buffer overflow.

Lack of Mitigations

- **System Management Mode is missing all of the traditional exploit mitigations you would expect to find in modern applications.**
- **No ASLR, NX, stack canaries, and so on....**
- **This means we can pursue any target with our overwrite, such as the return address for the rbu packet copying function...**

Exploiting the Vulnerability

```
kpr_ptr = *(_DWORD *)a1;
if ( *(_DWORD *)(*(_DWORD *)a1 + 12) == g_pktSetId
    && *(_WORD *)kpr_ptr + 16 // pktNum
    && (copy_dest = (void *)g_pktSizeMinusHdrSize * (*(_WORD *)kpr_ptr + 16) - 1) + 0x101000,
        *(_DWORD *)g_pktSizeMinusHdrSize * (*(_WORD *)kpr_ptr + 16) - 1) + 0x101000 == 'ROEG' )
{
    copy_src = (const void *)kpr_ptr + 4 * ((unsigned int)g_hdrSize >> 2);
    memcpy(copy_dest, copy_src, 4 * ((unsigned int)g_pktSizeMinusHdrSize >> 2));
    result = 1;
}
```

- There are actually a number of constraints on the RBU packet data that make exploiting this buffer overflow tricky.

Constraints Overview

- **Our copy destination must point to an area pre-initialized with the “GEOR” string.**
- **Copy_dest must be lower in memory than the return address.**
- **We can’t overwrite too much lest we die in the inline memcpy and never return.**
- **Copy source must be positioned such that attacker controlled data in the address space ends up overwriting the saved return address.**
- **Others.....**

More Problems

```
copy_dest = ((rbu_packet.pktSize << 10) - (rbu_packet.hdrSize << 4)) * ((rbu_packet.pktNum-1) + 0x101000);  
copy_src = rbu_packet.pktSize * (rbu_packet.hdrSize << 2);  
copy_size = (rbu_packet.pktSize << 10) - (rbu_packet.hdrSize << 4)
```

- **The source, destination and size operands are all derived from the same `rbu_packet` members.**
- **Changing one operand, changes the others.**
- **All of the constraints previously mentioned must be satisfied.**
- **Exploitation of this vulnerability can be modeled as a constraints solving problem.**

Constraints Corollary

```

pktSizeMinusHdrSize = (*( _WORD *) )(KPRstruct + 4) << 10) - g_hdrSize; // pktSize - hdrSize
totalDataSize = pktSizeMinusHdrSize * (totPkts - 1);
g_pktSizeMinusHdrSize = pktSizeMinusHdrSize;
if ( totalDataSize <= 0x800000 )
{
    memset((void *)0x101000, 0, 4 * (totalDataSize >> 2));
    endOfData = 4 * (totalDataSize >> 2) + 0x101000;
    for ( i = totalDataSize & 3; i; --i )
        *MK_FP(__ES__, endOfData++) = 0;
    for ( j = 0x101000u; j <= totalDataSize + 0x101000; j += g_pktSizeMinusHdrSize )
        *( _DWORD *)j = 'ROEG';
    result = 1;
}

```

- An initialization routine populates the “GEOR” string at the expected copy dest location under “normal” circumstances.
- We skip this normal initialization routine by setting `rbu_packet.totPkts` to 1 (which as you can see from the above code, sets `totalDataSize=0`). If we don't, the large `memset` and GEOR writing operation performed by the initialization routine are problematic.
- This means the expected “GEOR” string won't naturally occur in the address space, and we will have to inject it somehow to satisfy the `*copy_dest = “GEOR”` constraint.

Faux GEOR

```
kpr_ptr = *(_DWORD *)a1;
if ( *(_DWORD *)(*(_DWORD *)a1 + 12) == g_pktSetId
    && *(_WORD *)kpr_ptr + 16 // pktNum
    && (copy_dest = (void *)(g_pktSizeMinusHdrSize * (*(_WORD *)kpr_ptr + 16) - 1) + 0x101000),
    *(_DWORD *)g_pktSizeMinusHdrSize * (*(_WORD *)kpr_ptr + 16) - 1) + 0x101000) == 'ROEG' )
{
    copy_src = (const void *)kpr_ptr + 4 * ((unsigned int)g_hdrSize >> 2);
    memcpy(copy_dest, copy_src, 4 * ((unsigned int)g_pktSizeMinusHdrSize >> 2));
    result = 1;
}
```

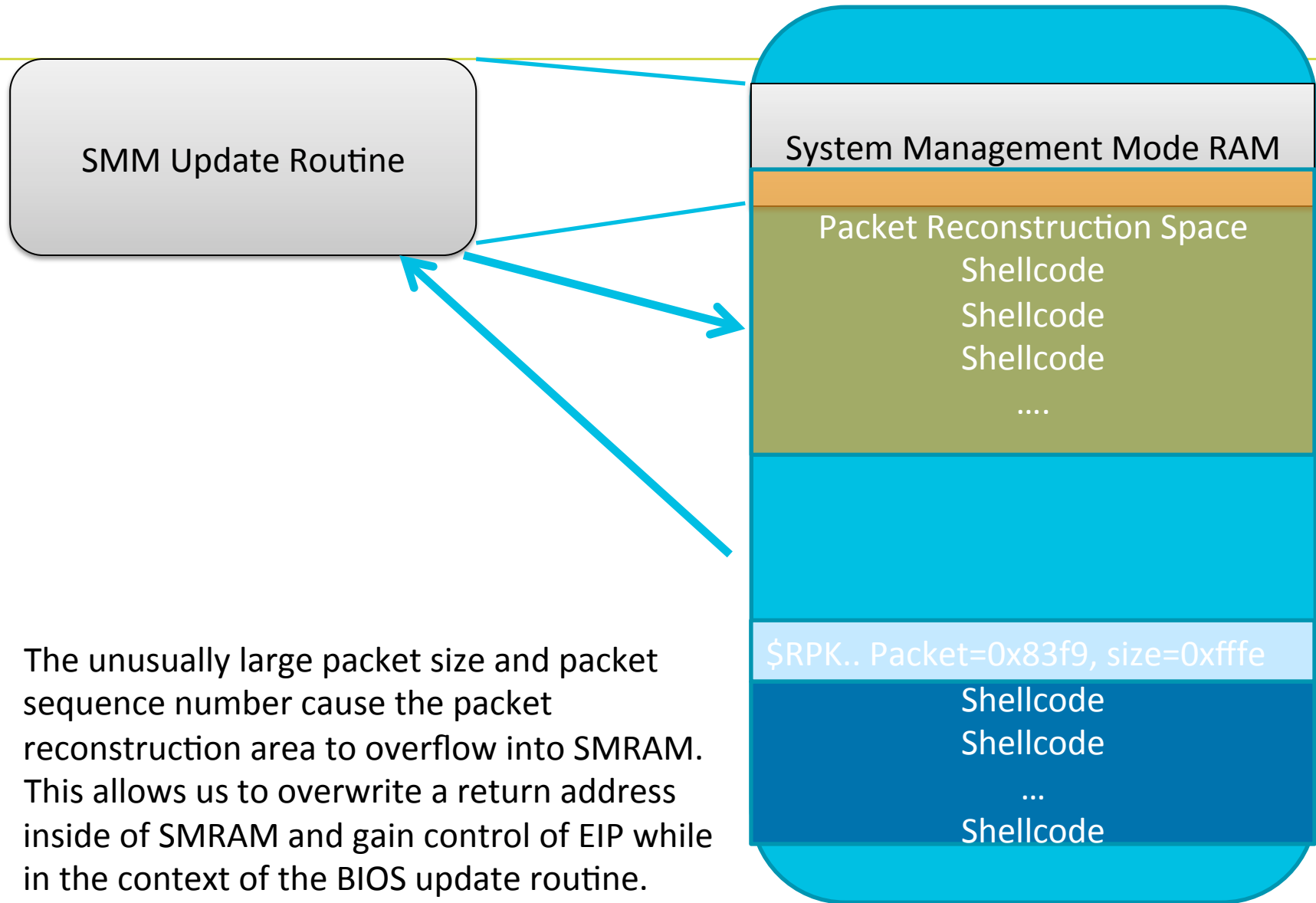
- The vulnerable memcpy will only execute if the copy destination points to a location containing this GEOR string.
- We use a Windows kernel driver that performs memory mapped i/o to write the GEOR string as high up in memory as possible, to allow us to force copy_dest to be within striking distance of the return address we want to overwrite.
- Like the BIOS update process, we are abusing the fact RAM remains intact during a soft reboot so the GEOR strings we wrote will remain in the address space.

RBU Packet Solution

- With all those constraints in mind, we brute force an `rbu_packet` configuration that allows us to pass the sanity checks and overwrite the return address gracefully.

```
$ ./rbusolver
found success with pktNum=83f9, pktSize=fffe
will write from dbf05000 to dff04800
g_pktSizeMinusHdrSize: 3fff800
g_min_copydest: dbf05000, g_pktnum: 83f9, g_pktsize: fffe
```

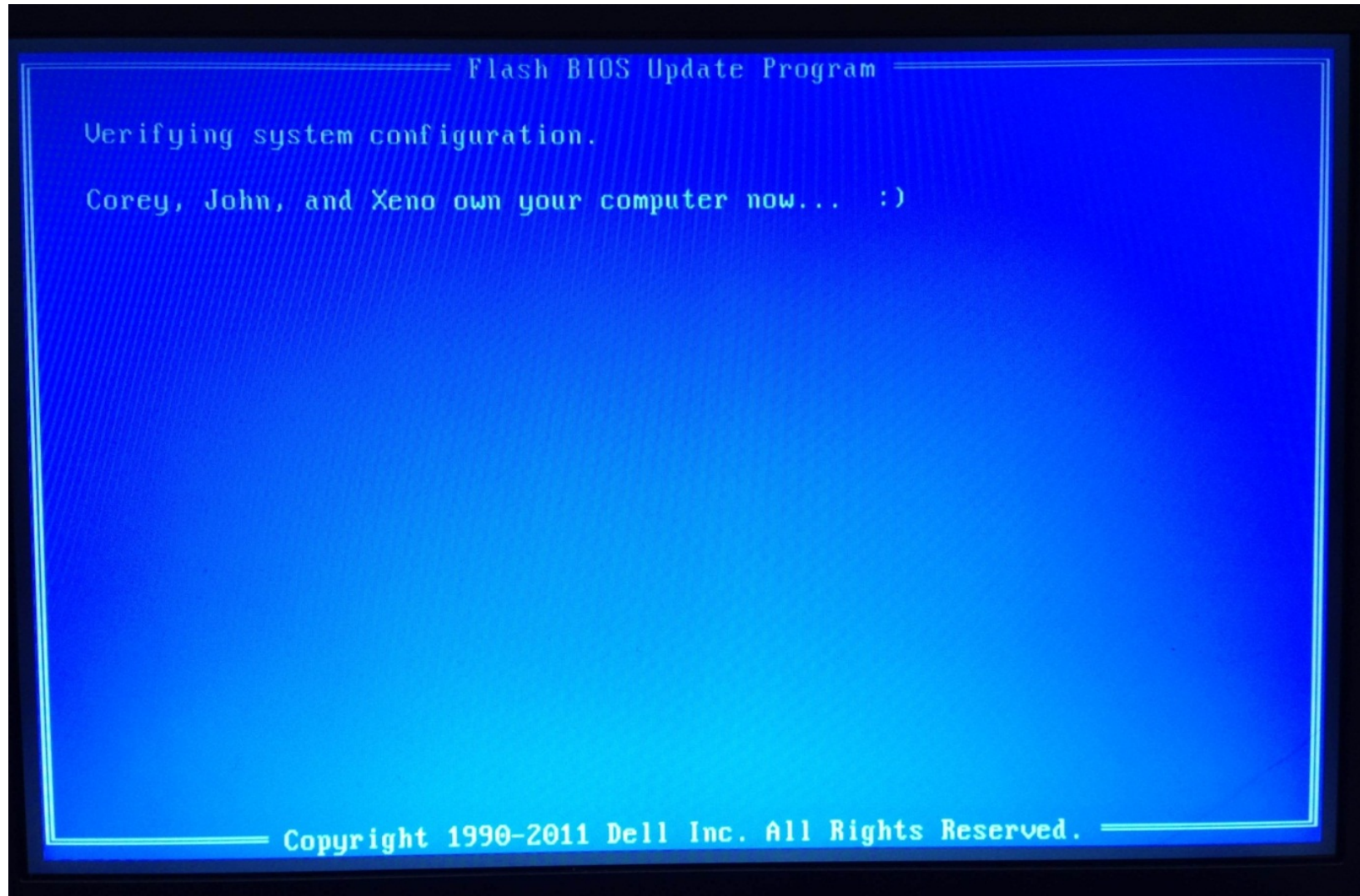

Malicious BIOS Update



- The unusually large packet size and packet sequence number cause the packet reconstruction area to overflow into SMRAM.
- This allows us to overwrite a return address inside of SMRAM and gain control of EIP while in the context of the BIOS update routine.

PoC Demonstration Video

http://youtu.be/V_ea21CrOPM



Vulnerability Conclusion

- **The vulnerability allows an attacker to take control of the BIOS update process and reflash the BIOS with an arbitrary image despite the presence of signed bios enforcement.**
- **Because BIOS is charged with instantiating System Management Mode (SMM), control of the BIOS implies complete control of SMM.**
- **Once the attacker has complete control of BIOS and SMM, really Bad Things can start to happen...**

How can we detect attackers in the BIOS?

Trusted Computing Group (TCG)

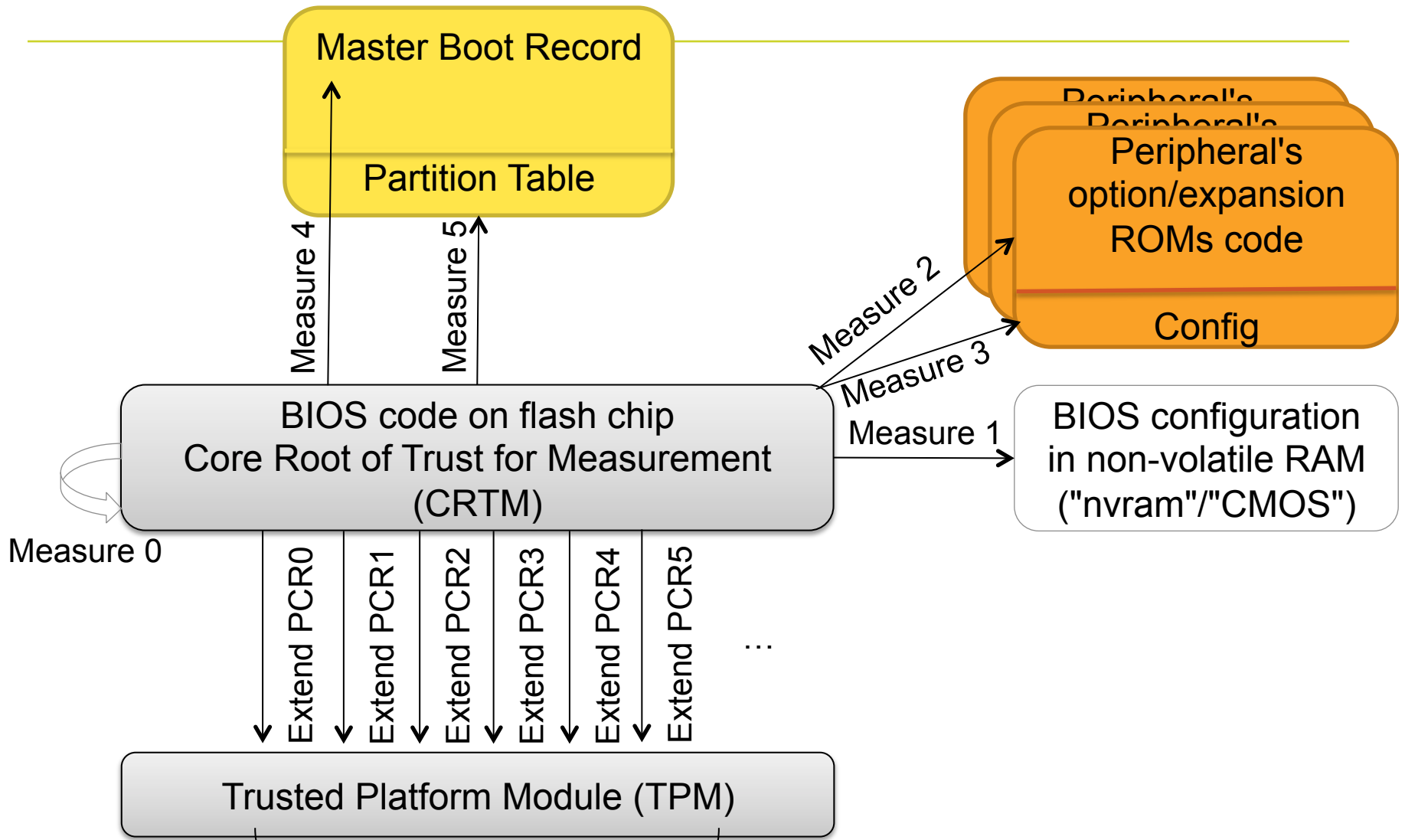
Static Root of Trust for Measurement (SRTM)

- In the PC Client Specification[2], the TCG lays out a strategy for obtaining measurements of critical boot-time components
 - This *should* detect things like MBR-based bootkits, or even BIOS attackers
- The SRTM is a chain of trust which is built up at boot time from the BIOS measuring itself, and measuring every other bit of executable code before control is passed to that code
 - Measurements stored in TPM, discussed shortly
- All these measurements are typically gained "for free" when the BIOS is configured to enable the TPM

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.
 - $\text{Extend_PCR0}(\text{data}): \text{PCR0}_{\text{new}} = \text{SHA1}(\text{PCR0}_{\text{old}} \parallel \text{SHA1}(\text{data}))$

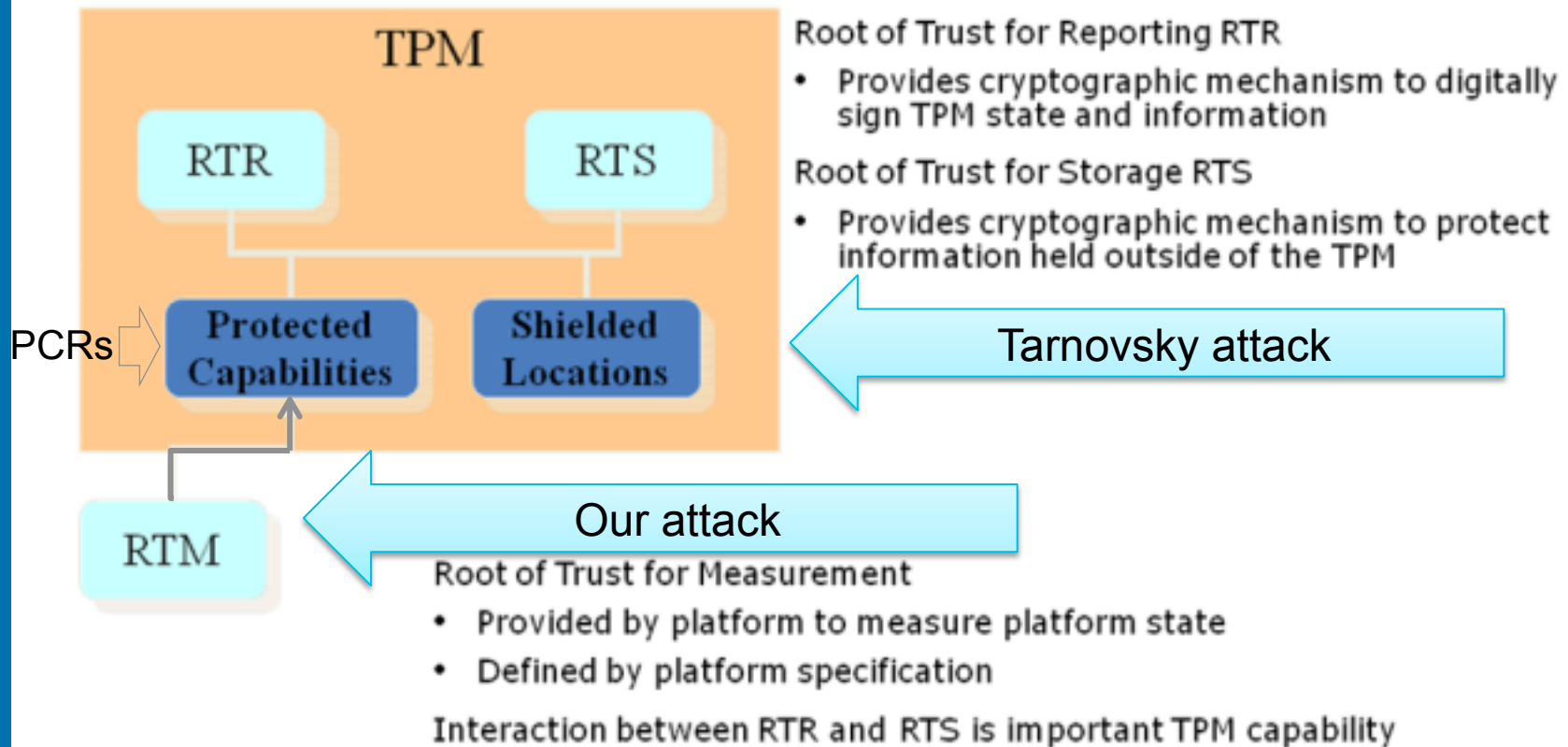
Example Measured Boot ("measured boot" != UEFI "secure boot")



This collection of measurements going forward is the Static Root of Trust for Measurement (SRTM)

All roots of trust are not created equal

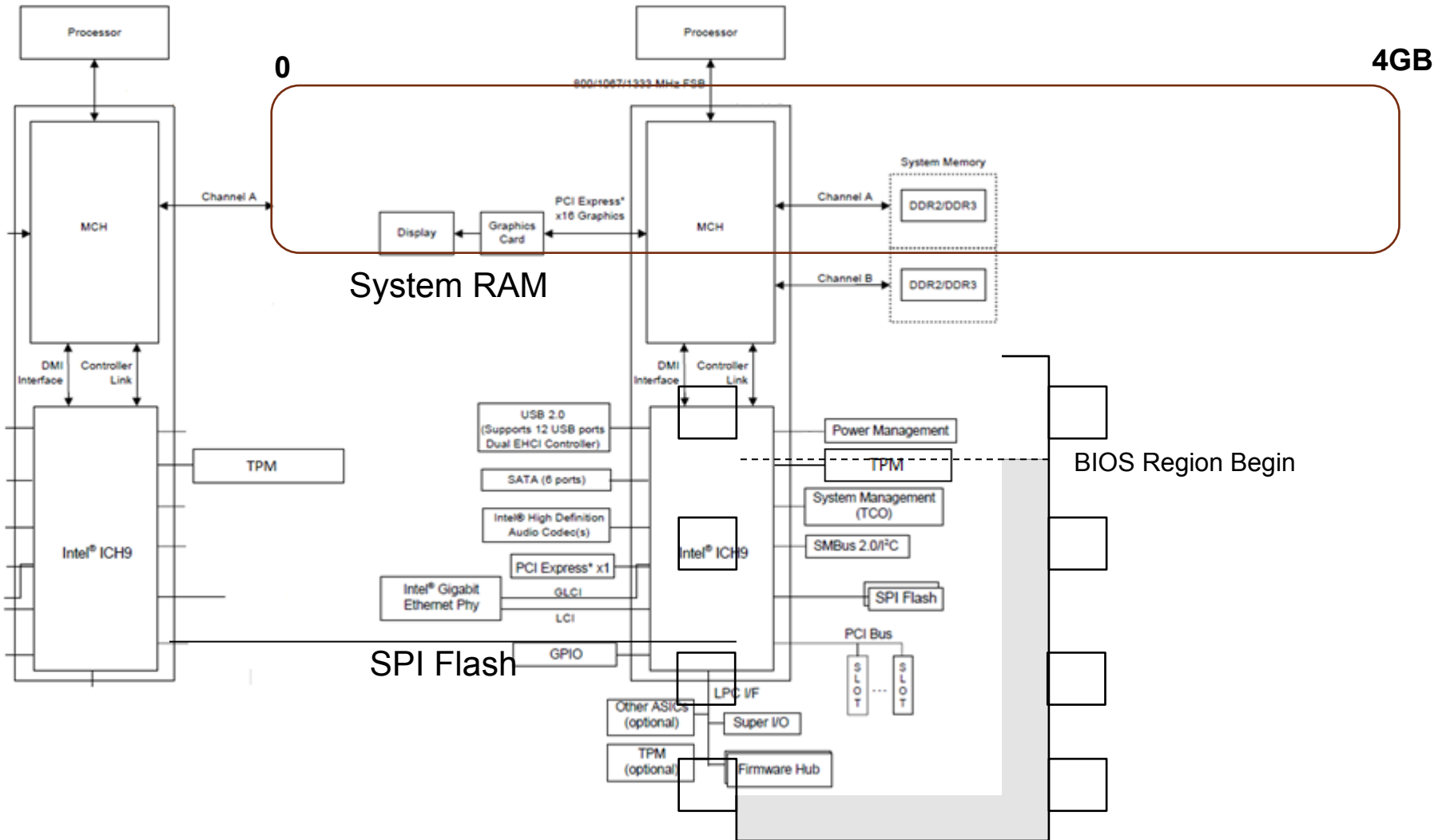
Functional TPM Diagram



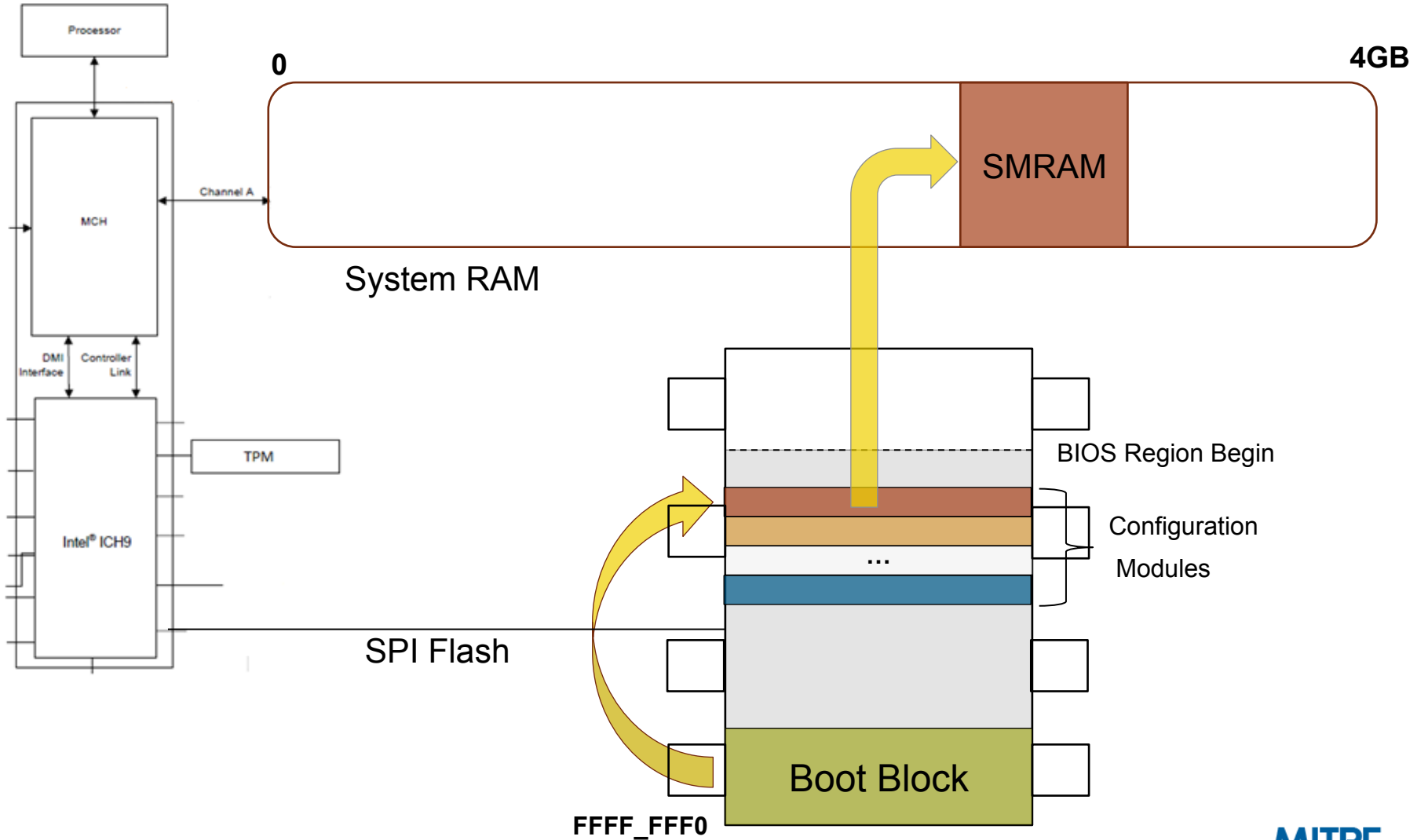
Base diagram from

<http://www.intel.com/content/dam/doc/white-paper/uefi-pi-tcg-firmware-white-paper.pdf> MITRE

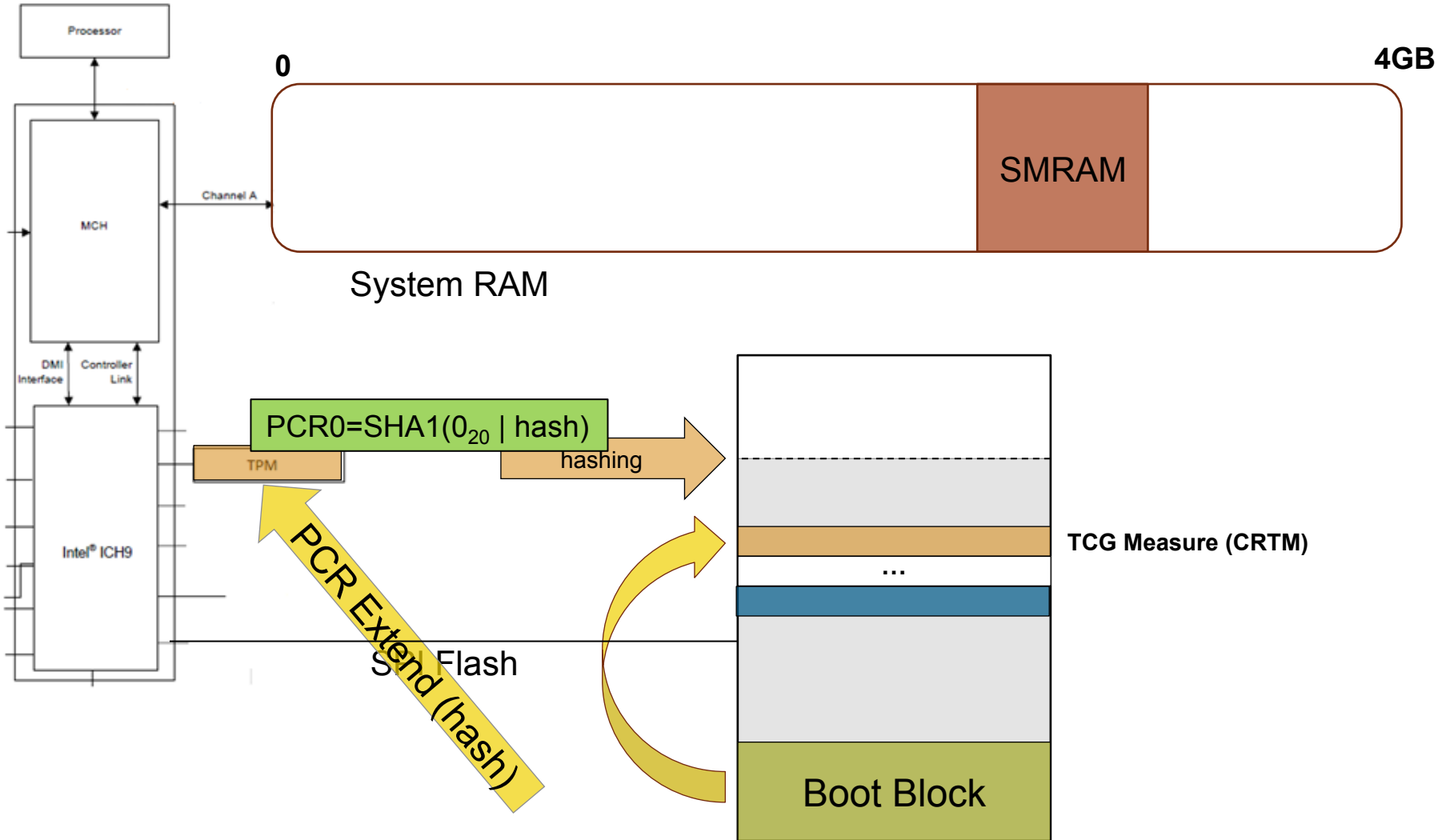
Q45 Express Chipset



Typical E6400 boot sequence 1



Typical E6400 boot sequence 2



General Problems with PCR Hashes

■ Opaqueness

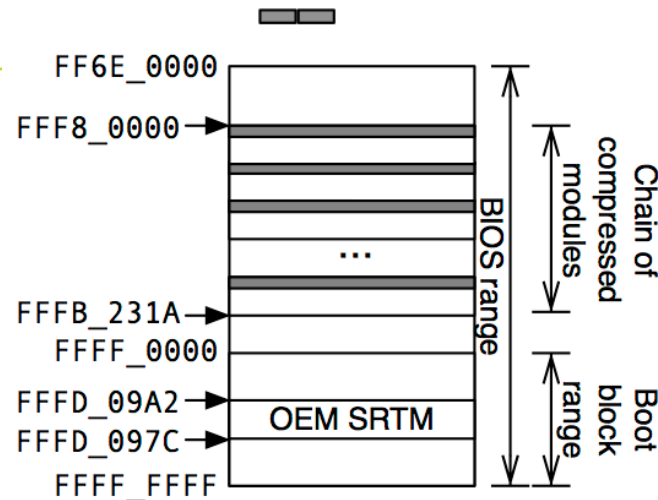
- No golden set of PCRs is provided by the OEM.
- No description of what is *actually* being measured and incorporated into the PCR values.¹
- Homogeneous systems can have different PCR values.²
- Duplicate PCR values are unexpected if they're measuring different data...

■ 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 (CRTM) Measurement



- PCR0 should contain a measurement of the CRTM 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 (.15%) in the BIOS range are incorporated, including:
 - The first 64 bytes of the 42 modules.
 - Two 8 byte slices at 0xDF4513C0 and 0xDF4513C7.
 - The CRTM is not incorporated at all.

*BIOS Base is located at FFE6_0000

Implications of the weak SRTM

- **Measurements for things like PCI option ROMs and BIOS configuration are not actually captured.**
- **We can modify the *majority* of the E6400 BIOS without changing any of the PCR values.**
 - Yuriy Bulygin presented a similar discovery at CanSecWest 2013 regarding his ASUS P8P67[3], but did not investigate the details of what information was being measured into what PCRs
- **What if we want to modify any part of the BIOS under the assumption that the entire BIOS is being measured?**
 - Like the splash-screen or the code that instantiates SMM?

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

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

Really Bad Things: Firmware Rootkits

- Created two proof of concept firmware rootkits.
- Each is installed programmatically; no hardware modification required.

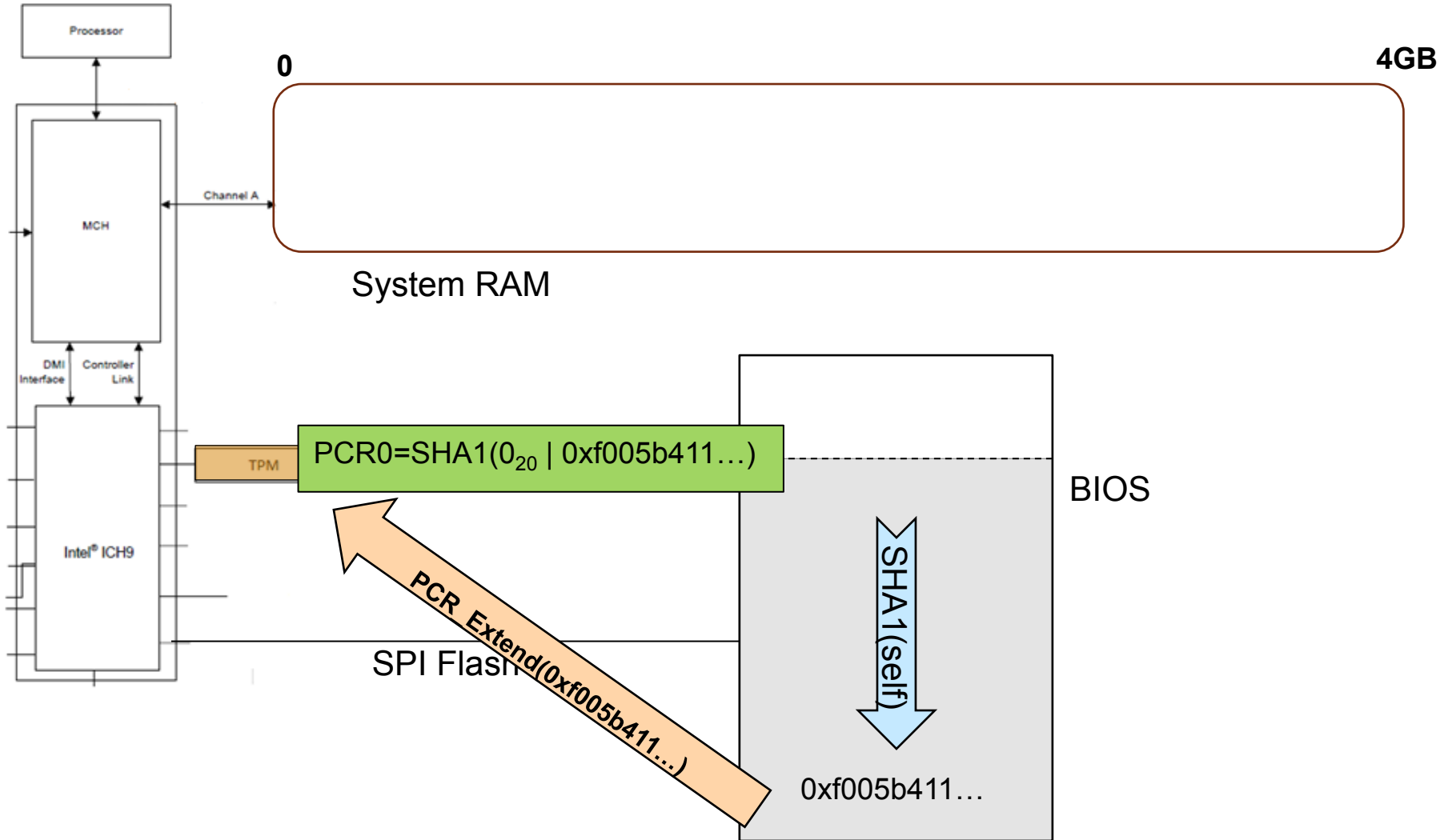
1. Tick

- Persistent stealth malware
- Called the Tick because it “embeds” itself in the firmware
- Evades detection by forging PCRs
- Once in place, can modify any other portion of the BIOS and inject itself into SMRAM.

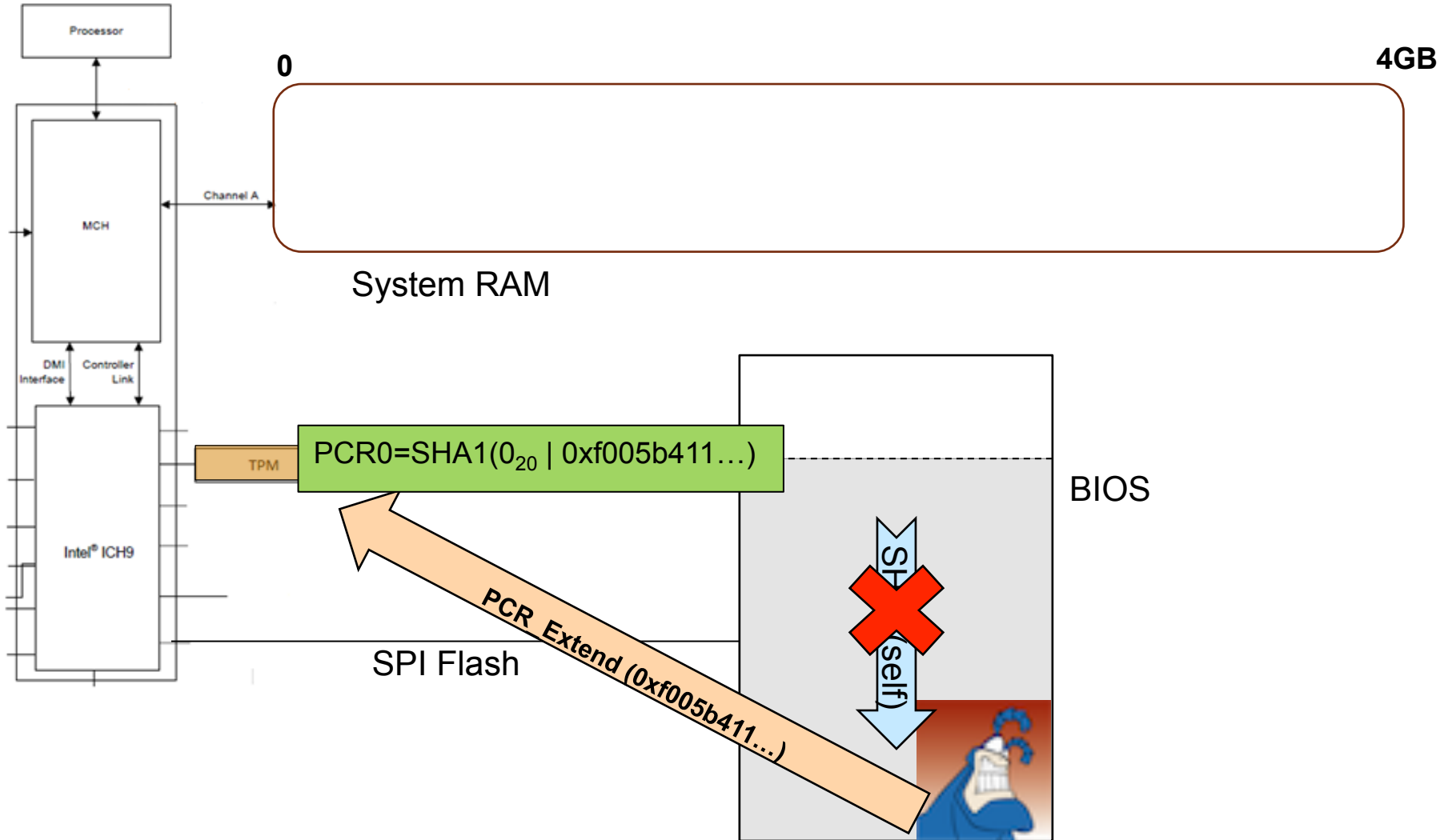
2. Flea

- Same stealth/persistence capabilities as the Tick
- Able to persist even beyond BIOS updates
 - “jumps” from one revision to the next

Normal BIOS PCR0 Measurement



PCR0 Measurement with a Tick



Tick Demo Video

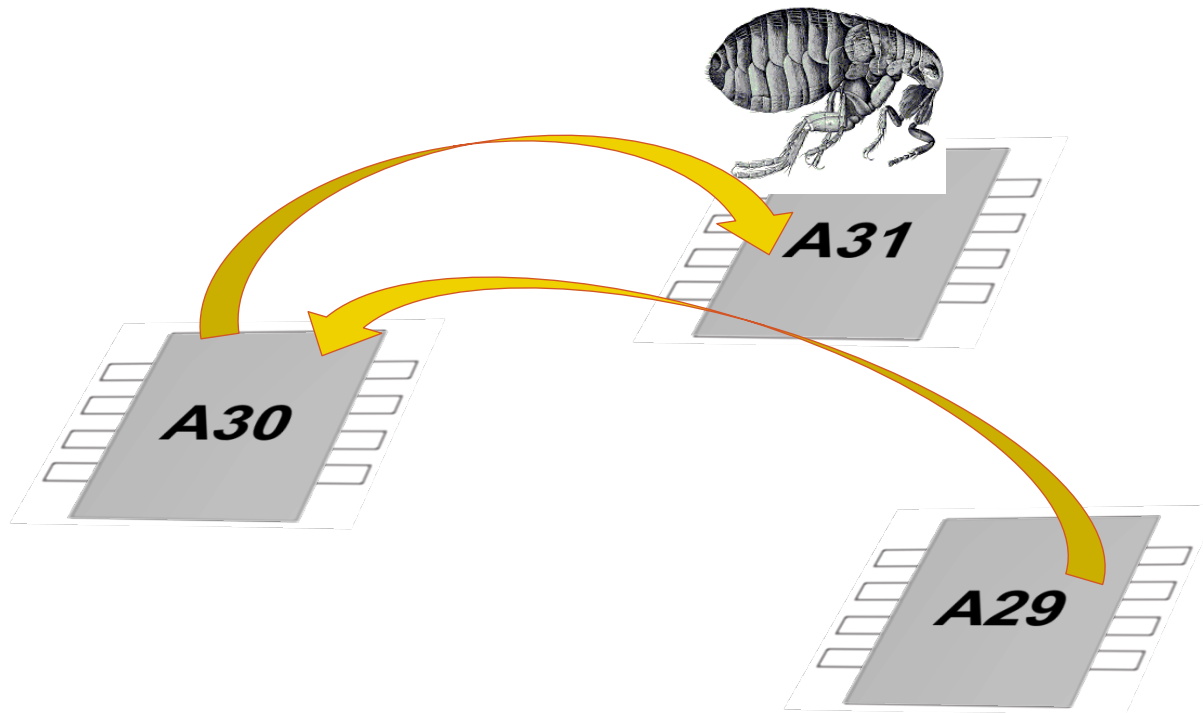
<http://www.youtube.com/watch?v=S0IRcm3jvFo>



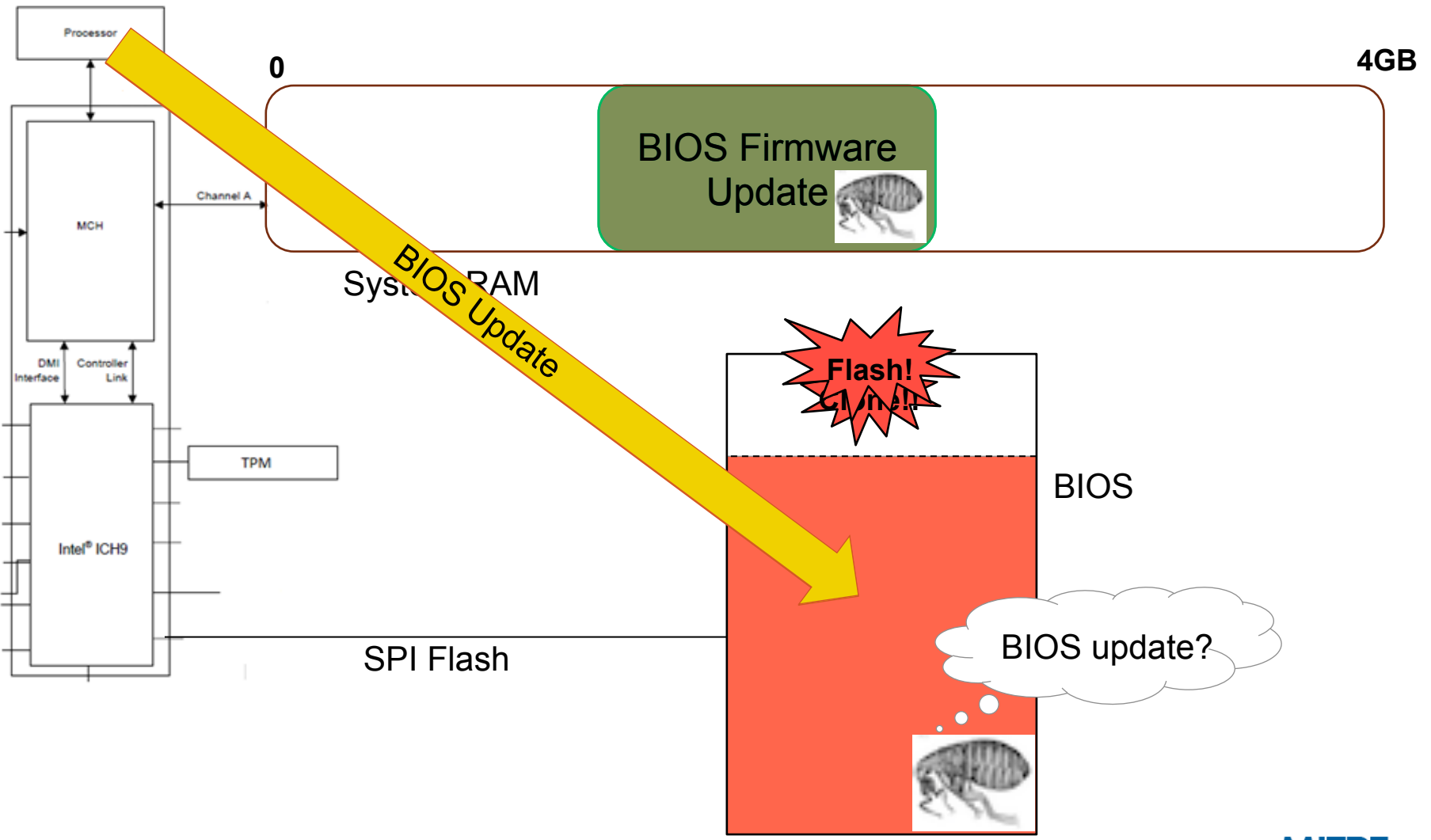
The Tick from http://th04.deviantart.net/fs6/PRE/i/2005/087/1/b/The_Tick_by_emucoupons.png

The Flea

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

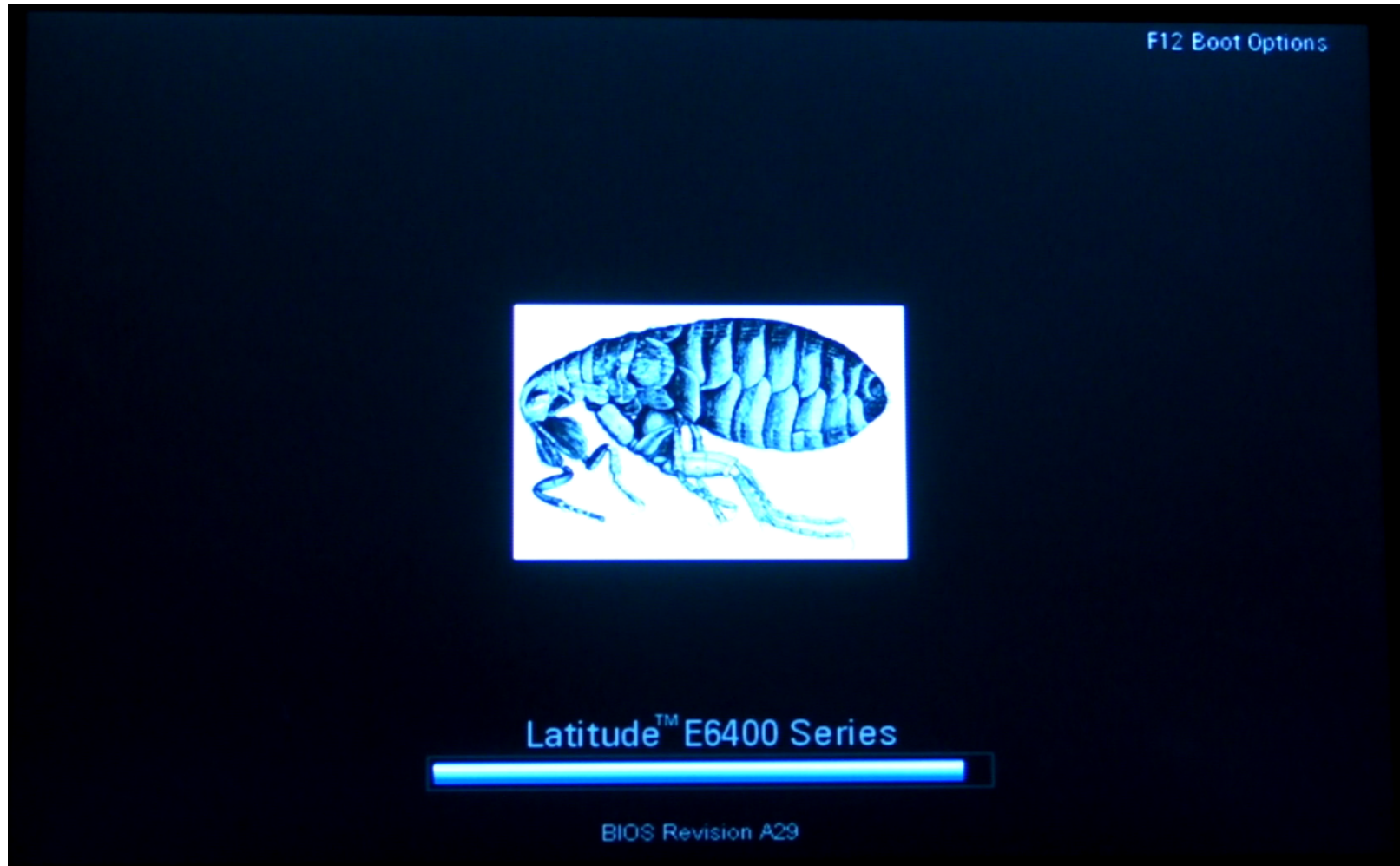


The Flea



Flea Demo Video

<http://www.youtube.com/watch?v=fvQjhgzxHR8>



The Flea – Robert Hooke – Micrographia - 1665 ;)

Countermeasure: Timing-Based Attestation "BIOS Chronomancy"

- **The fundamental premise:**
 - "Build your software so that if its 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-checking 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 justice. A timeline of related work here:**
 - bit.ly/11xEmlV (timeglider.com link)

Components of All Self-Checks

- **Read your own data**
 - Incorporated into checksum so if it changes the checksum changes
- **Read your own data pointer and instruction pointer**
 - Indicates where in memory the code itself is reading and executing
- **Nonce/PseudoRandom Number(PRN)**
 - Prevent trivial replay, decrease likelihood of precomputation due to storage constraints
- **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

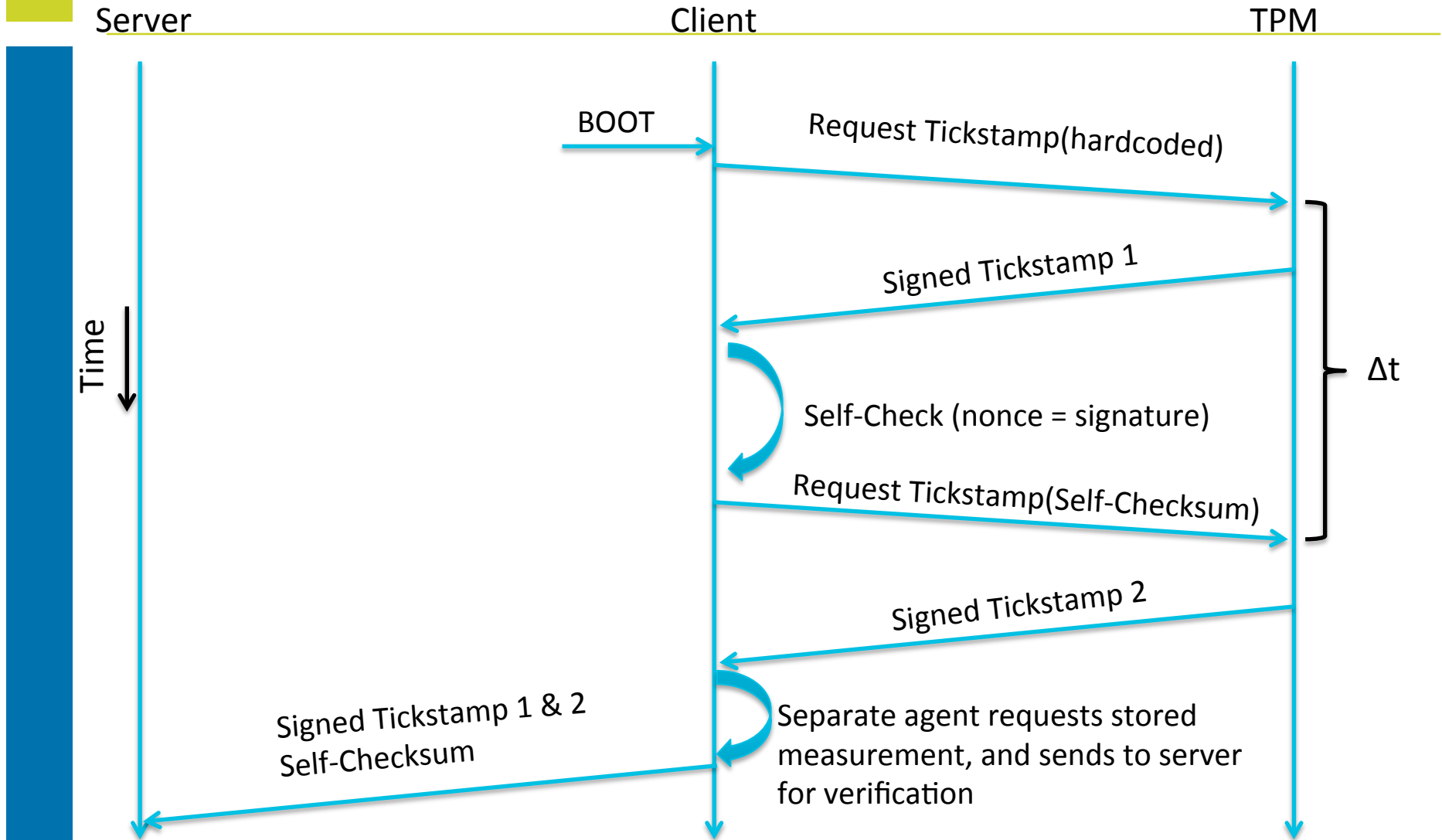
Simplified Selfcheck()

```
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++;  
    }  
}
```

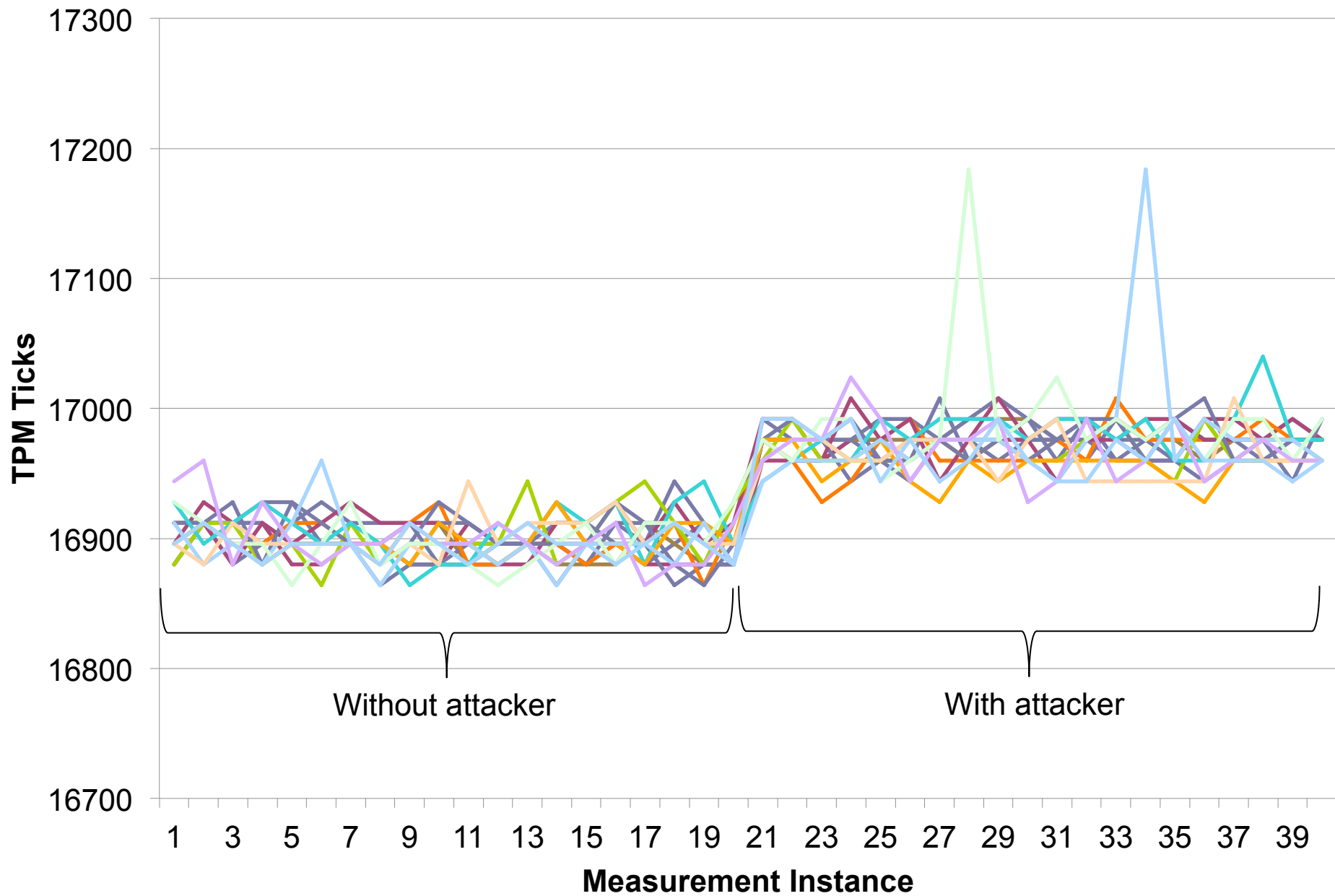

Simplified Selfcheck() Forgery

```
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++;
    }
}
```

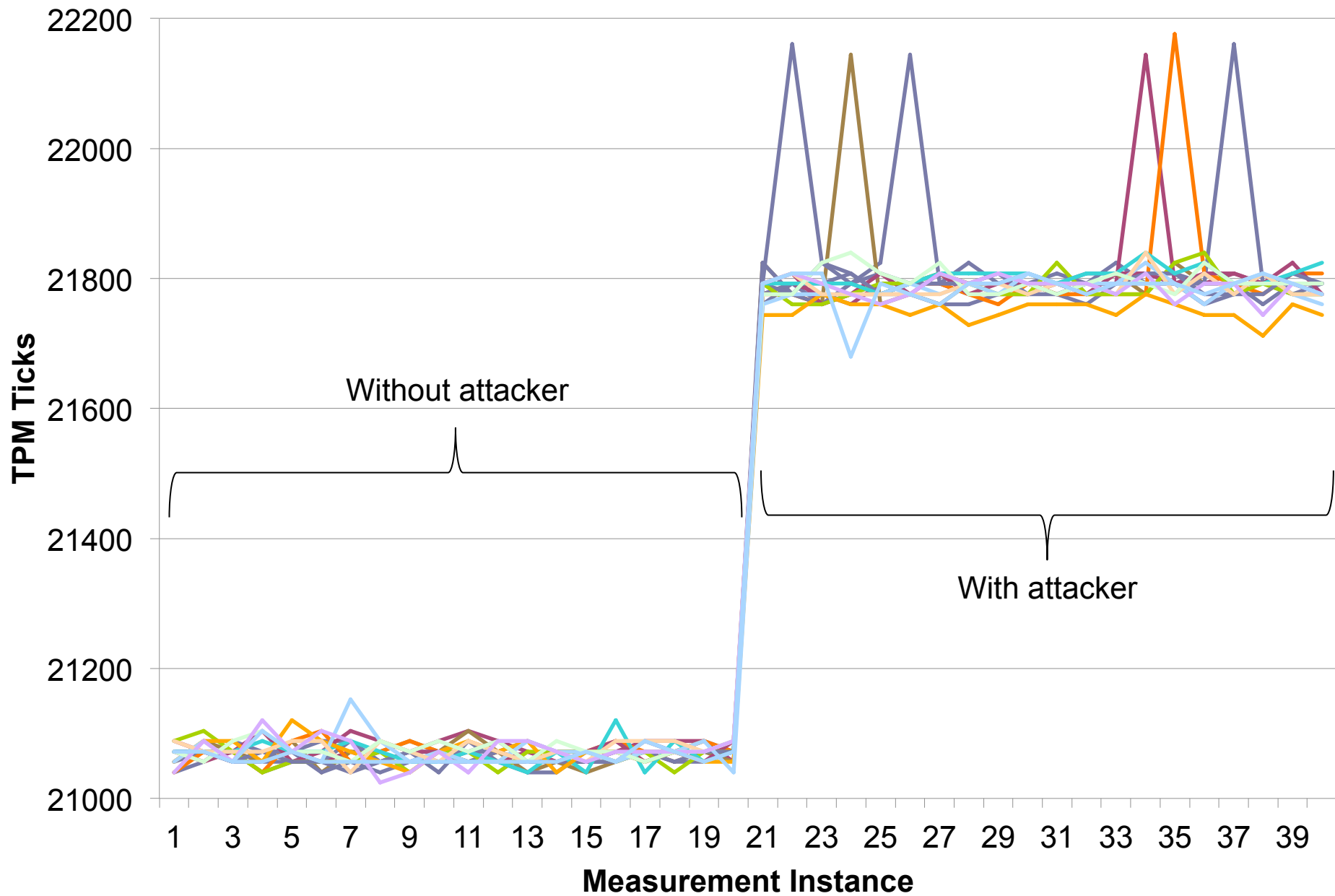
TPM-Timing Based Implementation (BIOS Boot-Time)



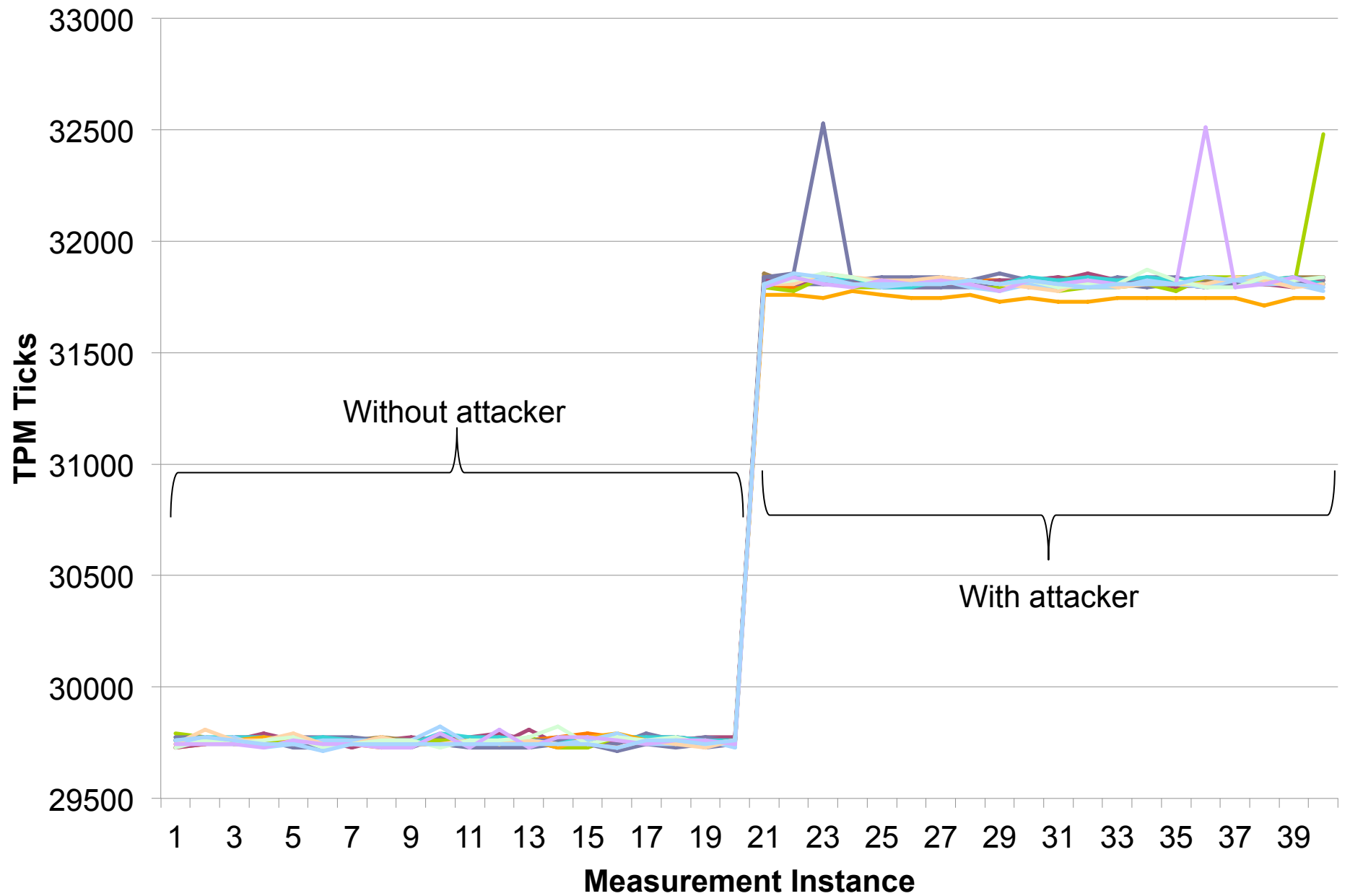
18 E6400s with customized BIOS Chronomancy firmware 625k self-check iterations



18 E6400s with customized BIOS Chronomancy firmware 1.25M self-check iterations

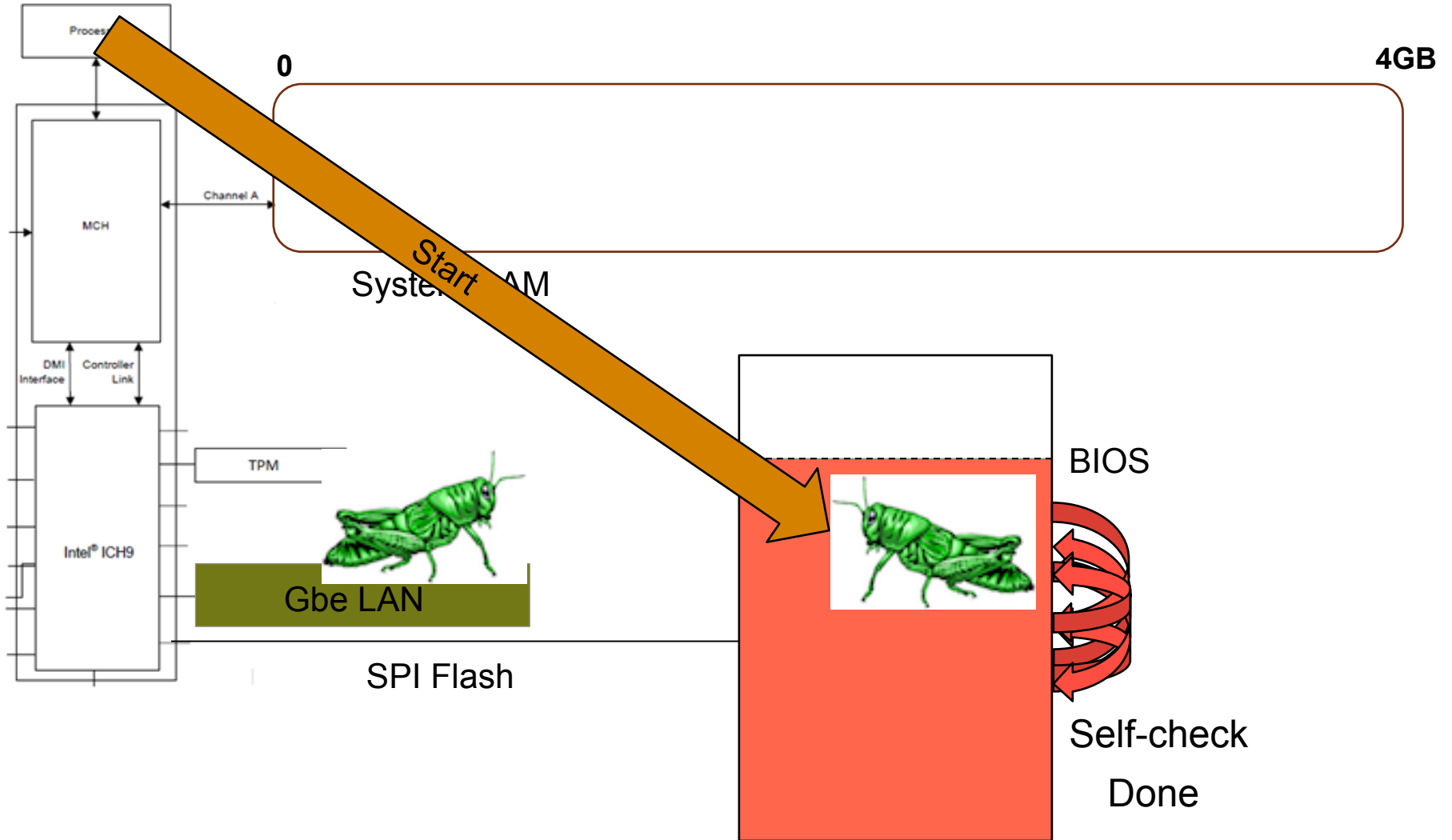


18 E6400s with customized BIOS Chronomancy firmware 2.5M self-check iterations



Is BC perfect? NOPE! TOCTOU attackers are ongoing work

Enter the "flash hopper" :P



Conclusion

- **Assume attackers can get in**
- **Bad things happen when attackers get in**
- **Trusted Computing implementations should not be trusted implicitly, they should only be trusted if they are open for independent review (and someone has actually reviewed them ;)**
 - It's ironic that they're overwhelmingly closed source & proprietary. (Even academics don't usually post their code for open review!¹)
- **As long as the CRTM is implemented in writable firmware, ticks and fleas will mean that you can't trust any of your SRTM.**
 - And as ITL has shown, a TXT-based Dynamic RTM *can* depend, in a security-critical way, on the BIOS/SRTM-generated info [5][6][7]
 - If you're not going to be using BC, you better be using super simple true ROM CRTM code

¹ Our code for our self-check is at <http://code.google.com/p/timing-attestation>

But wait...there's just One More Thing!

- **We have released Copernicus ("Question your assumptions!"), a tool to check for basic BIOS/SMM security vulnerabilities**
 - http://www.mitre.org/work/cybersecurity/blog/cyber_tools_butterworth1.html
 - Checks configuration bits to see if the BIOS/SMM is writable, ala Yuriy's talks[3][4]
- **Dumps BIOS image to allow diffing & analysis**
 - Can detect Rakshasa, last year's "undetectable" BIOS malware[7] ;)
- **Government organizations:**
 - Talk to us about running this in your environment (pushable via HBSS - but the data goes to a different server, not ePO)
- **Commercial security vendors:**
 - Contact us to incorporate Copernicus's capabilities into your kernel/hypervisor agents. We want maximum availability of this capability. MITRE is a not-for-profit company that only works for the government in the public interest.

Questions?

- **jbutterworth, ckallenberg, xkovah @ mitre.org**
- **To learn more about TPMs, Reverse Engineering, and other deep security stuff, check out**
- **<http://OpenSecurityTraining.info/Training.html>**
 - John will be creating BIOS/UEFI classes this coming year, follow @OpenSecTraining to keep up with news
 - And if you already know the stuff, take the materials and teach it!
- **Also Corey released OpenTPM so you too can play around with and learn more about the TPM**
- **<http://code.google.com/p/opentpm/>**

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/>

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

- PCR[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.
- $\text{PCR}[1..3] = \text{SHA1}(0x00_{20} \parallel \text{SHA1}(0x00))$

Future Work: Combat TOCTOU



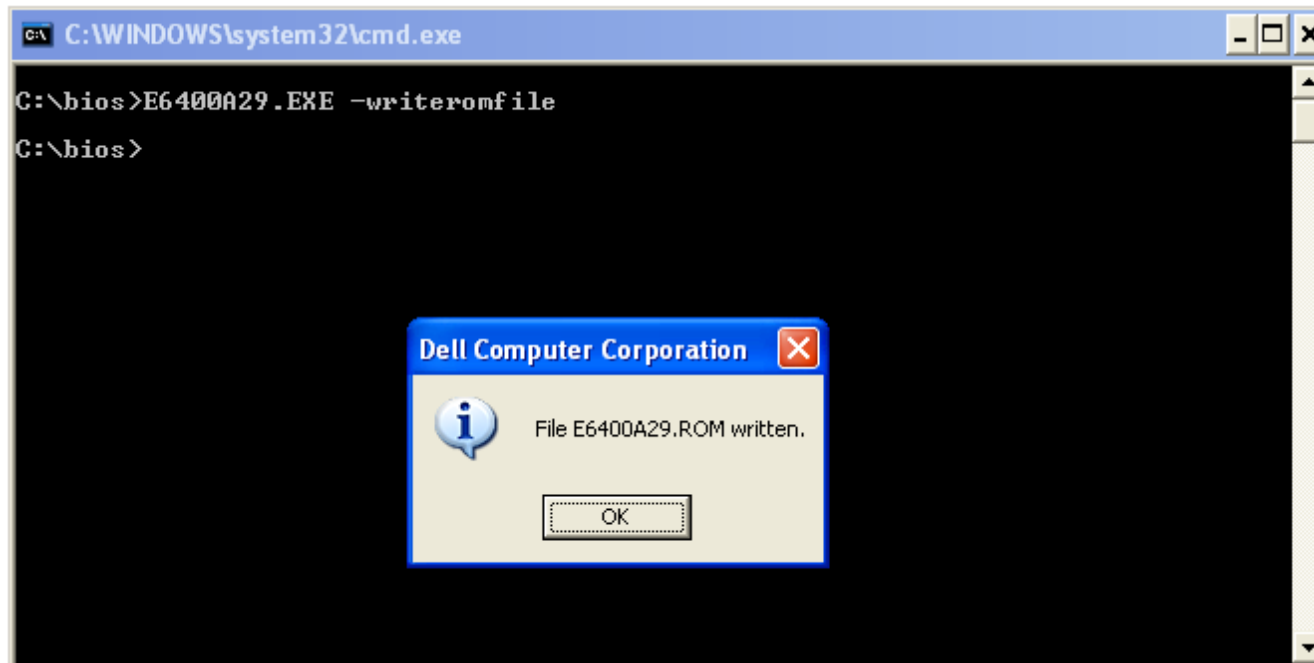
Attacker moves out of the way, just in time

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 recognized

BIOS Acquisition

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



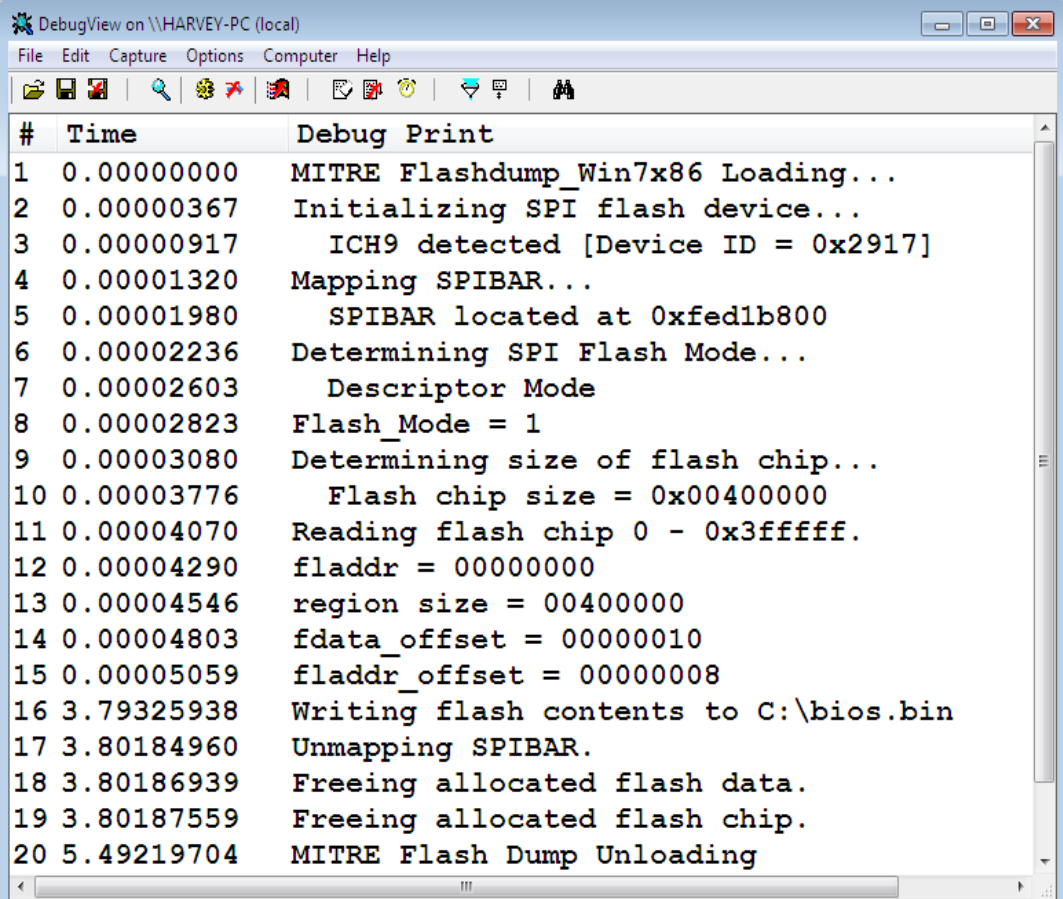
- **Dependent on manufacturer**

- 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

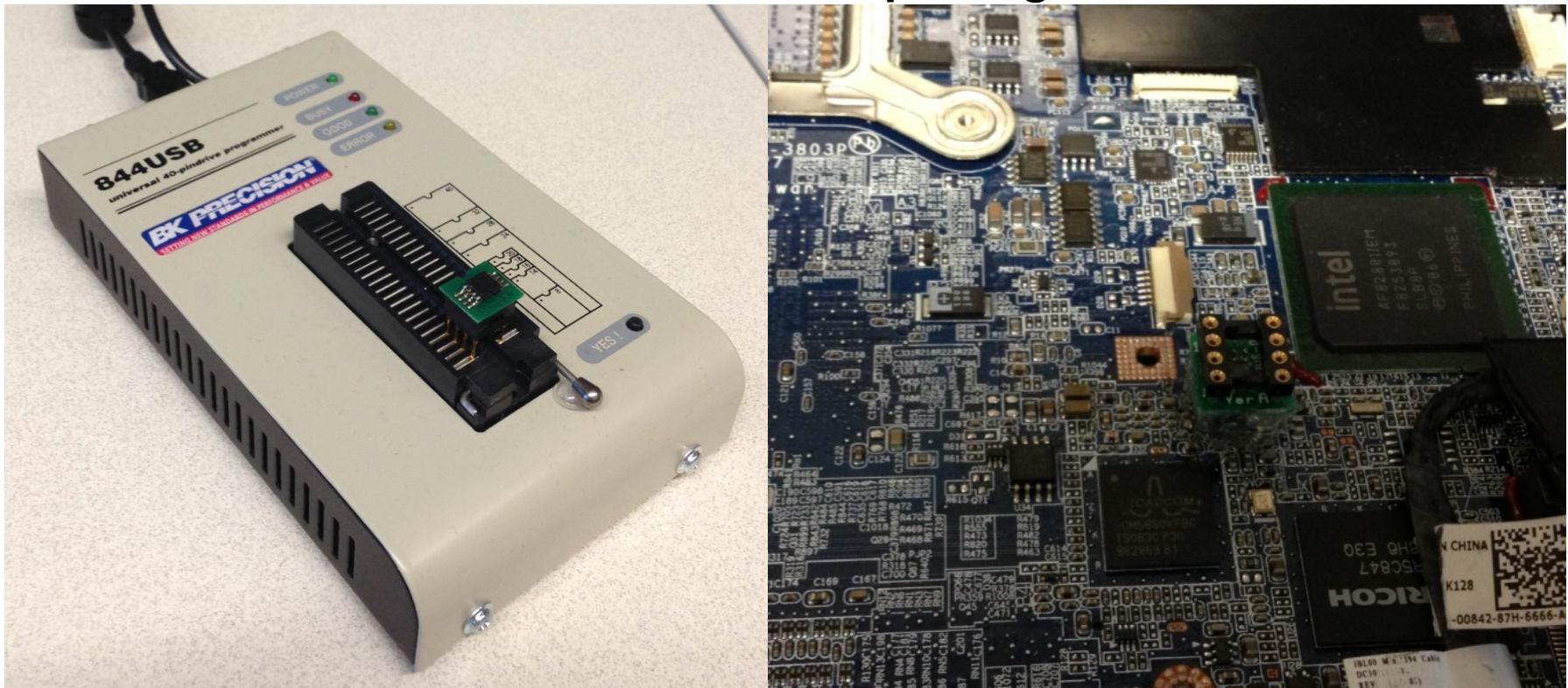


The screenshot shows a DebugView window titled "DebugView on \\HARVEY-PC (local)". The window displays a list of debug prints with columns for "#", "Time", and "Debug Print". The logs show the process of loading and dumping BIOS data from a flash chip.

#	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 - 0x3ffffff.
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

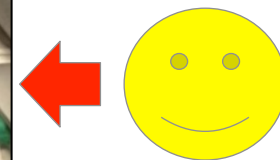
BIOS Acquisition

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



- 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.

BIOS Modification: Access Controls 2

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.										
	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:										
3:2	<table border="1"> <thead> <tr> <th>Bits 3:2</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>00b</td> <td>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</td> </tr> <tr> <td>01b</td> <td>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.</td> </tr> <tr> <td>10b</td> <td>Prefetching and Caching enabled. This mode is used for long sequences of short reads to consecutive addresses (i.e., shadowing).</td> </tr> <tr> <td>11b</td> <td>Reserved. This is an invalid configuration, caching must be enabled when prefetching is enabled.</td> </tr> </tbody> </table>	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.
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 PIBSET .										
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