Advanced x86:
BIOS and System Management Mode Internals
Trusted Computing Technologies

Xeno Kovah && Corey Kallenberg

LegbaCore, LLC

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

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

You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

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

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

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

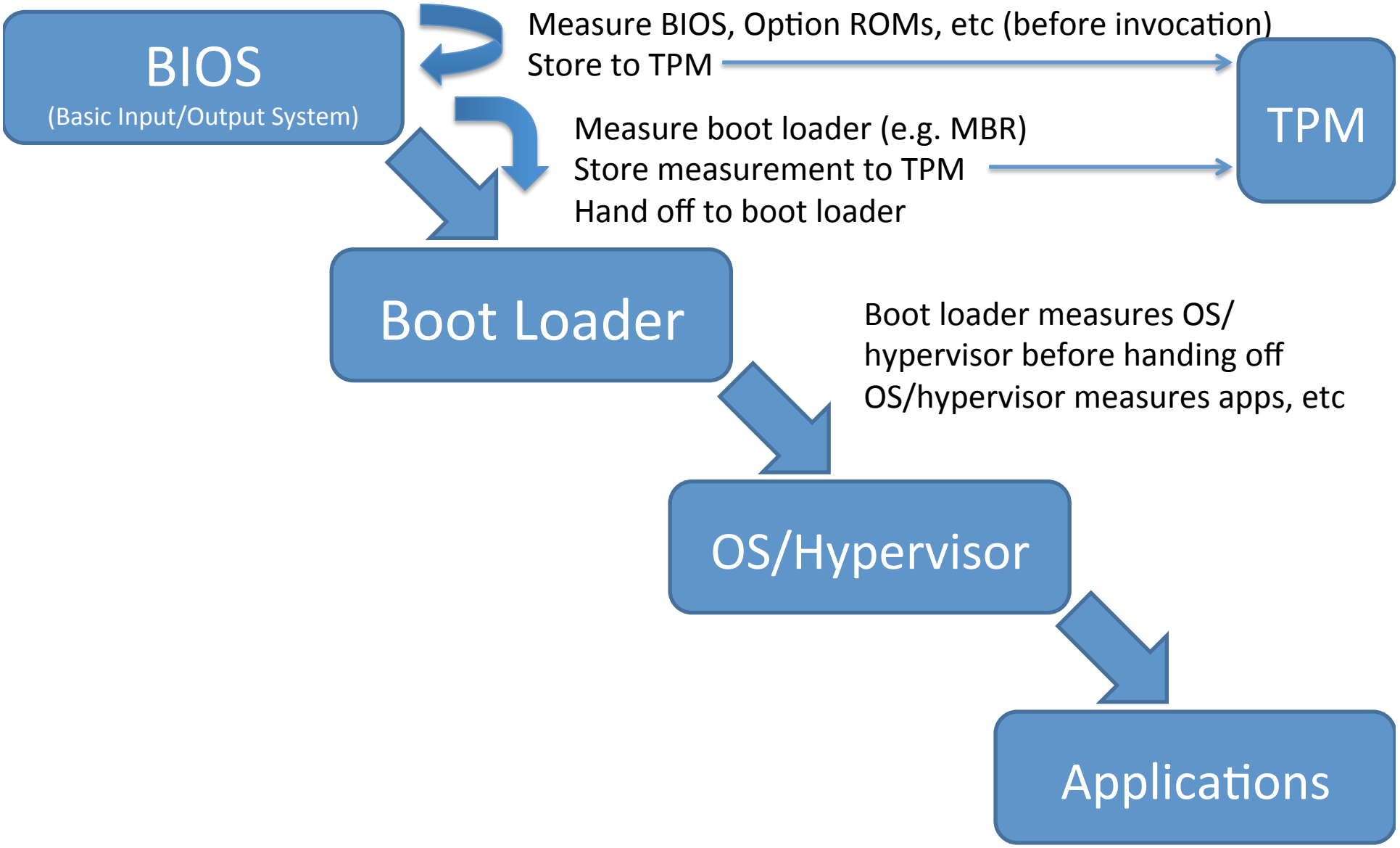
NOTE

- There's an entire 2 day class all about TPMs, and what they can and can't do, here:
- http://opensecuritytraining.info/IntroToTrustedComputing.html
- (Why John chose not to reuse some of that material, I don't know)

Motivation

- "Secure Boot" does some sort of check on the integrity of components (such as a digital signature check) while booting up. If the check works, it continues.
  - So you basically have a situation where it's either "
  - And as you saw, it can be bypassed
- "Measured Boot" *may* allow the system to still boot even if an integrity violation occurs, but it allows integrity evidence to be collected and stored into a trustworthy location like the TPM
  - Information can then be sent back to an "appraisal" server (in a process known as "remote attestation") for making the determination of whether a system is infected or not

How computers do measured boot



Trusted Platform Module* (TPM)

- A physical chip soldered to the motherboard
  - There are logical/software TPMs, but not relevant to this course
    - And *not* a good idea
- Passive chip. Programmed by applications (like the BIOS)
- Created by a committee of companies and organization collectively called the Trusted Computing Group (TCG)
- The goal of the TCG is to provide an architecture that implements Trusted Computing
- Trusted Computing means that your system will behave as expected or at least be able to provide reports indicating that it might not be

The TPM chip on the E6400

***This is only a basic primer on TPM; just enough to understand the BIOS relation to the trusted computing technologies which the TPM provides. Also, this is all based on the 1.2 Specifications, since 2.0 is not finalized, and therefore hardware using it is not common yet.**

TPM Functionality: Platform Integrity Reporting

- A TPM has 3 basic functions:
  1. Platform Integrity Reporting (aka: Root of Trust for Reporting)
  2. Platform Authentication
  3. Secure Storage
- Platform Integrity Reporting is actually the only one that is really applicable to this class
- Includes the measurement performed by the BIOS code (including UEFI)
- Also includes the integrity reporting feature of the TPM to provide a snapshot of the measurement state
- We'll cover this topic in a bit

TPM Functionality: Platform Authentication

- A TPM has 3 basic functions:
  1. Platform Integrity Reporting (aka: Root of Trust for Reporting)
  2. Platform Authentication
  3. Secure Storage
- Platform Authentication refers to creating Authentication Identity Keys (AIK)
  - Used to sign PCR quotes

TPM Functionality: Secure Storage

- A TPM has 3 basic functions:
  1. Platform Integrity Reporting (aka: Root of Trust for Reporting)
  2. Platform Authentication
  3. Secure Storage
- Secure Storage provides two functions:
  1. Binding – Encrypts data. Data can be encrypted with a migratable key so that it is bound to a specific TPM/platform or it can be encrypted with a migratable key so that the data can be migrated to another system. Caller provides the valid key to decrypt.
  2. Sealing – Encrypts data (keys, etc.) so that it will only be decrypted when the system PCRs are in a particular state. Sealed data must be encrypted with non-migratable keys so the data encrypted is bound to the platform/TPM.
- Microsoft BitLocker uses the TPM Secure Storage Sealing feature

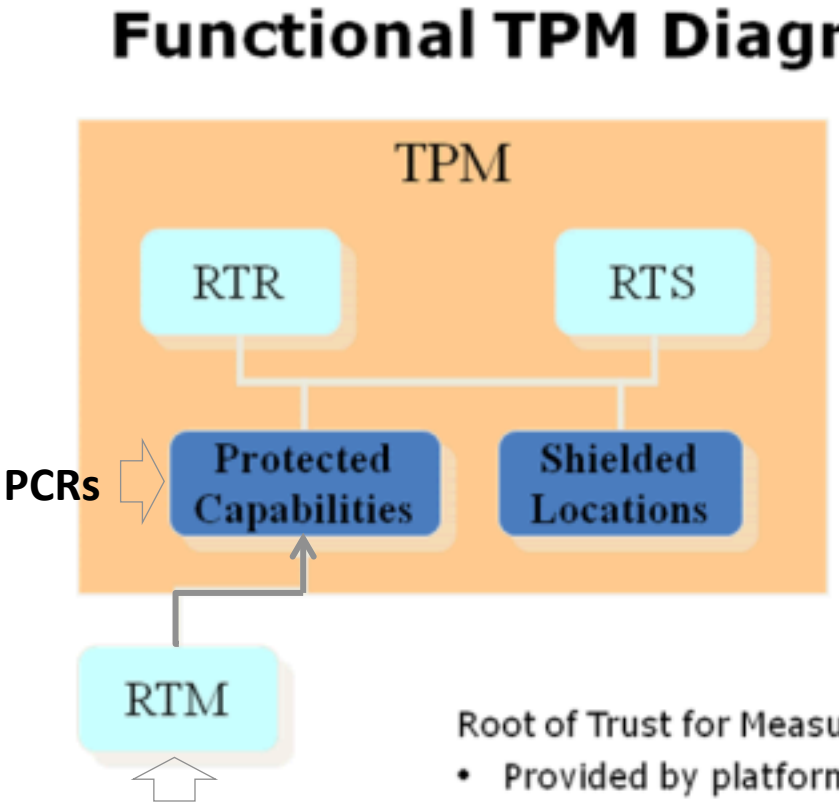
TPM Key Types

- Endorsement Key
  - Permanently embedded in the TPM hardware at the time of manufacture
  - The private part of the Endorsement Key is never released outside of the TPM
  - Can be used to verify that software is communicating with an actual TPM (as opposed to a malicious software application pretending to be a TPM)
- Storage Root Key
  - Created when the TPM is initialized by software
  - Used to encrypt/decrypt keys created by an application so that they can be stored outside the TPM
  - Embedded in the TPM hardware, can be overwritten if the TPM is cleared and re-initialized

TPM Key Types

- Migratable Keys
  - Can be migrated to another TPM/platform
- Non-Migratable Keys
  - Stored within the TPM shielded storage
  - Cannot be migrated to another platform/TPM
- Attestation Identity Keys (AIK)
  - Non-migratable keys
  - Used to sign "quotes" of PCR values when requested by an application
    - Therefore the main key we often care about for “remote attestation”

TPM Components



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

Platform Integrity Reporting

- This functionality combines what is called (in TPM-land) the Root of Trust for Reporting (RTR) and the Root of Trust for Measurement (RTM)
- Per TCG: "The RTM is a computing engine capable of making inherently reliable integrity measurements.*"
- The code that performs the measurements are implemented outside the TPM (as shown in the previous slide)
  - By the BIOS, for example.
- There are two types of RTMs, Dynamic and Static.
- Dynamic means that trust is established after the operating system has booted. Trust is established even when the system booted in an insecure state
- Intel's Trusted Execution Technology (TXT) uses DRTM
  - TXT is an entire course unto itself which Xeno is preparing
  - www.invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf

*ISO/IEC 11889-1 Information Technology Trusted Platform Module, Pt.1

Static Root of Trust for Measurement* (SRTM)

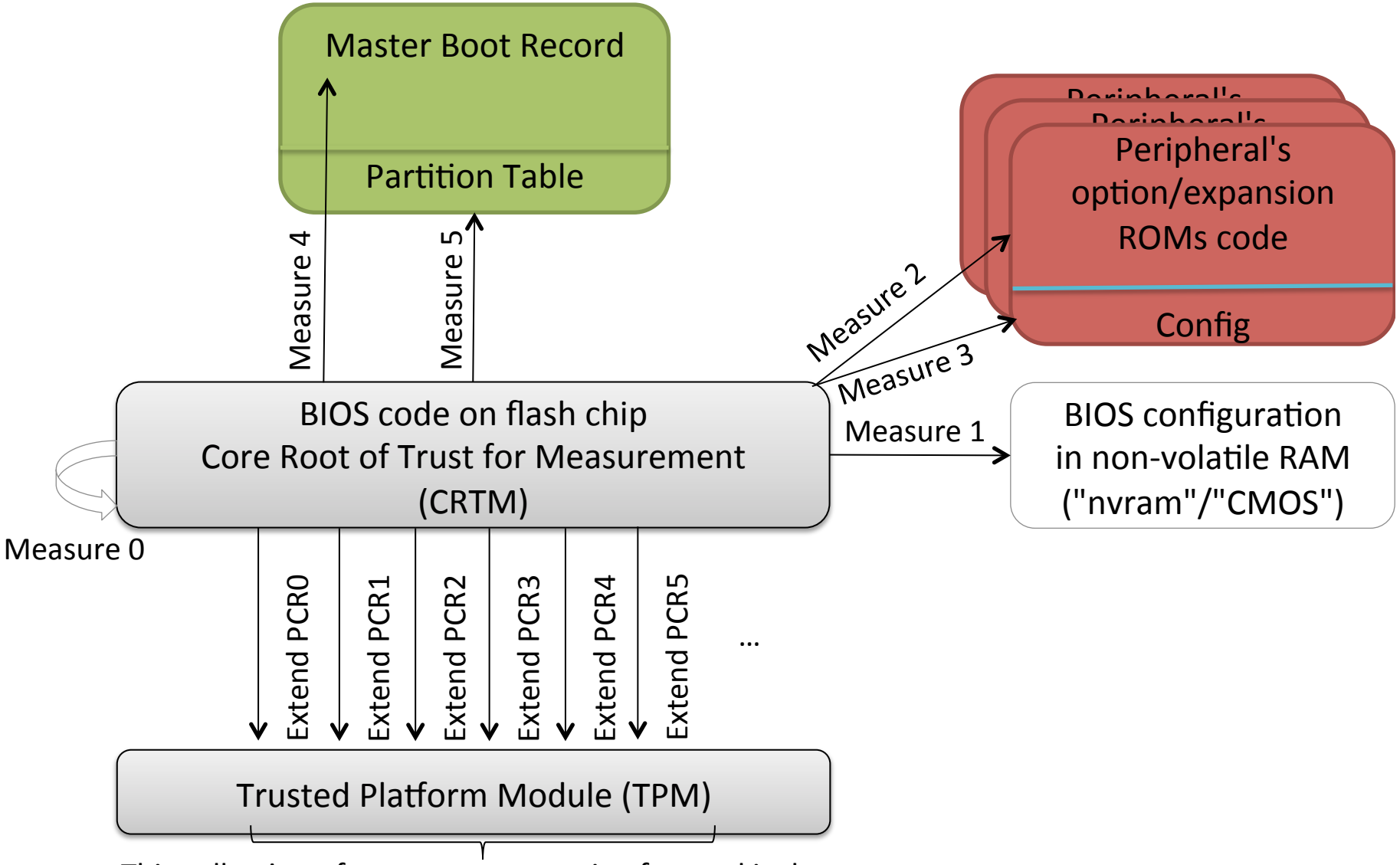
- Also called Measured Boot (Not to be confused with Secure Boot, that's a different entity discussed in the UEFI portion)
- General idea is that the next component of the boot sequence is measured before control is handed off to it
- Thus forms a "chain of trust" where each component has been measured before it executes
- "Static" refers to the idea that the same components are measured each time and that their measured values should not change
- Begins life in the BIOS so its implementation is thus the responsibility of the vendor
- The first of these measurements is called the Core Root of Trust for Measurement (CRTM)

*Often referred to as S-CRTM, Static-Core RTM

Core Root of Trust for Measurement (CRTM)

- Whereas it's said the SRTM forms a "chain of trust", the CRTM forms the "anchor"
- CRTM is responsible for measuring the next component in the boot sequence (next link in the chain)
- Being part of the overall SRTM, it always begins life in the BIOS
- As a guideline, CRTM should perform its measurements as soon as possible (start establishing trust sooner than later)
- According to the TCG, the "TPM and CRTM are the only trusted components on the Motherboard" (TCG PC Client Specification for Conventional BIOS)

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



CRTM (im)Mutability

- "The Core Root of Trust for Measurement (CRTM) MUST be an immutable portion of the Host Platform's initialization code that executes upon a Host Platform Reset"
- "immutable means that in order to maintain trust in the Host Platform, the replacement or modification of code or data MUST be performed by a Host Platform manufacturer-approved agent and method."*
- Basically they are telling vendors that they know the CRTM will be implemented on mutable flash hardware, but that they will be in compliance as long as its only their code that ever changes it.
- That works great until it doesn't…

*TCG PC Client Implementation for BIOS

Platform Configuration Registers (PCRs)

- The measurements of each component are stored on the TPM in registers
- There are at least 16 PCRs on a TPM, each 20 bytes long
- Initialized to 0 each time the platform is reset
- Can only be modified by an extend function
- PCR[n] = SHA-1 (PCR[n] || measured data)
  - where || denotes concatenation
- So basically, each PCR represents the state of one or more boot components (at the time of measurement)
- Each boot component is represented as a SHA-1 hash

PCR Standard Usage

PCR	Use
0	S-CRTM, BIOS, Host Platform Extensions, and Embedded Option ROMs
1	Host Platform (Motherboard) Configuration
2	Option ROM code
3	Option ROM Configuration and Data
4	IPL Code (usually the MBR) and Boot Attempts
5	IPL Code Configuration and Data
6	Power State Transition (sleep, hibernate, etc.)
7	Defined by OEM
8-15	Unassigned

- Each PCR is intended to store a different measured component, defined by TCG
- The implementation is actually up to the vendor

IPL = Initial Program Loader, typically the Master Boot Record (MBR)

General Problems with PCR Hashes

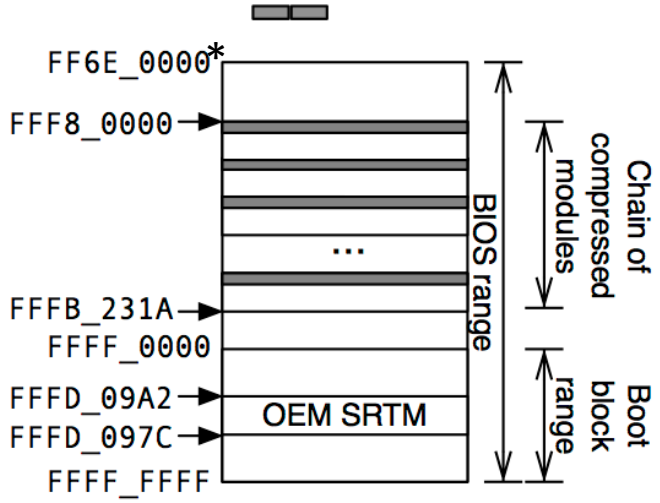
- Opaqueness
  - Generally no golden set of PCRs is provided by the OEM.
    - Some vendors like HP have started to finally provide this! Yay!
  - No description of what is *actually* being measured and incorporated into the PCR values.[1]
  - "Homogeneous" systems can have different PCR values.[2]
  - 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



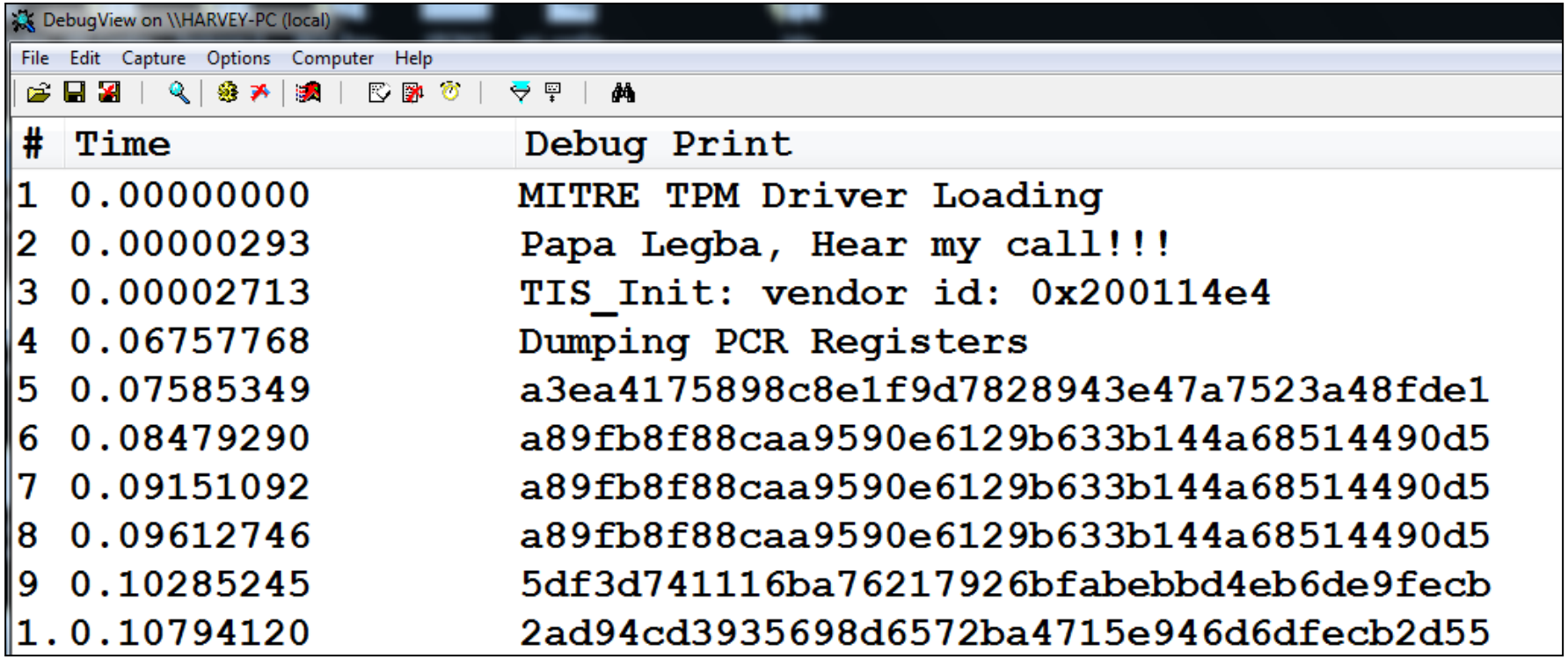
- PCR0 should contain a measurement of the CRTM and other parts of the BIOS.
- In the above diagram, the dark areas represent what the E6400 actually incorporates into the PCR0 measurement.
- Only 0xA90 of the total 0x1A0000 bytes (.15%) in the BIOS range are incorporated, including:
  - The first 64 bytes of the 42 compressed modules.
  - Two 8 byte slices at 0xDF4513C0 and 0xDF4513C7.
  - The CRTM is not incorporated at all.

*Typo in image: BIOS Base on the E6400 is located at FFE6_0000h

Implications of the weak SRTM

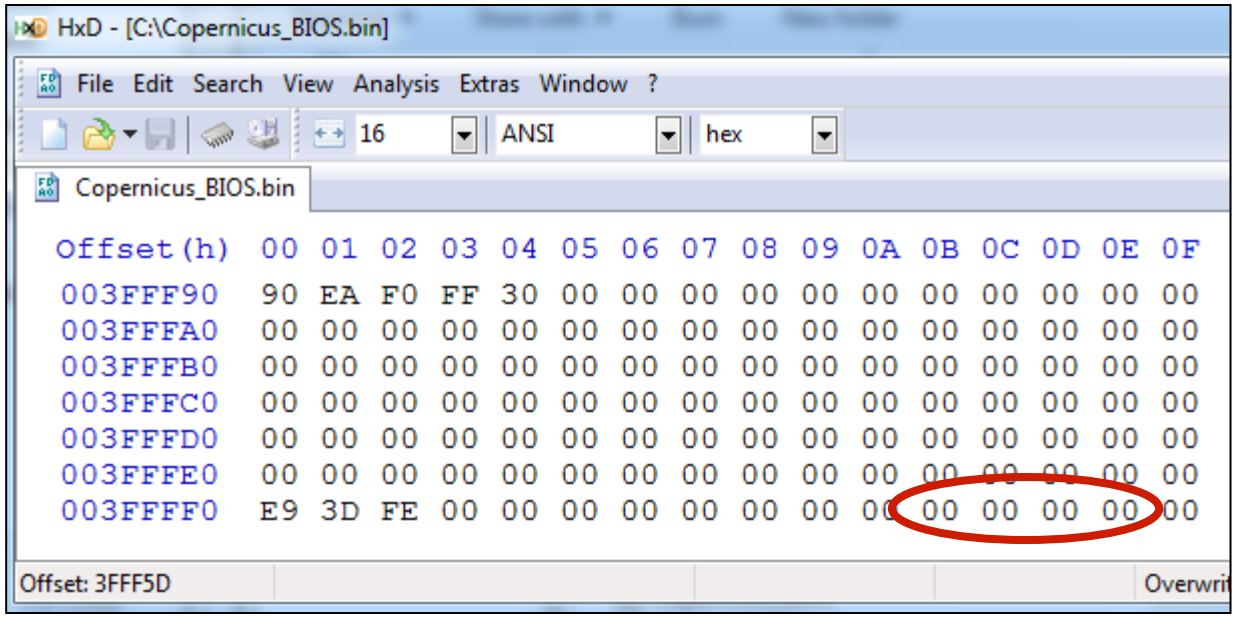
- Measurements for things like PCI option ROMs and BIOS configuration are not actually captured.
- We can modify the *majority* of the E6400 BIOS without changing any of the PCR values.
  - Yuriy Bulygin presented a similar discovery at CanSecWest 2013 regarding his ASUS P8P67
    - "Evil Maid Just Got Angrier: Why Full-Disk Encryption With TPM is Insecure on Many Systems" – Yuriy Bulygin – March 2013 http://cansecwest.com/slides/2013/Evil%20Maid%20Just%20Got%20Angrier.pdf
- As long as the Flash can be modified, the measurement code which executes from the flash can be modified to report false negatives
- Let's take a look at some weaknesses that come along with a S-CRTM that provides incomplete coverage

Reading PCRs with OpenTPM



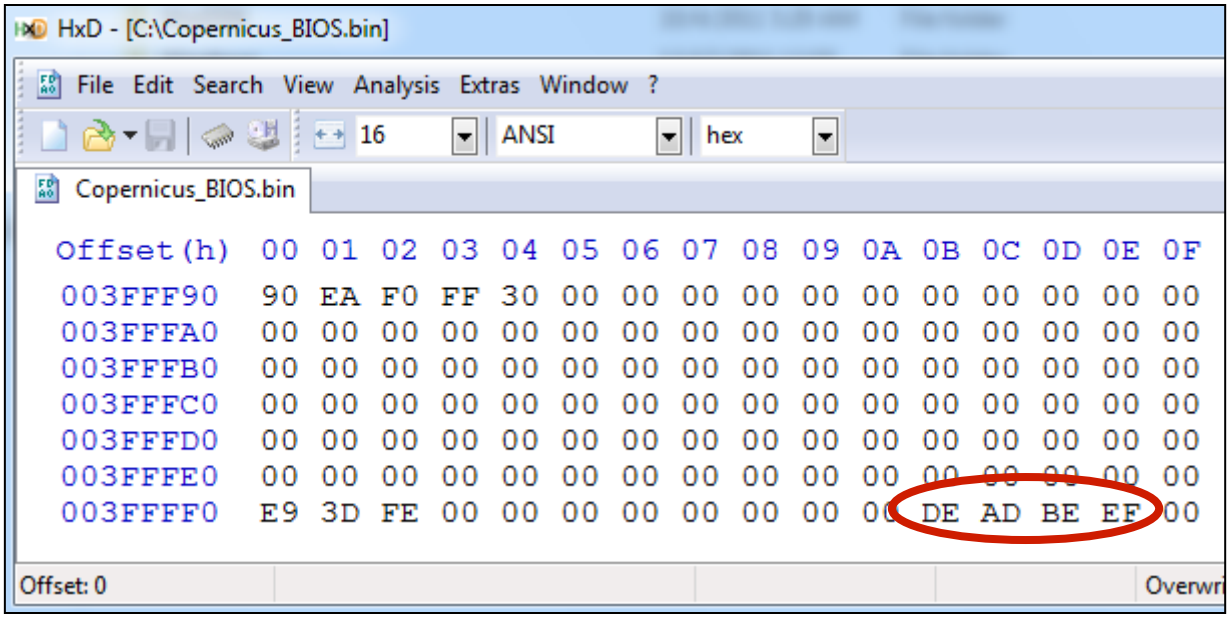
- Corey Kallenberg wrote OpenTPM which queries and dumps the PCR register set
- Open source: https://code.google.com/p/opentpm/
- Activate/enable your TPM in your BIOS settings to use it

vulnBIOS Example: Incomplete S-CRTM Coverage



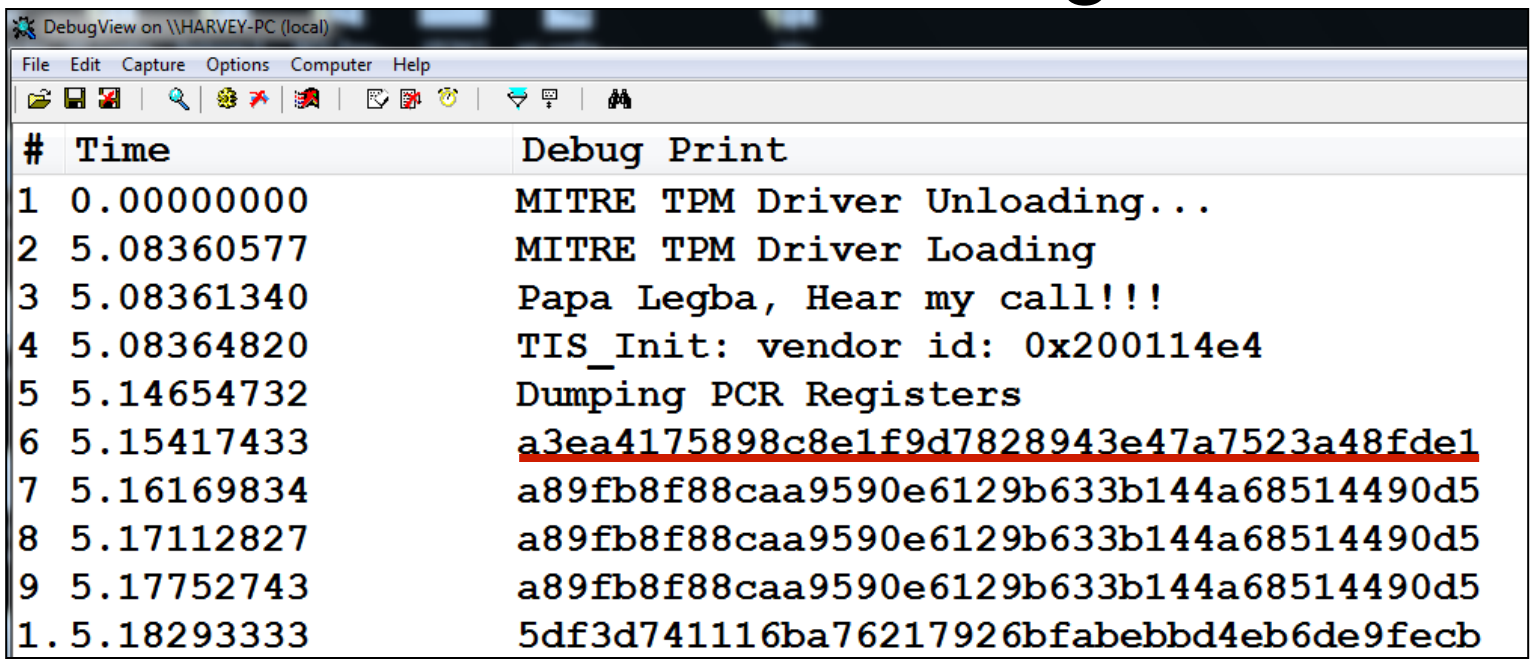
- Either view your existing BIOS dump or make a new one using Copernicus
- Open in HxD and skip to the end (entry vector)
- Notice bytes 3F_FFFB – 3F_FFFE are 0h

vulnBIOS Example: Incomplete S-CRTM Coverage



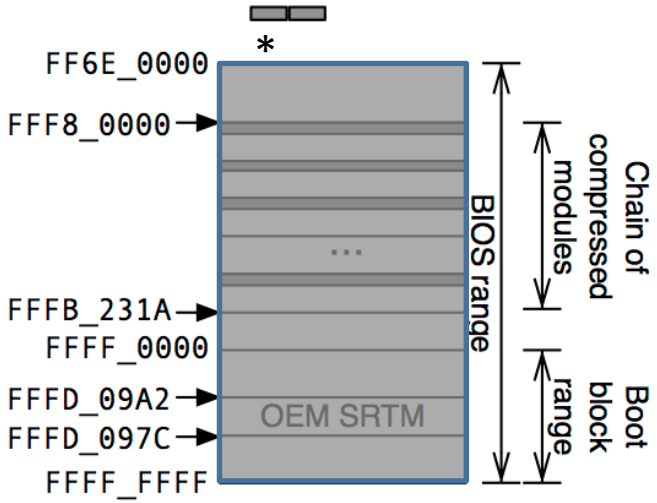
- This will only run on E6400 systems!
- Execute the ‘tpm_spi_write_no_change_pcr.sys’ driver
- This writes DEADBEEFh as you can see (in this case, on this system, this does not prevent the system from booting, YMMV!!!)
- Reboot

vulnBIOS Example: Incomplete S-CRTM Coverage



- After rebooting the system,
  - Required because the BIOS has to re-run the measured boot process and re-populate the PCRs
- Re-run the OpenTPM driver
- Notice in particular PCR0 is the same (they are all the same)
- So an attacker can modify big chunks of the BIOS without triggering a change to PCR0!
- But wait – it gets worse!

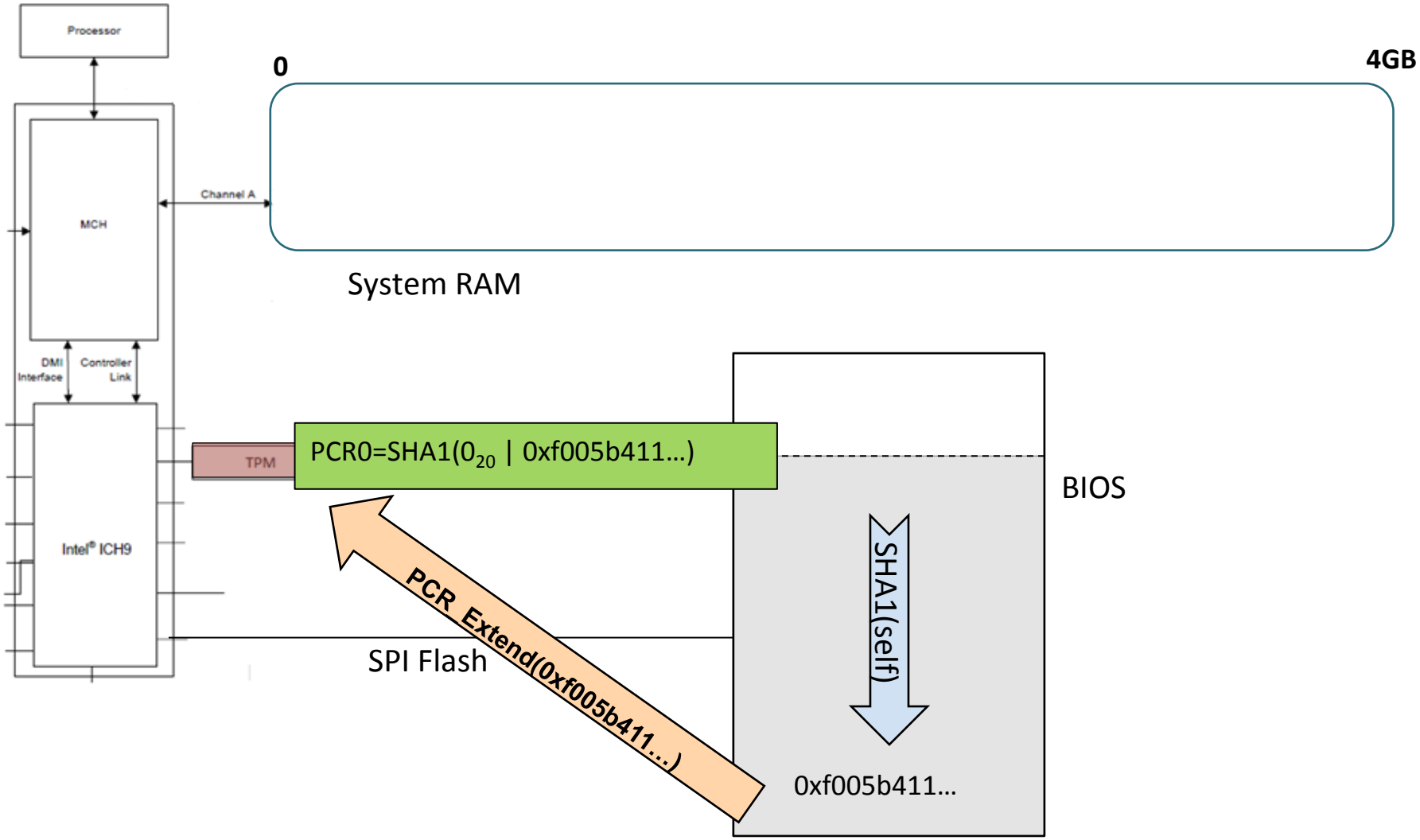
The Real Weakness: Mutable CRTM



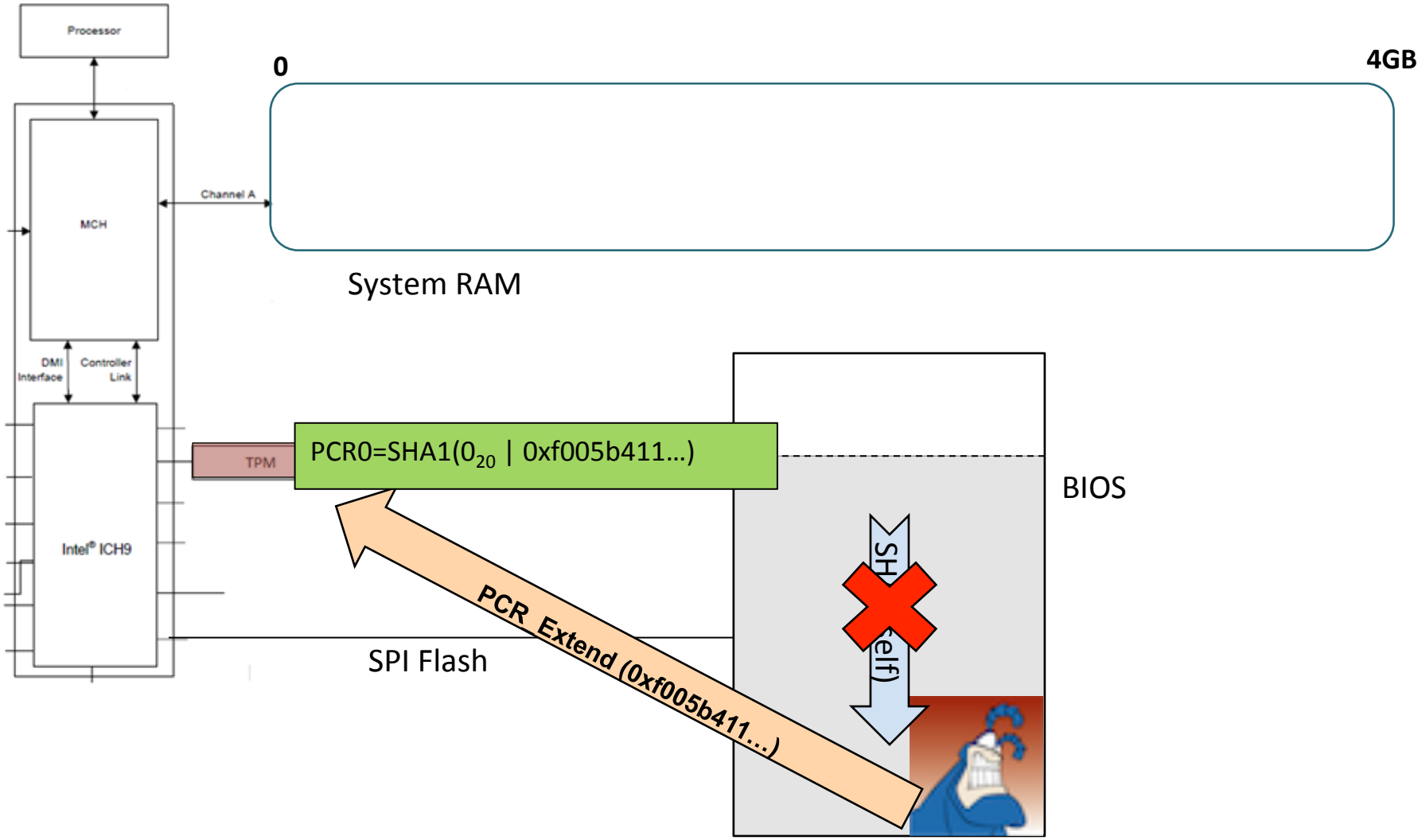
Still weak; assuming the attacker can get into SMRAM or overwrite the flash either by exploiting a code vulnerability or misconfigured system

- Don't let the sparse measurement coverage in the previous slide distract you from the real issue – it is a red herring!
- What *really* makes the S-CRTM weak is the fact that the CRTM is implemented on mutable flash hardware (the BIOS)
- As we've seen, it can be trivial to overwrite the BIOS flash
- An attacker who identifies the part of the BIOS that performs the CRTM measurement can simply overwrite it and therefore control it
- It doesn't matter even if the ENTIRE BIOS is being measured
  - The attacker may just have to work a little harder

Normal BIOS PCR0 Measurement



PCR0 Measurement with a Tick



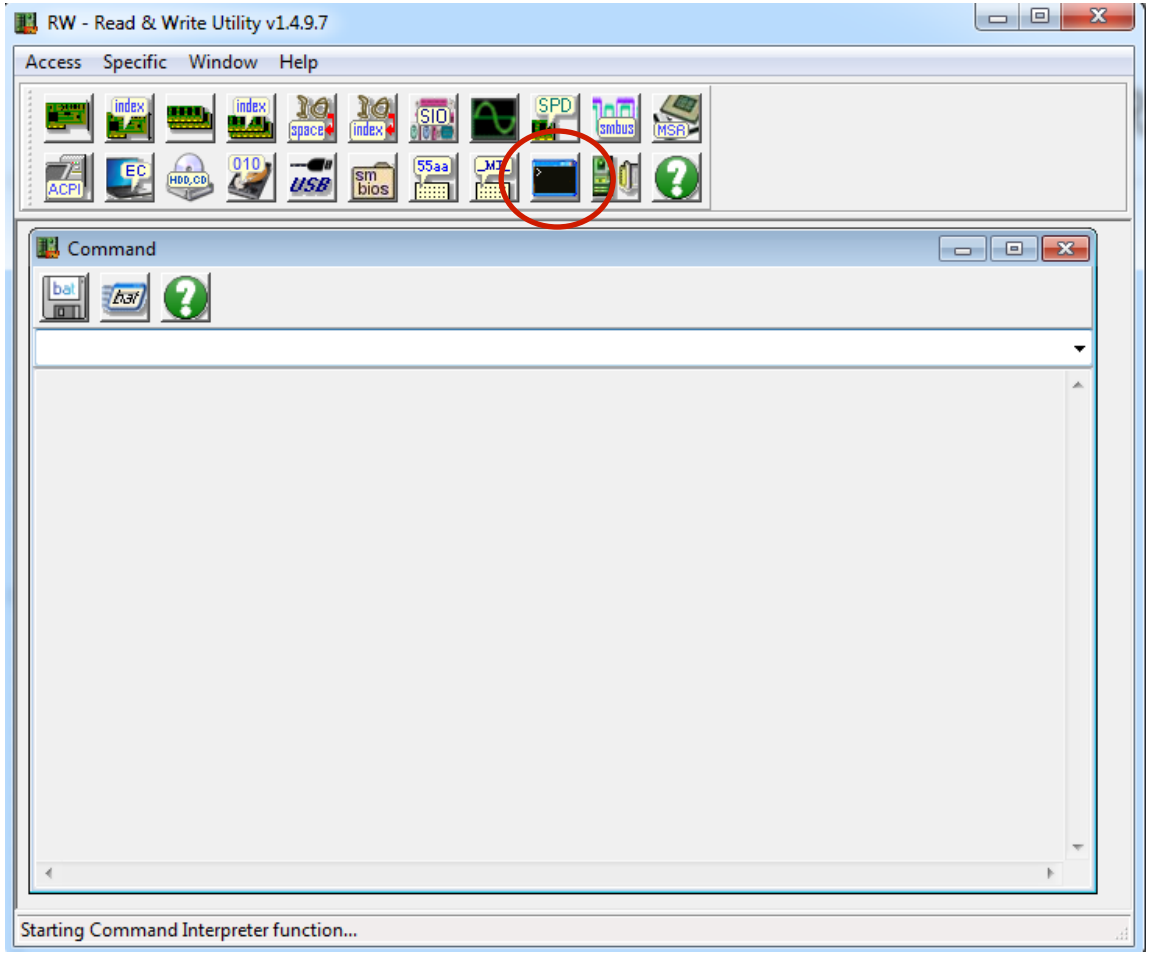
Mutable S-CRTM Problem

- But actually, if the attacker can write to the BIOS, they can modify any/all of the BIOS regardless of whether its being measured and simply forge the PCR values
- http://www.youtube.com/watch?v=S0lRcm3jvFo

Quick Diversion: RW Everything Scripting

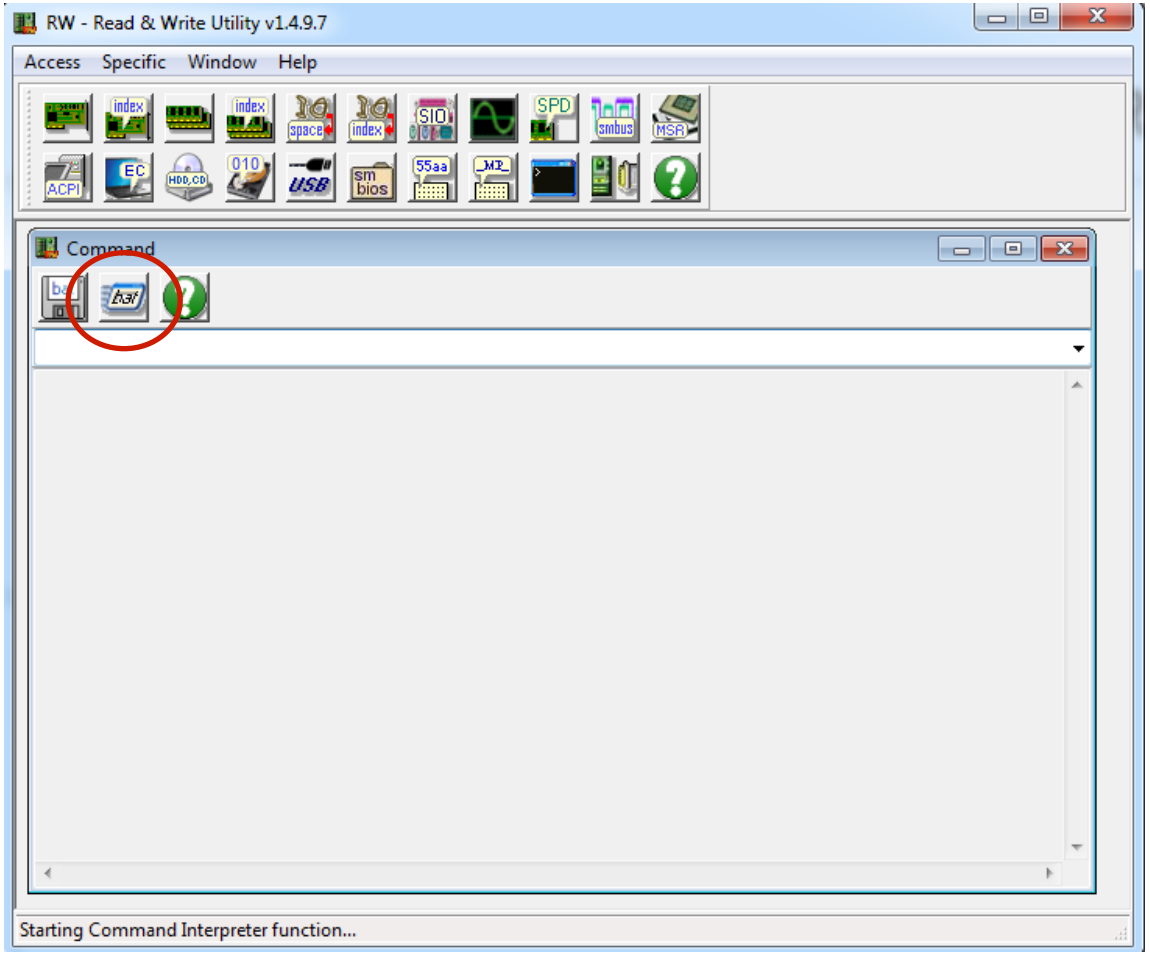
- RW Everything is a good tool for gathering all kinds of information about the system
- It has a scripting interface and a good command set so it's good for prototyping commands that touch bare-metal without writing a kernel driver (assuming windows, if Linux then just IOPL your way to Ring0 bliss)
  - Just be wary of syntax and expect a kernel crash now and then if you make a mistake
  - I've found that 64-bit Windows is more sensitive to RWE-related crashing
- This lab serves to show you some of RWE's scripting capabilities and how to use it for quick testing of ideas

Use RW-E Scripting to Read a PCR



- Open RW Everything
- Click on the CMD icon

Use RW-E Scripting to Read a PCR



- On your Desktop find the file named “ReadPCR0.rw”
- The file contents are on the next slide in case you have to enter it by hand or don’t want to do the lab but just see what features RW Everything offers

RW Everything Commands

```
# This is the PCR Command Blob
>W 0xfed40000 0x20
>W 0xfed40000 0x02
>W 0xfed40018 0x40
>W 0xfed40024 0x00
>W 0xfed40024 0xc1
>W 0xfed40024 0x00
>W 0xfed40024 0x00
>W 0xfed40024 0x00
>W 0xfed40024 0x0e
>W 0xfed40024 0x00
>W 0xfed40024 0x00
>W 0xfed40024 0x00
>W 0xfed40024 0x15
>W 0xfed40024 0x00
>W 0xfed40024 0x00
>W 0xfed40024 0x00
# THIS IS THE # PCR YOU WANT TO READ (0-16 OR WHATEVER)
>W 0xfed40024 0x00
>W 0xfed40018 0x20
# ok now read the PCR:
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
>R 0xfed40024
```

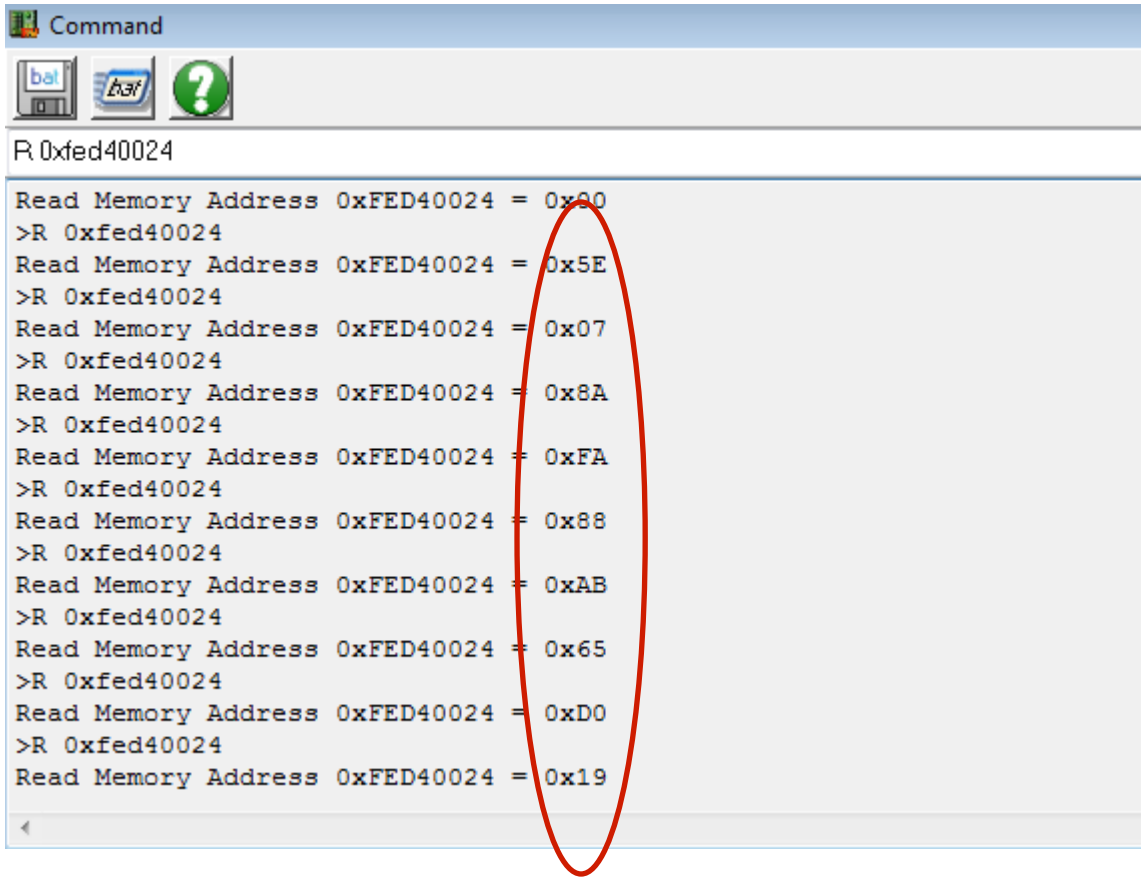
MMIO to the memory address range reserved for TPMs

Change this one to read a different PCR (0 - N)

Note how we just keep reading the same data location and will keep getting back different data

- Commands in ReadPCR0 are on the left
- Be wary of syntax!
- W 0xFED4000 0x20
  - Writes a Byte to 0xFED4000 (physical address)
  - W16 will write a Word
  - W32 will write a DWord
- R 0xFED40024
  - Reads a Byte from physical address 0xFED40024
- Yes it's ugly but can still be leveraged to save you time to test a PoC

Use RW Everything to Read a PCR



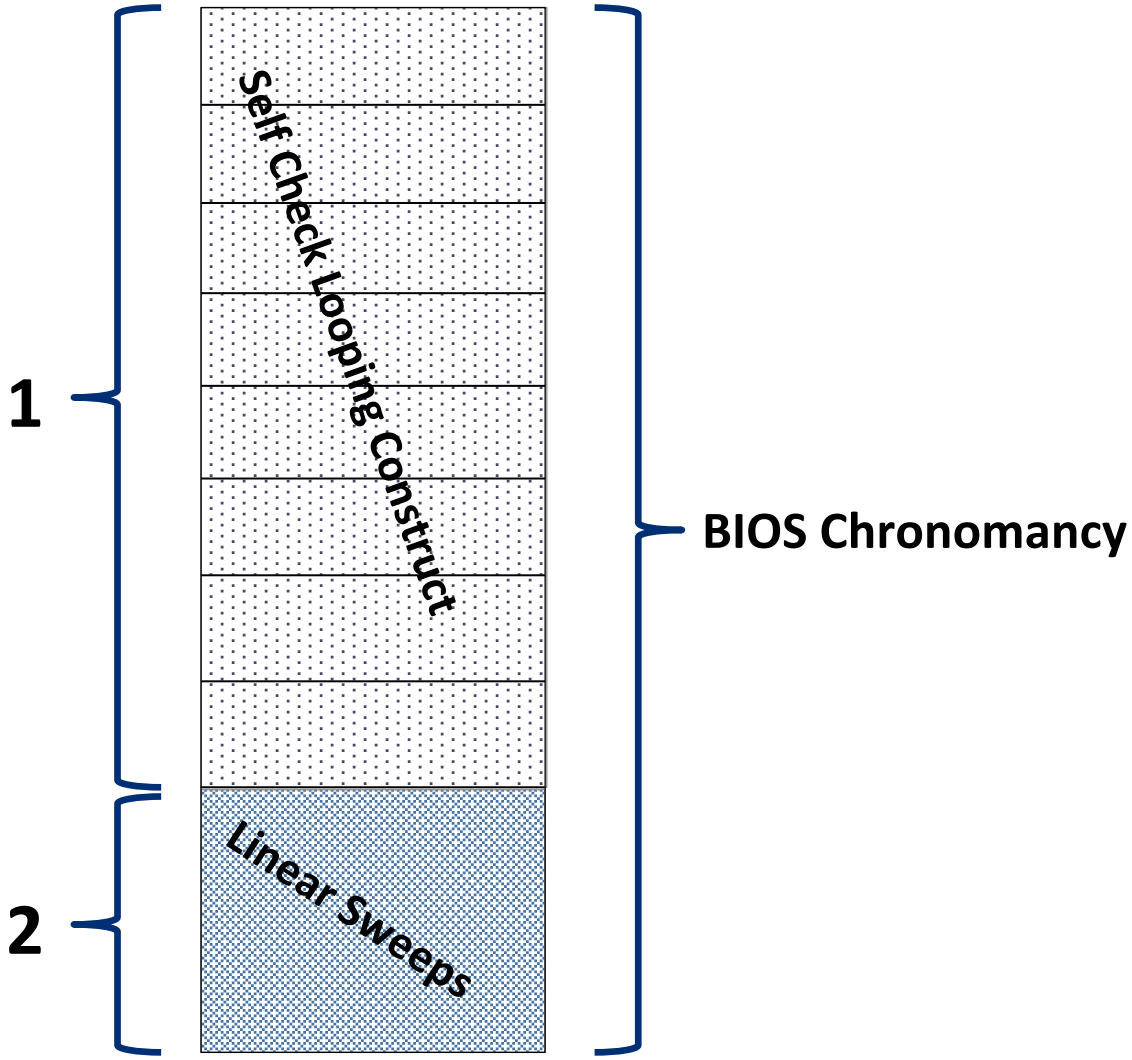
- If OpenTPM works on your system, then this will too
- Output is a bit verbose, but the PCR value will be in there after the header
- I couldn't expand the output window to capture the whole PCR for the screenshot
- But still good for prototyping, testing, probing

Trusted Computing Research: Timing-Based Attestation (TBA)

"Build your software so that if its code is modified, it runs slower."

- CMU has done a lot of research in this area (Seshadri, et al) and we applied it to the protection of Windows kernel memory & the BIOS
- Uses a timing side-channel to provide constant runtimes in absence of an attacker
- For the BIOS, it's meant to replace the CRTM only, not the entire SRTM
- Presenting this briefly just to provide an example of one way to protect a mutable codebase (e.g. embedded systems, HD firmware, NIC firmware, phone bootloaders, etc)
- Could be executed immediately upon system boot

Two Components of "BIOS Chronomancy"

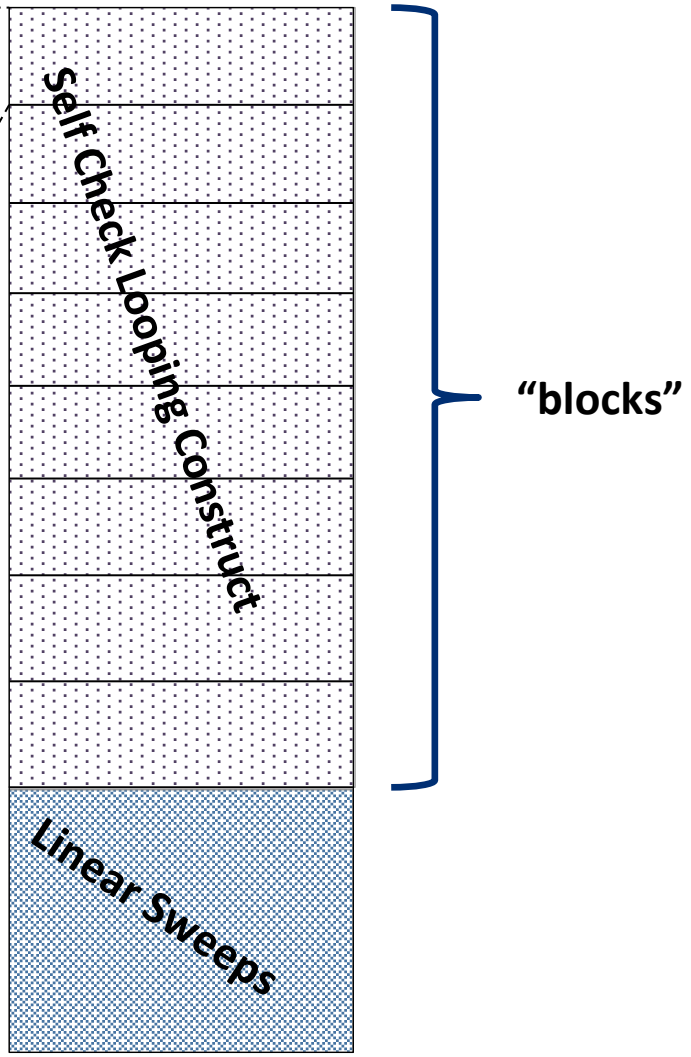


Self-Check Requirements

- Reads its own data
  - Incorporated into checksum so if it changes the checksum changes
- Reads its 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 Self-Check Component

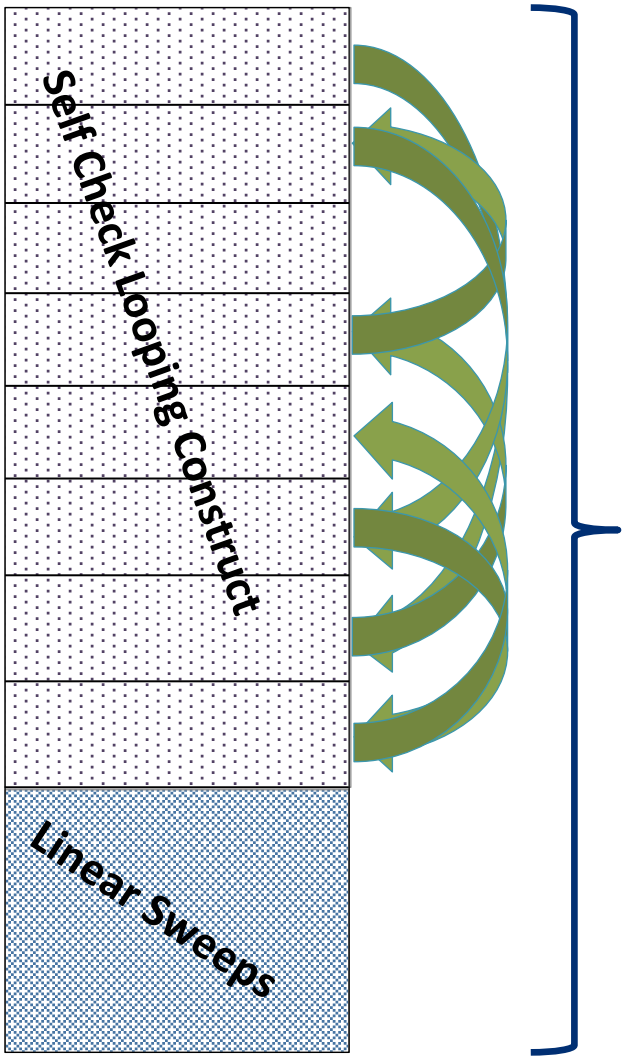
```
checksum[0] += nonce;
checksum[1] ^= DP;
checksum[2] += *DP;
checksum[4] ^= EIP;
mix(checksum);
nonce += (nonce*nonce) | 5;
DP = codeStart + (nonce % codeSize);
iteration++;
```



- **Each block differs from the others so attacker will have to forge every block**

Self-Check "Pseudo-Random Walk"

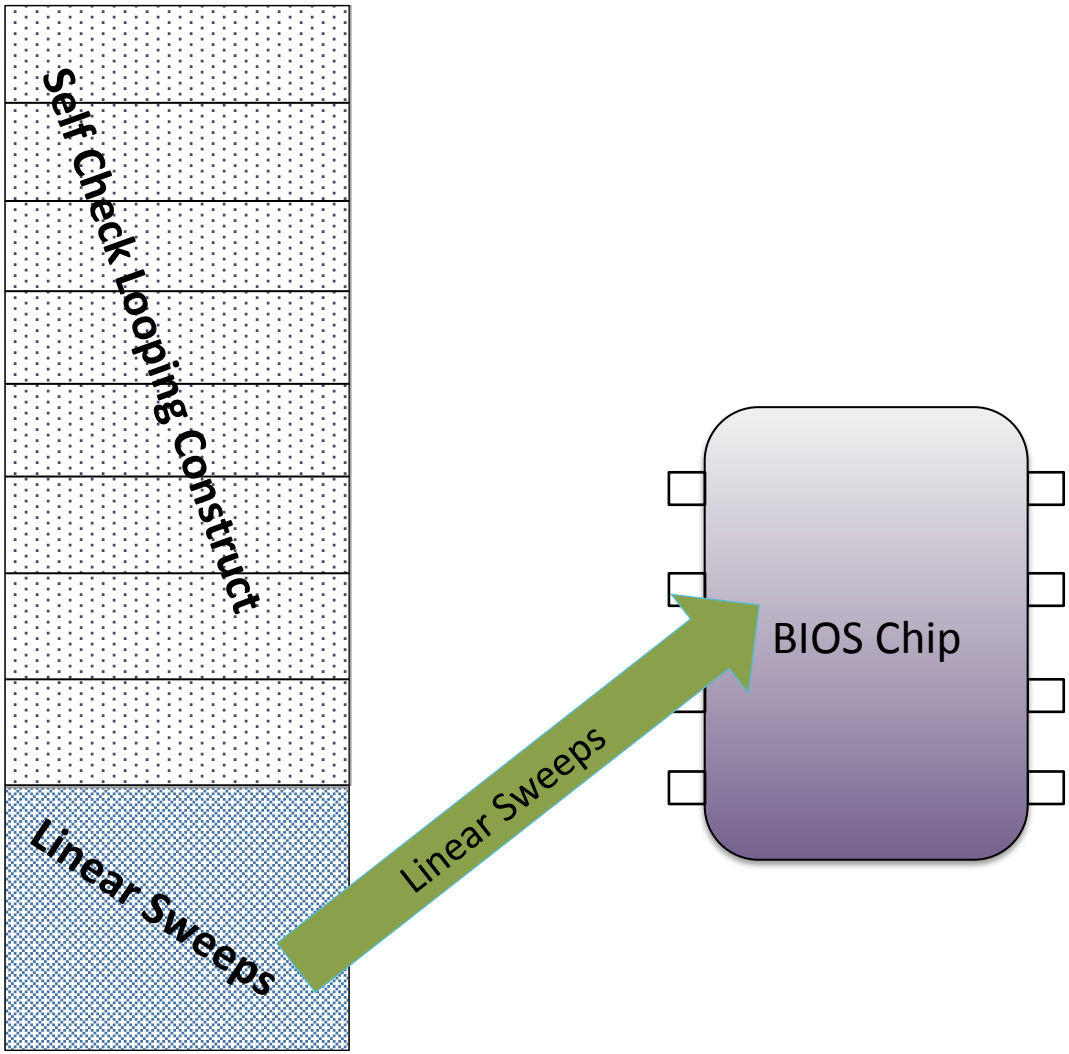
- **Pseudo-Random based on Tick Session Nonce obtained from TPM**
- **Iterates through the blocks a million times or so**



<u>Entire</u> BIOS Chronomancy range is measured by the self-check, including the portion that performs the linear sweep measurements

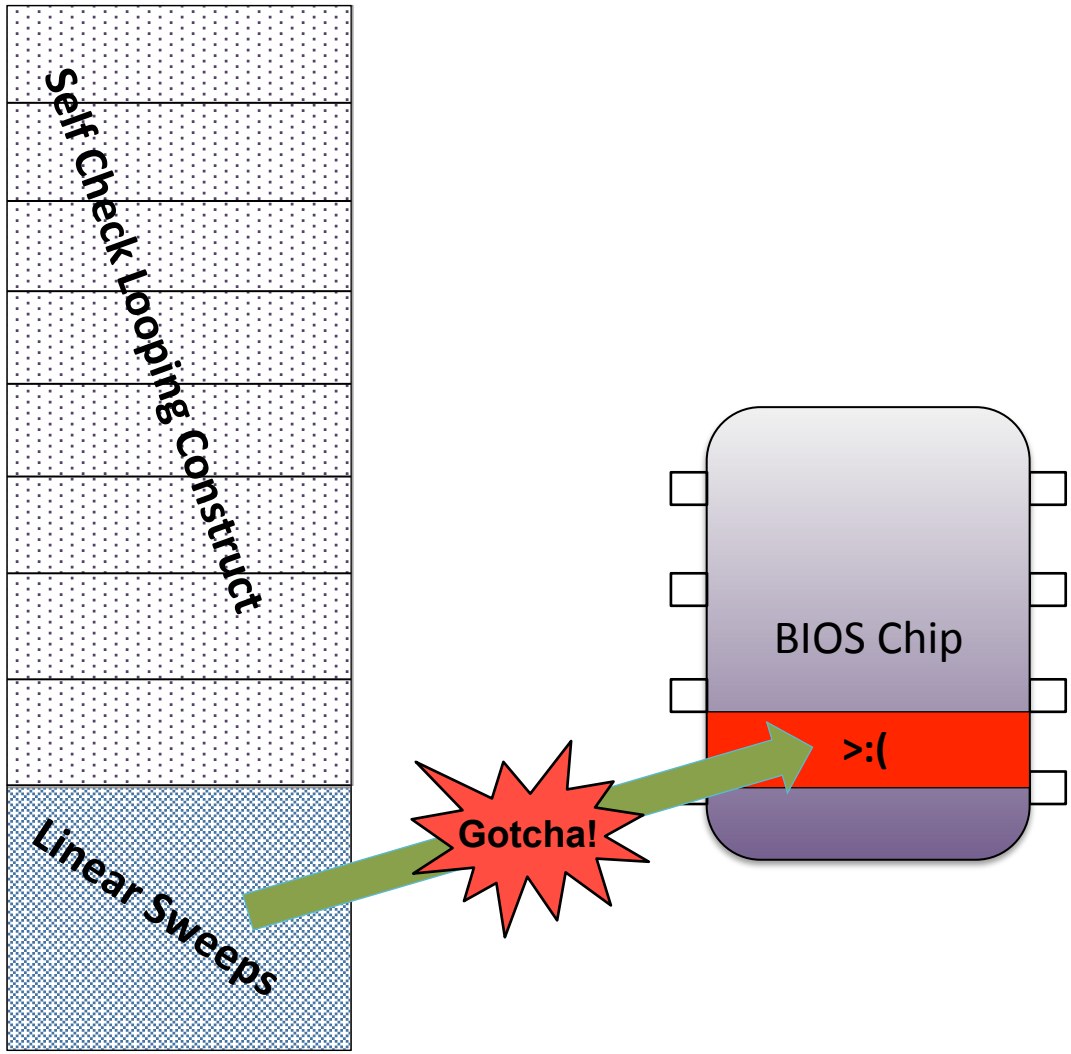
Linear Sweeps

- **Measures BIOS, Option ROMs, SMRAM, IVT, and anything else you want.**



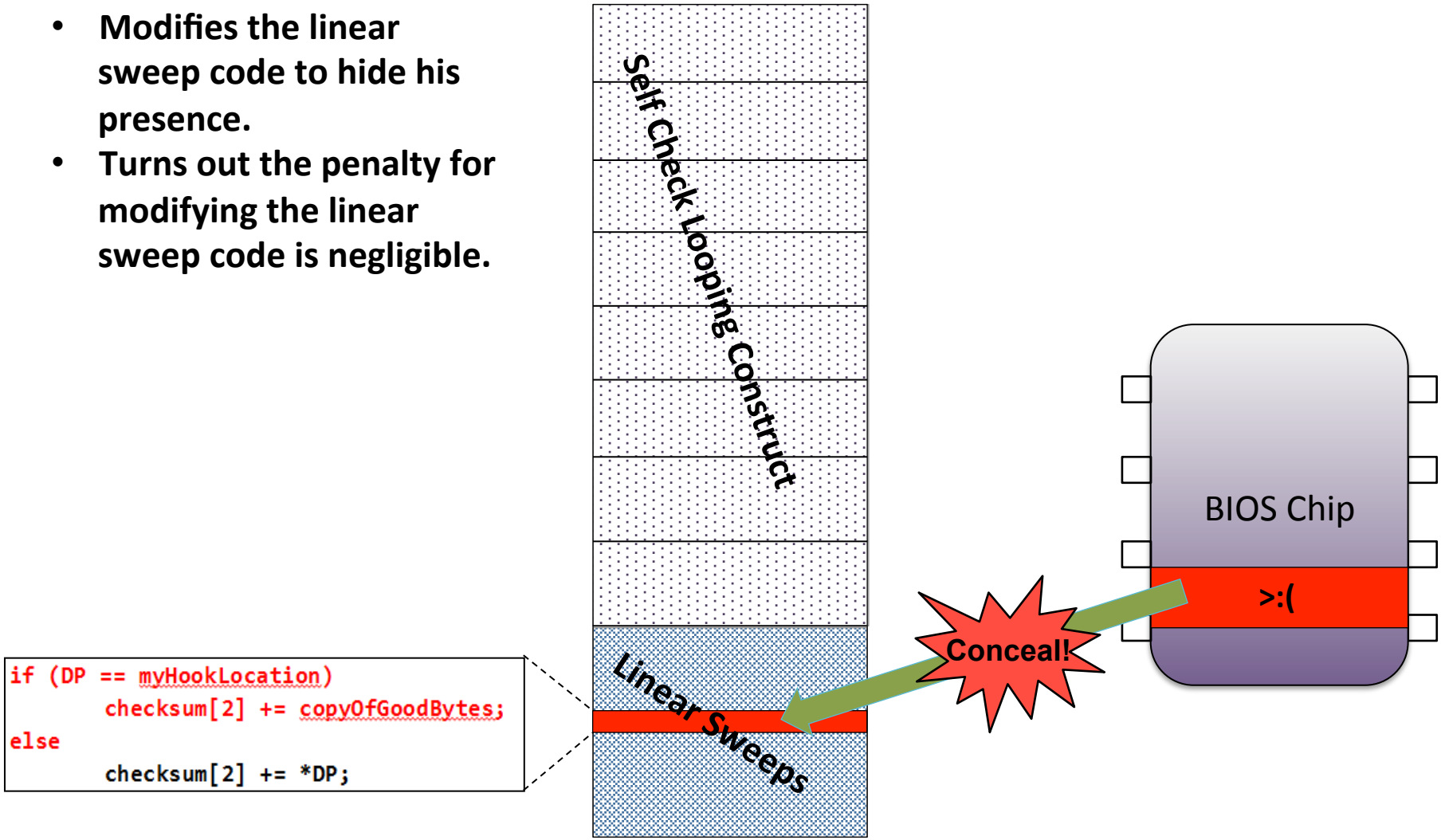
Attackers Dilemma: 1 of 3

- **Attacker wants to implement a rootkit and of course wants to hide its presence.**
- **Attacker is aware of BIOS Chronomancy and understands how to works.**
- **Attacker knows if he does nothing the linear sweeps will detect his presence in the BIOS.**
- **The timing measurement will be okay, but the calculated checksum will differ from the expected.**



Attackers Dilemma: 2 of 3

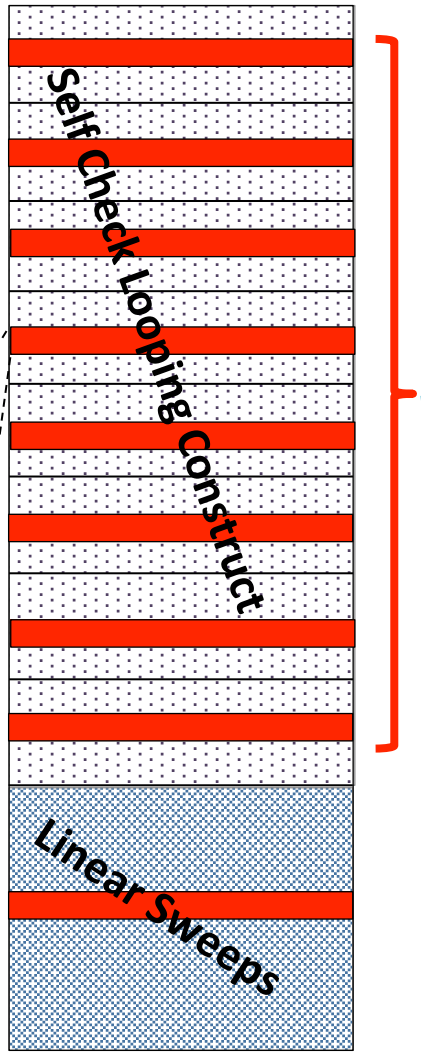
- **Modifies the linear sweep code to hide his presence.**
- **Turns out the penalty for modifying the linear sweep code is negligible.**



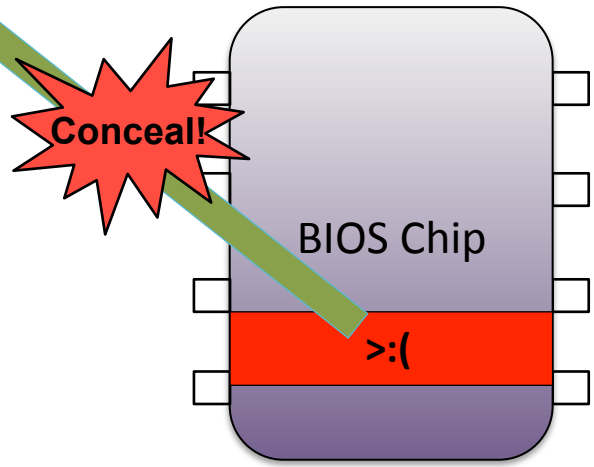
Attackers Dilemma: 3 of 3

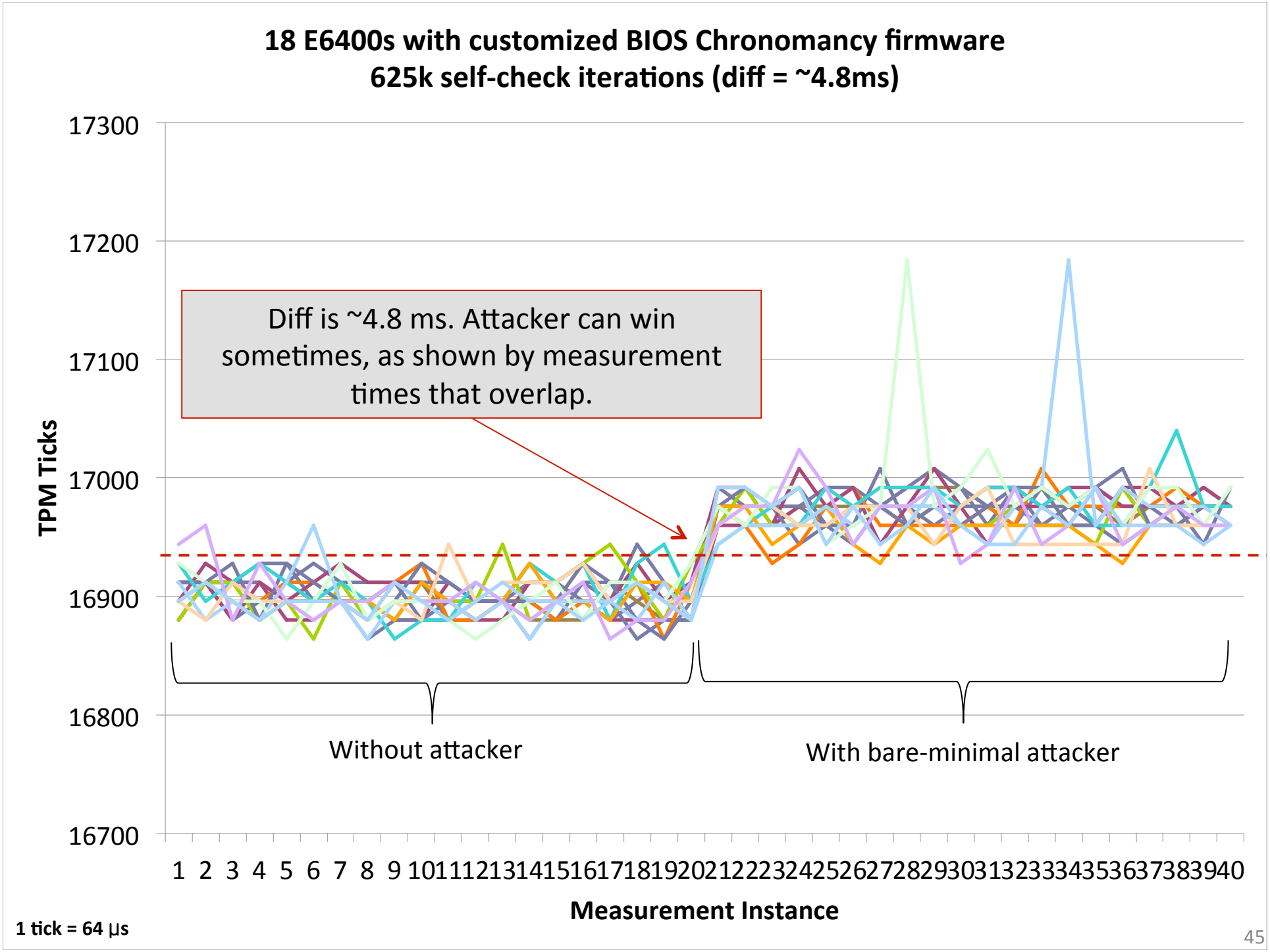
- However, now the attacker must also hide the changes he made to the linear sweep code from the self-check measurement.

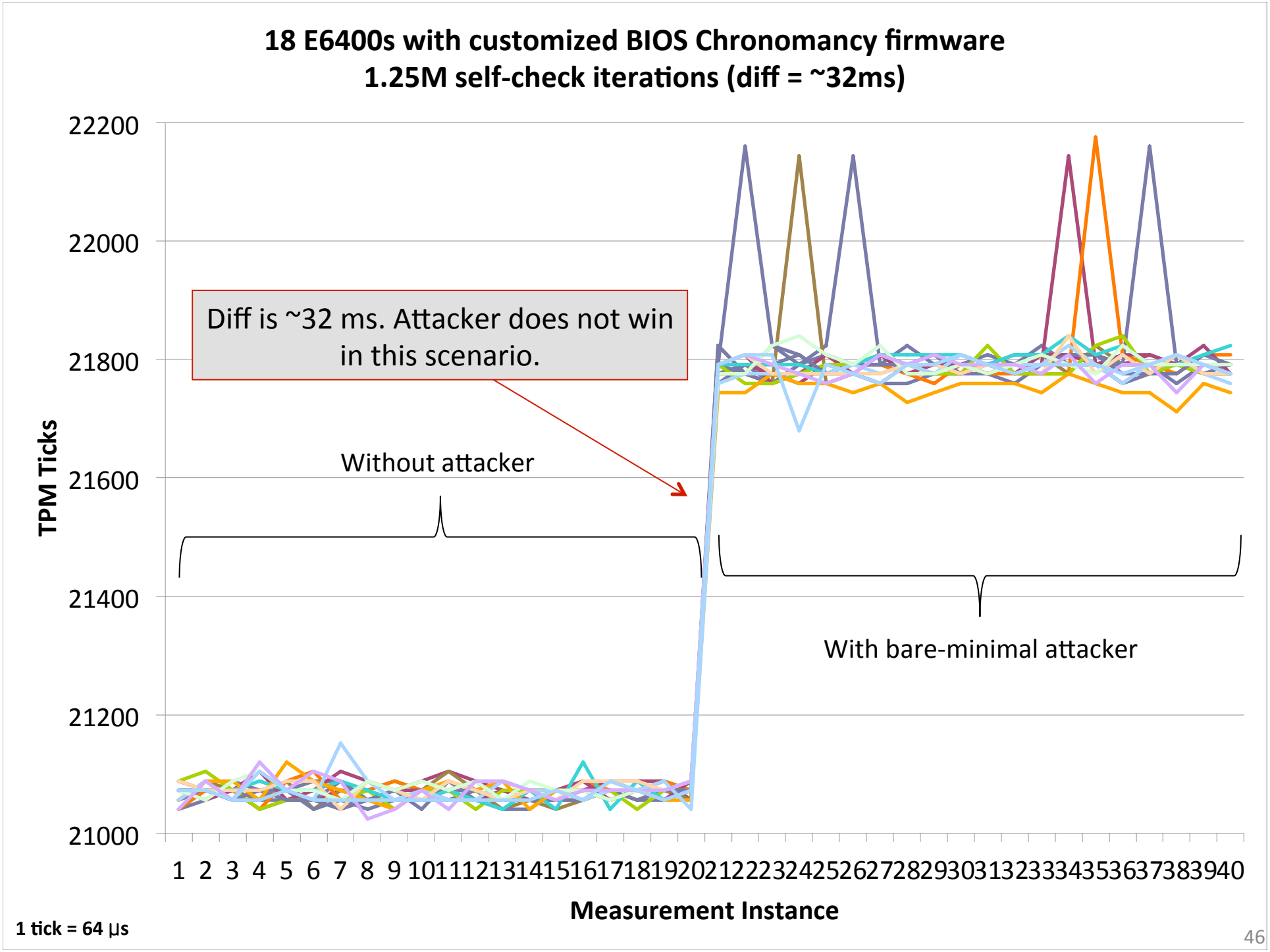
```
checksum[0] += nonce;
checksum[1] ^= DP;
if (DP == myLinearSweepMod)
        checksum[2] += copyOfGoodBytes;
else
        checksum[2] += *DP;
checksum[2] += *DP;
checksum[4] ^= EIP;
mix(checksum);
nonce += (nonce*nonce) | 5;
DP = codeStart + (nonce % codeSize);
iteration++;
```

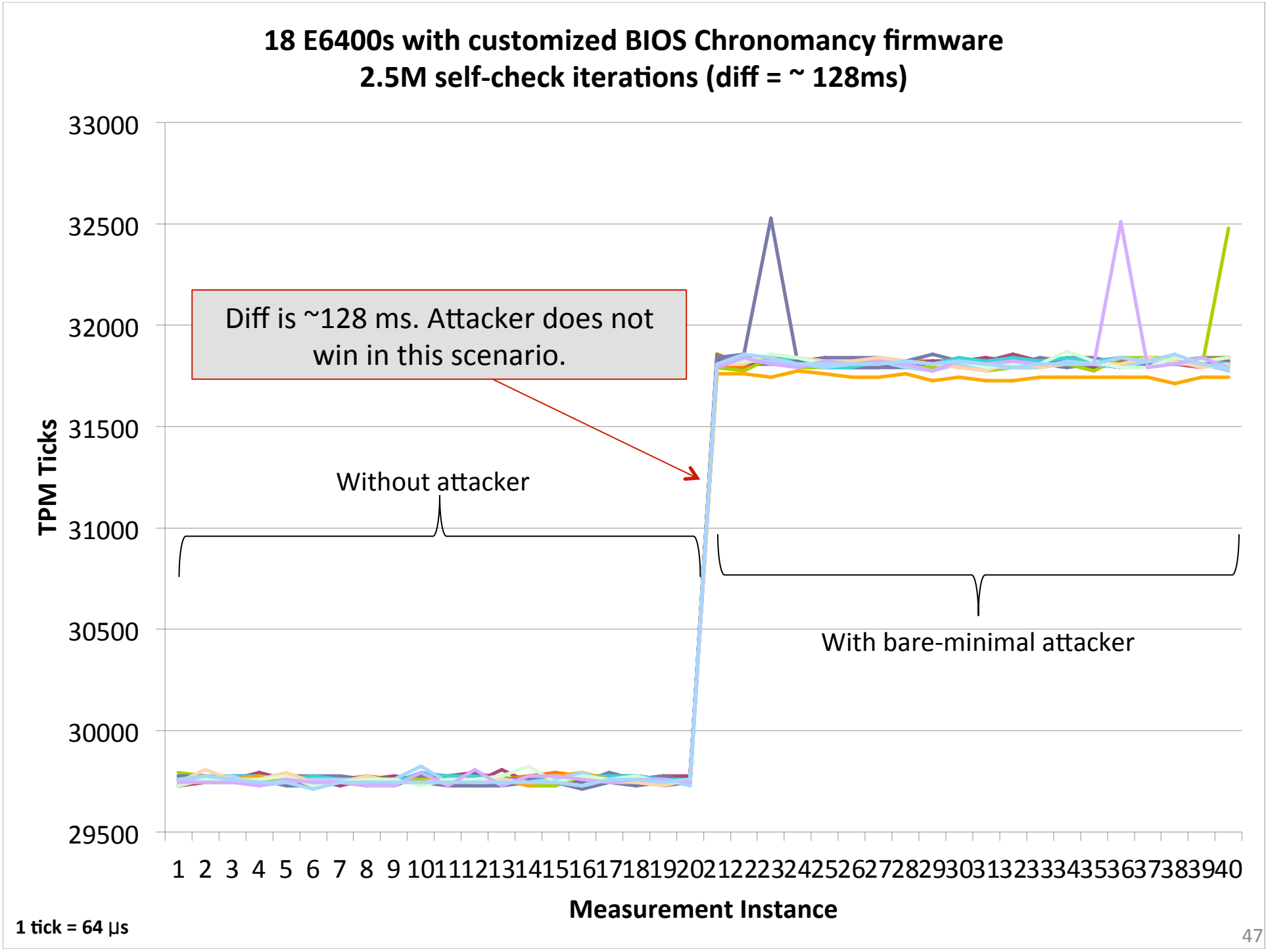


- Turns out the attacker suffers tremendous penalties when modifying the self-check.




18 E6400s with customized BIOS Chronomancy firmware
625k self-check iterations (diff = ~4.8ms)
17300
17200
17100
17000
16900
16800
16700
TPM Ticks
Diff is ~4.8 ms. Attacker can win sometimes, as shown by measurement times that overlap.
Without attacker
With bare-minimal attacker
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
Measurement Instance
1 tick = 64 µs


18 E6400s with customized BIOS Chronomancy firmware
1.25M self-check iterations (diff = ~32ms)
22200
22000
21800
21600
21400
21200
21000
TPM Ticks
Diff is ~32 ms. Attacker does not win in this scenario.
Without attacker
With bare-minimal attacker
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
Measurement Instance
1 tick = 64 µs


18 E6400s with customized BIOS Chronomancy firmware
2.5M self-check iterations (diff = ~ 128ms)
Diff is ~128 ms. Attacker does not win in this scenario.
Without attacker
With bare-minimal attacker
TPM Ticks
33000
32500
32000
31500
31000
30500
30000
29500
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
Measurement Instance
1 tick = 64 µs

TBA Summary

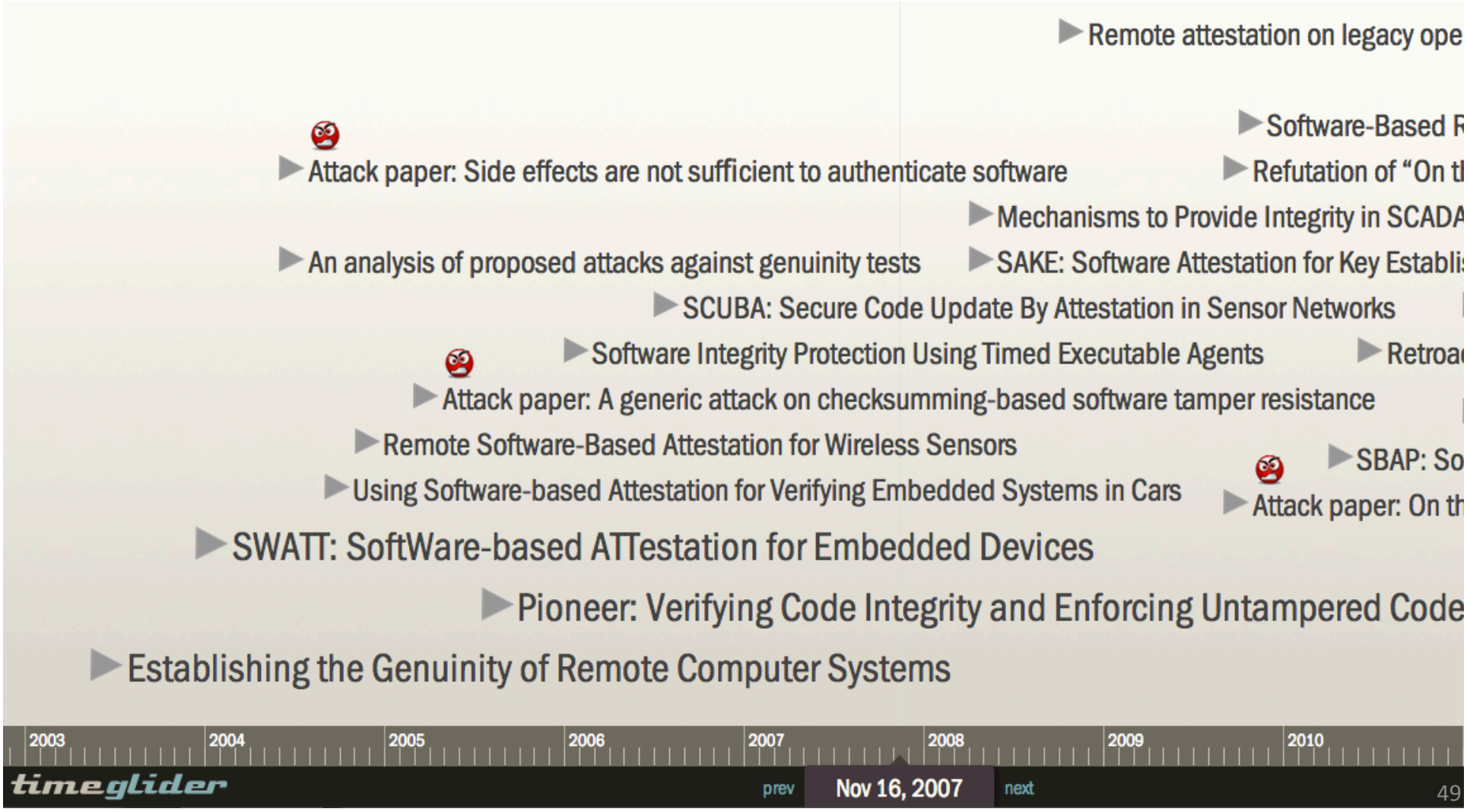
- TBA was discussed briefly just to introduce you to one technique that can mitigate the weakness of a mutable CRTM
- Is it perfect? Nope – and I'll explain why in a sec
- TBA code is open sourced for others to investigate if you feel inspired to experiment with it and improve it:
  - http://code.google.com/p/timing-attestation/
- Timeline of other related work here:
  - http://bit.ly/11xEmlV

- Of course the simplest fix would be to:

Implement the CRTM on a small, truly immutable, ROM
- Provided the remainder of the chain of trust is measured properly

The past decade

(available at http://bit.ly/11xEmlV)



TBA Problems: There are a few

- Does not prevent a TOCTOU attack
- The timer on the TPM must be reset to zero each time the system is reset
  - Provide a consistent “window of time” in which a good measurement was initiated
  - Otherwise the attacker could simply perform the measurement at a later time after having made the proper preparations
- There is no trusted way to determine whether a measurement has been “skipped” due to platform reset before (or after) TBA execution (BIOS reset, etc.)
  - Still doesn’t solve the evil maid problem
  - Measurement will run while Evil Maid’s evil boot loader is installed, but after the evil maid resets the system and removes herself the next measurement will report a clean system

Measured Boot Early in Boot Process

- One question about the measured boot process is regarding the question of ‘when’ it is to begin
- Since the spec wants each boot component to be measured before execution control is handed off to it, it seems logical that the measured boot process should begin as early as possible in the boot process
- Of course this is vendor-dependent as to when they start this process
- I’ve observed that on the Dell E6400 that this process begins very “late” after “a lot” of code has already executed (chipset configuration, SMM instantiation, etc.)
- I believe it should be done following the entry vector after the system has switched to protected mode
- And not because “it’s better if it’s run early” since the flash is still mutable regardless…
- My reason: because an analyst will be able to easily find the CRTM code and verify that it “looks right” without having to RE so much of the BIOS just to find it

Measured Boot Early in Boot Process

- Technically, this is feasible because the TPM extends hashing functions to the BIOS and it is a memory-mapped device
- So this is technically feasible:
  - Ensure TPM is mapped to memory (location is typically hard-coded in chipset)
  - Initialize the TPM itself and extend measurements to PCR0
  - As in accessing the memory-mapped BIOS, we're not actually reading memory, we're accessing a different device
- Why isn't this done? Probably because the BIOS flash and TPM are both very slow when compared to memory
- And booting quickly is the most important thing in the world!
- Vendors should at least provide the option for those who care and need it…it only has to measure a small amount of binary. (Ok I'm off my soap box now)

Better CRTMs coming down the pipe: Intel Boot Guard

- Can't say much on this at the moment (most docs under NDA, and haven't evaluated it yet anyway)
- Intel says *"Hardware-based boot integrity protection that prevents unauthorized software and malware takeover of boot blocks critical to a system's function, thus providing added level of platform security based on hardware."*
- Implements the CRTM notionally on the processor itself
- Firmware boot block is measured/verified before the processor starts executing the SPI flash entry vector
- Provides the following two basic functions
  - Measured Boot
  - Verified Boot

http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/4th-gen-core-family-mobile-brief.pdf

Better CRTMs coming down the pipe: Intel Boot Guard

- It's effectively bringing to the client a superset of the way Intel Trusted Execution Technology (TXT) on Intel Xeon servers has apparently always worked: Running a TXT Authenticated Code Module (ACM) right from CPU reset

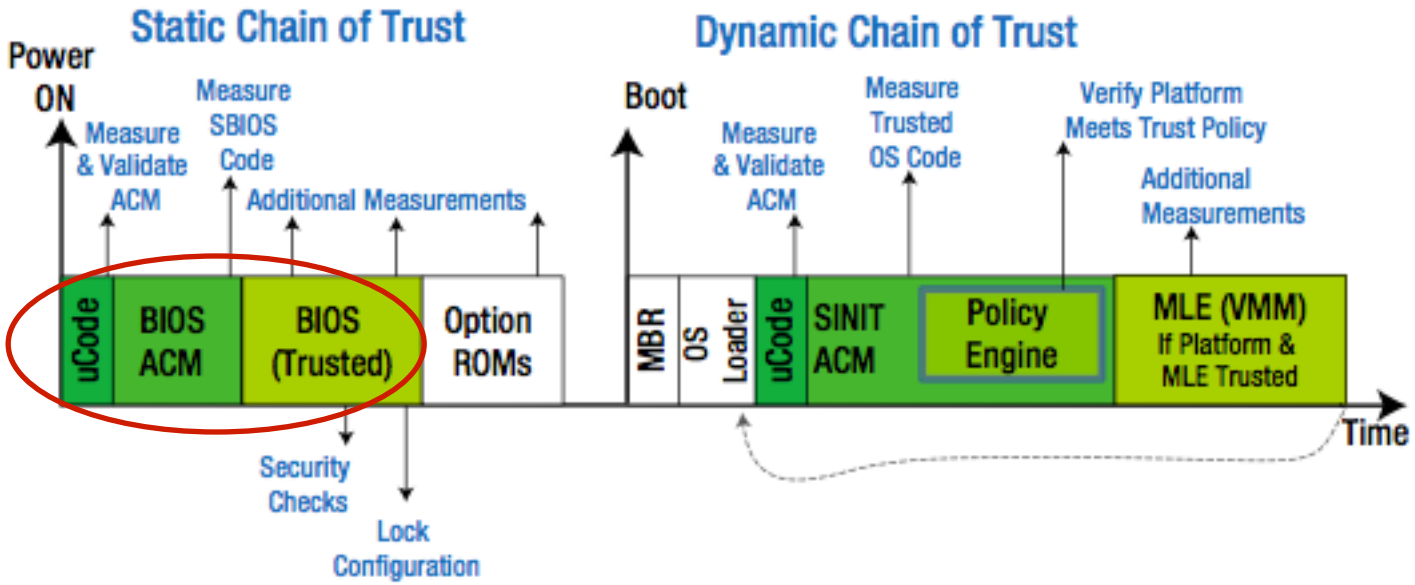


Figure 1-1. Intel® TXT launch timeline with static and dynamic chain of trust

From "Intel Trusted Execution Technology for Server Platforms"

Better CRTMs coming down the pipe: HP Sure Start

- New integrity & availability technology
- Implemented in Embedded Controller (EC)
  - They say the EC starts from a true ROM, which would essentially be the S-CRTM for subsequent measurement of BIOS
- Integrity: Checks *a portion* of the flash chip (likely only the boot block), and if it does not have the expected configuration, restores that portion from EC
- Availability: If the integrity check fails, as it might if the chip was wiped to attempt to brick the BIOS, then this provides a non-attach-probes-to-the-SPI-chip recovery
- We generally see this as a Good Thing™ , and we'd like to see more and more robust tech like this from other vendors

HP Sure Start

- Supported models as of April 2014, according to an email to us from HP
  - Elitebook 820 G1
  - Elitebook 840 G1
  - EliteBook 850 G1
  - Zbook 15
  - Zbook 17
  - EliteBook Folio 1040 G1
  - EliteBook Revolve 810 G2

TPM and Bootkits

- We have learned that signed firmware updates ensure that only an authorized BIOS can be installed to flash
- However firmware signing won't protect the system from a malicious boot loader, for example, which can be located on the hard disk
- We know that the measured boot process can detect changes to critical boot components like the BIOS and MBR
- But unless that detection is paired with something which provides protection (like Bitlocker or Secure Boot), a malicious MBR, for example, can still execute
- Detection alone could be enough if your TPM is active and you are actively observing your PCRs
  - Few seem to be

TPM: Additional thoughts

- You should activate your TPM in your BIOS
- Flawed or not, it's better than nothing ☹
- Typically the BIOS will recognize automatically that the TPM is activated and you will get all the vendors measured boot functionality "for free"
- Additionally you have to actually observe your PCRs for changes
  - Believe it or not, some people enable the TPM, check a box, and say "I'm secure now"
- Your OEM *might* have a tool you can use to that effect; otherwise use OpenTPM

Another High-Level Problem: Untrusted Tools

- So we've covered how the TPM CRTM may not really provide trustworthy information
- But every tool we use to gather system information shares this problem
- We're relying on tools which have no means of attesting that the data we intended to read was in fact the data that was reported to us
- We saw this in the Flea attack video where the PCRs were forged
  - We thought we were reading good BIOS, but in fact it was pure concentrated evil!
- An attacker could either attack the tool itself or MitM the data as it's being read by the application (via VMX for example)