2012 IEEE Symposium on Security and Privacy

New Results for Timing-Based Attestation

Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, John Butterworth
The MITRE Corporation
Bedford, MA, USA
{*xkovah,ckallenberg,cweathers,aherzog,malbin,butterworth*}*@mitre.org*

***Abstract*—In this paper we present a comprehensive timing-based attestation system suitable for typical enterprise use, and evidence of that system's performance. This system, similar to Pioneer [20] but built with relaxed assumptions, successfully detects attacks on code integrity over 10 links of an enterprise network, despite an average of just 1.7% time overhead for the attacker. We also present the first implementation and evaluation of a Trusted Platform Module (TPM) hardware timing-based attestation protocol. We describe the design and results of a set of experiments showing the effectiveness of our timing-based system, thereby providing further evidence of the practicality of timing-based attestation in real-world settings. While system measurement itself is a worthwhile goal, and timing-based attestation systems can provide measurements that are equally as trustworthy as hardware-based attestation systems, we feel that Time Of Check, Time Of Use (TOCTOU) attacks have not received appropriate attention in the literature. To address this topic, we present the three conditions required to execute such an attack, and how past attacks and defenses relate to these conditions.**

***Keywords*-remote attestation; software-based attestation; timing-based attestation; trusted platform module; TOCTOU attack**

I. Introduction

While a plethora of commercial security products are available today, the vast majority do not make use of existing academic work in the area of remote attestation. Overwhelmingly, security tools protect themselves with access control mechanisms such as file permissions, sandboxes, user-kernel separation, or OS-hypervisor separation. The ineffectiveness of this approach can be seen in the decades-long history of successful techniques for bypassing access control mechanisms.

Mechanisms are needed that can ensure security software continues to behave correctly even in the presence of an equally-privileged attacker. Previous work on *timing-based attestation* [20] [17] [16] attempted to address this need, but failed to provide experimental data for typical enterprise systems. In this paper, we present a comprehensive view of a timing-based attestation system designed for typical enterprise use and experimental evidence of its performance.

The system, implemented within our Checkmate tool suite, offers a number of contributions to the state of the art. First, we have produced a timing-based attestation implementation that functions within typical enterprise environments. Second, we have shown its performance is similar to previous research [20] despite a number of relaxed assumptions and additional checks: Our implementation does not rely on open-source network drivers; it functions in the presence of Address Space Layout Randomization (ASLR)-like kernel module loading within Windows XP, and real ASLR in Windows 7; it functions well even when the timing messages are sent over 10 network links (6 switches, 3 routers); it includes an implementation of a self-check timing measurement based on the TPM's tickstamp counter. This system is described in Section III.

While preparing this capability for deployment, we performed numerous experiments on 31 homogenous enterprise hosts to which we had temporary access; in Section IV we discuss the experimental set-up and results. The data show the effectiveness of timing-based attestation for code integrity within an enterprise setting.

Throughout the paper, we discuss possible attacks on timing-based attestation and how we addressed these attacks within our system. We also discuss the often misunderstood problem of Time Of Check, Time Of Use (TOCTOU) attacks against code integrity in Section V and how remote attestation systems must adapt their designs to defend against them.

Based on our experimental data and our early deployment experience, we believe that timing-based attestation systems like Checkmate can provide strong code integrity guarantees today, and strong control flow integrity guarantees with more work.

II. Related Work

This work was undertaken specifically to determine whether timing-based attestation systems built for general purpose PCs, like Pioneer [20] and PioneerNG [17], behaved as described when adapted to a different environment. Pioneer runs on x86-64 Linux and is implemented by inserting the attestation code into an open source network driver kernel module. PioneerNG runs in System Management Mode (SMM), is implemented as 16 bit x86 assembly, and attests to a verifier via USB rather than via the network. In contrast our code is implemented in 32 bit Windows XP making use of the existing network driver abstraction layer. We did not implement DMA protection as PioneerNG did because we consider that to be one of many TOCTOU attacks requiring more generic countermeasures, as described in Section V.

DOI 10.1109/SP.2012.45

Software designed to provide timing-based attestation has also been applied to embedded systems beginning with SWATT [21] and then expanding into areas such as verifying peripherals [10] [9], wireless sensors [18] [23] [4] and SCADA systems [22].

An interesting recent result which straddles the boundary of general purpose PCs and embedded systems is Jakobsson & Johansson's work [7] which shows the practicality of software-based attestation on mobile phones. However, we believe that their core technique of memory printing, when applied to the increased RAM on desktops, would lead to computation times of tens of seconds. We do not consider it acceptable to lock a user's system for this amount of time, and we strive to keep attestation runtime in the 100ms range, so that it is not noticeable to users.

For attacks on timing-based attestation systems, Castelluccia et al. [3] provide an example control flow based attack against a software-based attestation system using return oriented programming. They also outlined a compression attack which is not applicable to this work, because they admitted it is detectable by a change in timing. Wurster et al. [27] & Shankar et al. [24] have also suggested a type of attack which exhibits TOCTOU characteristics in the way that it changes virtual to physical mappings for the duration of measurement. Only during measurement the attacker points a virtual memory address to clean physical memory, and when not measured the virtual address points to attacker-modified physical memory. Yan et al. [28] also recently proposed an attack which exploits pipeline parallelism, and suggested a countermeasure of introducing backwards data dependency within checksum loops. This is an easy condition to achieve, and will be added to our implementation. They also proposed a TOCTOU attack predicated on control flow integrity violation. They use an idle CPU to change a function pointer (such as the return address) just in time after the self-check function has run, but before secondary measured code has run.

III. Self-checking Implementation

Our attestation system is part of a project called Checkmate, which performs both attestation and Windows kernel integrity measurement. To distinguish the two components, we will refer to the attestation system as CMA and the measurement system as CMM. The primary purpose of this paper is to focus on CMA, as CMM is described in a separate paper under submission. Our current implementation does not deal with attackers residing in system management mode or utilizing hardware support for virtualization. That is ongoing research, but some existing work [17] already indicates that virtualization-based malware will cause timing anomalies that will be detected "for free" by timing-based attestation.

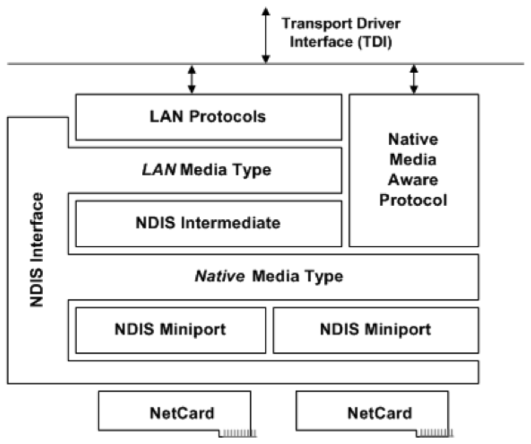

Figure 1. NDIS Architecture diagram from [12]

A. Networking

CMA is implemented as a kernel module for 32 bit Windows XP/7 (with a 64 bit Windows 7 port underway). This paper describes primarily the Windows XP implementation. It is built as a Network Driver Interface Specification (NDIS) version 5.1 Intermediate Driver (IM), sometimes called a network filter driver. In Figure 1, "NDIS miniport" drivers are the device-specific drivers written by NIC manufacturers. The box labeled "Lan Protocols" are "NDIS protocol" drivers implementing generic higher level drivers such as tcpip.sys (which implements TCP/IP) or npf.sys (which implements WinPcap.) Normally a miniport driver talks to a protocol driver through the NDIS abstraction layer. However, an NDIS intermediate driver exposes a protocol driver interface to miniports, and a miniport interface to protocol drivers. In this way it can receive incoming and outbound network traffic while residing at the lowest possible layer that is not hardware-specific. Showing that the system works while not being NIC-specific like past work is an important improvement to the system's practicality.

B. Self-check assembly

Because of limited space, we can only give an abbreviated background description and readers must have familiarity with [20] [17]. As with past work in timing-based attestation, our system is composed of two parts. A server/verifier that requests attestations that are a function of a server-supplied nonce, and a client that responds to requests by computing a self-checksum and sending it back to the server. The server/client use the existing Pioneer Protocol [20]. The verifier must be capable of re-computing the checksum as it believes the client would have. If a naive attacker modifies any of the memory checked by CMA, this will cause the checksum to be incorrect, and the verifier will know that the system is untrustworthy. A sophisticated attacker can alter the checksum computation to yield a correct checksum even in the presence of modified CMA memory, however the construction of the self-checksum function is designed specifically so that a forged checksum takes more time to compute than a genuine one. This is because the self-checksum is composed of millions of reads over its own code. If an attacker is forced to put in even one extra instruction to create the forgery, this will lead to millions of instructions of overhead to compute the forged checksum. The client code and the number of loops are tuned so that the verifier can detect over the network the delay in responding to the attestation request.

Like past work our self-check function incorporates multiple pieces of system state in order to attest to code integrity.

1) EIP_DST - The inclusion of the instruction pointer helps to indicate that the memory being executed exists in the expected memory range. This is the address of the start of the next block that will be pseudo-randomly called to. This cannot be used alone because an attacker could hardcode, rather than calculate, the value. It is therefore included only because it is readily available and costs only a single add or xor instruction.
2) EIP_SRC - This is like EIP_DST, but instead indicates the location where the calling block resides. Unlike EIP_DST, an attacker cannot precompute or hardcode this value, since it will differ according to the pseudo-random jump order.
3) DP - The data pointer helps to indicate that the memory being read exists in the expected memory range.
4) *DP - This is the 4 bytes of data read from memory by dereferencing DP. It is included so that the checksum is reading its own memory, so that the final checksum will change if there are any code integrity attacks on its own memory.
5) PRN - We include the pseudo-random number which is derived from the nonce sent by the server in the Pioneer Protocol, and updated in each block. We use the same PRNG as Pioneer.
6) EFLAGS - Included so that any changes to flags such as the trap flag, interrupt flag, or the various conditional code flags, will have to be fixed by the attacker.
7) DR7 - We place the lower 16 bits of the nonce into the upper 16 bits of the DR7 hardware debug breakpoint control register. The lower 16 bits of DR7 are set to 0, which disables all hardware breakpoints. This is read in each block to help ensure hardware breakpoints are still disabled, or makes the attacker incur fixing overhead in each block if he is still using hardware breakpoints.
8) PARENT_RET & GRANDPARENT_RET - We pseudo-randomly include either the return address on the stack that would return to the parent, PARENT_RET, or the return address that would return to the grandparent, GRANDPARENT_RET. Whichever value is selected is mixed with the PRN, so that the attacker cannot hardcode the expected value.

The incorporation of the parent and grandparent saved return instructions is in part in response to the ROP TOCTOU attack presented in [3]. But it is also in response to having had an external group attempt to attack our self-check, and having one person independently come up with a simpler form of this attack. It is because this independent assessment leveraged multiple TOCTOU attacks that we realized the real-world importance of taking them into account in designing measurement systems.

Table I
HIGH LEVEL: SELF-CHECK FUNCTION

Prolog
Block Variant 0
Block Variant 1
Block Variant 2
Block Variant 3
Block Variant 4
Small Block Variant 0
Minichecksum Fragment 0
Small Block Variant 1
Minichecksum Fragment 1
Small Block Variant 2
Minichecksum Fragment 2
epilog

As shown in Table I our code has eight blocks variants. The figure is not to scale, as the first five "large" blocks have nine sub-blocks, and the last three "small" blocks have seven sub-blocks. The two extra sub-blocks in the large blocks were only added to the increase the block size. This allowed the size of a small block plus the size of a minichecksum fragment to equal the size of a large block, for easier block start address calculation. The use of a minichecksum is borrowed from PioneerNG. It allows the primary checksum, as implemented by the blocks, to read memory only in the memory range of the self-check function itself, which includes the minichecksum fragments. This inter-mixed construction optimizes cache behavior. The minichecksum fragments together implement code that can read from arbitrary memory ranges outside of the verification function and incorporate data into the checksum. In this way the blocks check the minichecksum, the minichecksum checks the CMM Windows memory integrity measurement code, and the CMM code checks elsewhere on the system for evidence of compromise. An attacker wishing to hide from the CMM code must modify the earlier stages of the dynamic chain of trust, finally modifying the calculation of the blocks, which leads to a time overhead in their calculation.

Our checksum is treated as an array of six 4-byte values and is stored on the stack below all local variables. Like Pioneer, we store part of the checksum below esp so that any interrupts that occur while computing the result will destroy part of the checksum. The checksum is organized so that checksum[0] is at [esp-8], a gap for storing temporary values on the stack is located at [esp-4], and checksum[1]-[5] are stored at [esp] through [esp+16] respectively. Although we only use a 32 bit nonce/PRN, there is still value in having the checksum be 192 bits. An attacker wanting to precompute all possible responses to all possible nonces would require 2^{32} * 6 * 4 (96G) bytes of memory for this table. While this could reasonably be stored on a single server system, or across multiple client systems, it would not fit on a single typical client system. Additionally, because Windows loads our module at different addresses across reboots, as described in Section III-C, the attacker would have to recompute this table every time the system was rebooted. Given that TOCTOU and proxy attacks are more effective currently, as described in Sections IV-H and V, we believe this is a reasonable design optimization for the time being, as updating a larger PRN requires more instructions.

The overall structure of a block is shown in Table II, with the details of one variant of each of the sub-blocks being shown in Table III. Typically the only difference between *VAR0 and *VAR1 versions of code is a reordering of add and xor instructions in order to maintain the strongly ordered checksum property. As described in Pioneer, this construct of adds and xors is used to prevent parallelization of the self-check function, because if the sequence is not computed in the same order, e.g. (A+B) ⊕ (C+D) instead of ((A+B) ⊕ C) + D, then the result will be different with high probability. In Tables II and III, "codeStart" is the address of the beginning of the self-check function, "memRange" is the size of the self-check function, and "addressTable" is a precomputed table of the start addresses of each block. In Table III the following long-lived register conventions are in place: eax and edx are scratch registers, ebx holds the checksum loop counter, edi holds DP, and esi holds the PRN. In the body of the code, ecx is used to accumulate the mixing of values before they are mixed with the overall checksum. And before inter-block transfer, ecx is loaded with EIP_DST.

In Tables II and III the MIX_EIP sub-block starts by accumulating the value (EIP_SRC + EIP_DST) into ecx. Then in an UPDATE_PRN*[1] sub-block, the PRN is updated so that each block has a fresh PRN, and ecx is updated to hold (EIP_SRC + EIP_DST ⊕ PRN). In a READ_AND_UPDATE_DP* sub-block DP and *DP are accumulated so that ecx holds (EIP_SRC + EIP_DST ⊕ PRN + DP ⊕ *DP), and then DP is updated to a new pseudo-random location. After the READ_UEE_STATE* sub-block, ecx holds (EIP_SRC + EIP_DST ⊕ PRN + DP ⊕ *DP + DR7 ⊕ EFLAGS). In READ_RAND_RETURN_ADDRESS ecx will be updated to add in either PARENT_RET ⊕ PRN or GRANDPARENT_RET ⊕ PRN depending on bit 0 of the PRN. In CHECKSUM_UPDATE the accumulated value in ecx is mixed with the 192 bit checksum stored on the stack. Finally, in INTERBLOCK_TRANSFER the code exits if the loop counter is zero, or pseudo-randomly picks the next block to jump to based on the bottom 3 bits of PRN.

[1]A * at the end of a block name is meant to be interpreted like a regular expression, meaning any variation of characters from that point.

Table II
MID-LEVEL: BLOCK VARIANT 0

EXAMPLE_BLOCK(codeStart, memRange, addressTable)
MIX_EIP
UPDATE_PRN_VAR0
READ_AND_UPDATE_DP_VAR0(codeStart, memRange)
UPDATE_PRN_VAR1
READ_AND_UPDATE_DP_VAR1(codeStart, memRange)
READ_UEE_STATE_VAR0
READ_RAND_RETURN_ADDRESS
CHECKSUM_UPDATE
INTERBLOCK_TRANSFER(addressTable)

C. Windows-specific design considerations

Because we did not implement our code as a standalone addition to a network driver as the original Pioneer did, our kernel module has some dependencies on external code. Specifically we import functions from ndis.sys (the NDIS abstraction interface driver, hereafter referred to as *ndis*), ntkrnlpa.exe (the kernel utilizing Physical Address Extensions, *nt*), and hal.dll (*hal*). Therefore we must consider these three modules to be part of CMA's dynamic root of trust as shown in Figure 2. We achieve this by modifying our minichecksum to run over arbitrary memory ranges, unlike in PioneerNG. This is done with a switch statement that feeds range information into the start and end registers used by the minichecksum. The ability to run over multiple independent memory ranges is useful for including portions of these modules, while avoiding certain areas within them. For instance, the System Service Descriptor Table (SSDT) is a table of function pointers that are often modified by 3rd party software (despite this being discouraged by Microsoft.) This table is measured by the CMM code, and not called directly or indirectly by CMA code. Therefore we do not think it should be incorporated into the self-checksum, as that needlessly complicates checksum appraisal. Instead we separately read the portions of *nt*'s .text section before and after the SSDT. Similarly there is a complication with *hal* in that it has a region within its .text section where data is changed from zeros for the binary on disk, to stack garbage left over from code that transitions into the "Virtual 8086"

Table III
LOW LEVEL: SINGLE BLOCK WITH EXPANDED SUB-BLOCKS (VARIANT BLOCKS REMOVED)

	MIX_EIP
add ecx, [esp]	EIP_SRC ([esp]) + EIP_DST (ecx)
	ecx is then used as an accumulator
add esp, 4	Reset stack after EIP_SRC push
	UPDATE_PRN_VAR0
mov eax, esi	Create a copy of x before squaring
mul eax	eax = x*x
or eax, 5	eax = (x*x OR 5)
add esi, eax	PRN = x + (x*x OR 5)
xor ecx, esi	Mix PRN with the accumulator ecx
	READ_AND_UPDATE_DP_VAR0
add ecx, edi	Mix DP with accumulator ecx
xor ecx, [edi]	Mix *DP with accumulator ecx
mov eax, esi	Move PRN to eax
xor edx, edx	Clear edx
div memRange	edx = PRN modulo memRange
add edx, codeStart	edx=codeStart+(PRN mod memRange)
mov edi, edx	Update DP to new value
	READ_UEE_STATE_VAR0
mov eax, dr7	Copy the DR7 register
add ecx, eax	Mix DR7 with accumulator ecx
xor ecx, [esp]	Mix EFLAGS with accumulator ecx
add esp, 4	Reset stack after EFLAGS push
	READ_RAND_RETURN_ADDRESS
test esi, esi	AND PRN with self and set flags
mov eax, [ebp+4]	Move PARENT_RET to eax
JP(6)	Hardcoded bytes for if(PF) jump 6
	PF is parity flag set by test esi, esi
	The jump would land at the next xor
mov edx, [ebp]	If not jumped over,
mov eax, [edx+4]	move the GRANDPARENT_RET to eax
xor eax, esi	Xor saved ret with PRN
add ecx, eax	Mix xored saved ret with accumulator
	CHECKSUM_UPDATE
mov eax, ebx	Copy loop counter to eax
and eax, 3	Use bottom 2 bits of loop counter
	to specify which checksum memory
	entry to directly update.
xor [esp+eax*4], ecx	Xor checksum[eax+1], accumulator
	(+1 because checksum[0] is below esp)
bt [esp+0x10], 1	Set carry flag based on LSB
	of checksum[5]
rcr [esp-0x08], 1	Rotate right with carry checksum[0]
rcr [esp], 1	Rotate right with carry checksum[1]
rcr [esp+0x04], 1	Rotate right with carry checksum[2]
rcr [esp+0x08], 1	Rotate right with carry checksum[3]
rcr [esp+0x0C], 1	Rotate right with carry checksum[4]
rcr [esp+0x10], 1	Rotate right with carry checksum[5]
	INTERBLOCK_TRANSFER
sub ebx, 1	Decrement loop counter
test ebx, ebx	Check if loop counter is 0
jz setRange	If 0, jump to minichecksum switch
lea edx, addressTable	Otherwise, prepare to jump
	to next block. Load address of table
	holding start address of each block
mov eax, esi	Copy PRN to eax
and eax, 7	Use bottom 3 bits to decide which
	block to call to next
mov ecx, [edx+eax*4]	Move EIP_DST to ecx
call ecx	Call to next block
	Implicitly push EIP_SRC

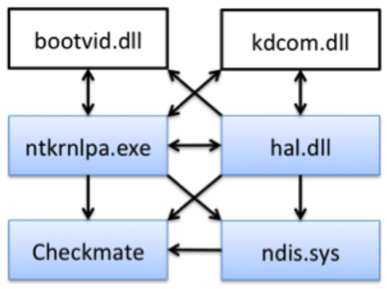

Figure 2. Checkmate kernel module dependencies

CPU mode.[2] This region can be skipped because the verifier will not be able to reconstruct the expected values. There are no such complications with reading the code from *ndis*.

The inclusion of the external modules means that in order for the appraiser to reconstruct the self-checksum, it will have to reconstruct the state of memory for these modules too. This capability is also necessary for verification of the CMM code's measurement of binaries. To achieve this, the appraiser must have a trusted copy of the file that was loaded in memory on the client system. The file is then memory mapped and processed in the same manner it would have been by the OS loader, starting with applying relocations based on the base address where the module was loaded. Verification also requires reconstructing the import address table (IAT), because in these modules the IAT is in the .text section. This is why in Figure 2 we point out *nt* and *hal*'s dependence on bootvid.dll and kdcom.dll. The appraiser must also have trusted copies of those files in order to accurately fill in the reconstructed IATs for *nt* and *hal*.

An important element of implementation on Windows rather than Linux is that the appraiser cannot assume it knows exactly where in memory the client is. This is because on Windows XP SP3 the OS loader does not respect the preferred base address listed in the binary header for kernel modules. This is despite the fact that officially ASLR support only begins in Windows Vista and later. By default for .sys kernel module files, even if the preferred base address is not already taken, the OS loader will not load it at that address. Implementation in the presence of this behavior on Windows XP led us to believe that our implementation would work equally well in the presence of ASLR on newer Windows systems, and a subsequent Windows 7 port confirmed this. The inability of the verifier to know the address where the CMA module will be located a priori necessitates that a client send back its base address with attestations. However this means that the attacker can forge the base address that is returned. We take advantage of this fact for our reference attacker in Section III-E.

[2]This *hal* self-modification is absent in Windows 7. But Windows 7 also added 3 new modules which *nt* depends on. They were easily dealt with through 3 new cases in the minichecksum switch statement.

Our module originally used Windows API calls in order to make the other CPUs spinlock while our code ran with exclusive access to a single CPU. What we ultimately found is that this code was introducing up to 16ms in non-deterministic delays in our observed network RTT. We believe this is because the APIs used Windows Deferred Procedure Calls (DPCs) to schedule the spinlocks on the other CPUs. We believe that these functions are using inter-processor interrupts (IPIs) underneath, and we could reintroduce the locking behavior in the future by using IPIs directly. However because any attacks that leveraged the fact that we weren't locking the other CPUs would be TOCTOU attacks, we think it will be more beneficial to utilize the multiple CPUs to implement a randomized overlapping scheme as outlined in Section V-C.

D. Microarchitectural optimizations

Past work such as Pioneer has suggested the need to optimize the self-check function with awareness of microarchitectural details in order to fully occupy all subunits of the processor. This attempts to ensure there are no empty slots where additional attacker code does not incur overhead. However, as past authors have made clear, there is still no mechanism to prove the optimality of a self-check implementation. Therefore, while we did make attempts to optimize our self-check through guess-and-test reconfiguration of code, we cannot guarantee the optimality of our implementation, or our attack. However, in general we do not think it is reasonable to expect security software vendors to gain deep expertise in every microarchitecture variant of x86 processors. We believe that as timing-based attestation becomes more mature it is more beneficial to work directly with chip manufacturers to provide a single reference implementation per microarchitecture. We will be making our self-check code widely available as a starting reference implementation from which future improvements and reference attacks can be built, hopefully in cooperation with chip vendors and other researchers who have deep systems knowledge.

E. Self-checksum forgery attack

It is important to remember that while timing-based attestation systems are useful to an enterprise, they are not infallible. They can be thought of as providing robust tamper-evidence, rather than tamper-proofing, for security software. Our reference attack takes advantage of the requirement on Windows for the client to send back the base address where it says the Checkmate driver is loaded. For simplicity of testing, we implemented our attack in an experimental branch of the same kernel module where the normal CMA code resides. This allowed us to toggle the attack on/off in our experiments.

As setup for the attack, when the legitimate module is loaded, the embedded attacker code reads the client binary from disk and copies it into dynamically allocated kernel heap space as shown in Figure 3. The attacker code processes the relocations of the clean client copy so that it looks as it would if the OS had loaded it at the given base address. When the attack is activated, it places an inline jump instruction as the very first instruction called in the self-check function. This jump redirects to the attacker code. The attacker then sets the data pointer to the address in the clean memory range. He does not have to incur performance overhead fixing DP in every loop because he will ultimately be lying about the base address when the response is sent.

The attacker cannot simply invoke the clean self-check function with a corrupted data pointer in order to avoid forging the EIP. This is because when the appraiser is reconstructing the self-checksum, it will reconstruct both the expected DP and EIP to be within the range where it was told the module is based. If the attacker invoked the clean self-check, with correct stack pointers, it would return to invoke clean CMM code, un-modified to hide attacker changes. Therefore the attacker must still run separate code that forges the EIP components of the checksum to be within the range where he is lying that the module is loaded.

Currently our attacker requires 14 instructions to the defender's 5 for computing EIP_SRC, EIP_DST, and transitioning between blocks, as shown by comparing Table IV to the last sub-block of Table III. The first 3 instructions of both sub-blocks are the same check for the exit condition. The next 3 instructions are the attacker looking up the original EIP_SRC from a table he computes in his prolog. This EIP_SRC is pushed onto the stack as it would be by the clean code's "call ecx". The next 8 instructions serve a dual purpose. They calculate the EIP_DST that would be put in ecx in the clean code, and they also update the currentIndex variable which helps the attacker with lookup of the next block to jump to. While the attack may not seem optimal, we experimented with variants to try to decrease the attacker overhead. For instance we tried forging the EIP_DST by moving a precomputed immediate to ecx at the beginning of each block. The performance remained unchanged. We also tried replacing the attacker's EIP_DST forgery with a precomputed table as is done with EIP_SRC forgery, instead of computing it with the multiply and add. This actually increased attacker time overhead. We will be making multiple reference attacks available with our public reference implementation. These will include attacks that are less efficient than the attack we have used, so that other researchers can potentially improve their effectiveness in ways we have not considered.

F. TPM tickstamp-based timing measurement

Schellekens et al. proposed to create a hybrid system using the existing Pioneer Protocol, but using a TPM for trustworthy timing of the self-check function [16]. On the face of it, this is somewhat contradictory; the whole

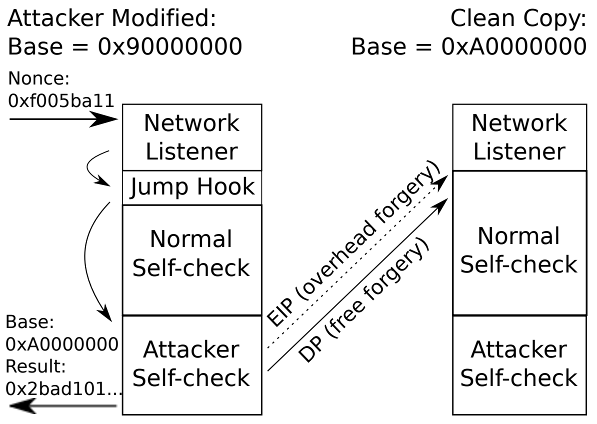

Figure 3. Visualizing the reference attack

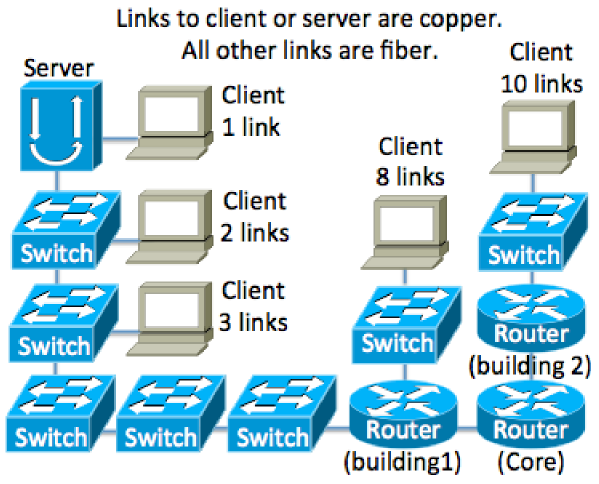

Figure 4. Network topology for timing experiments

Table IV
ATTACKER INTER-BLOCK TRANSFER FORGES EIP_SRC AND EIP_DST

	INTERBLOCK_TRANSFER_ATTACK
sub ebx, 1	Decrement loop counter
test ebx, ebx	Check if loop counter is 0
jz attackSetRange	If 0, jump to minichecksum switch
lea edx, origEipSrcArray	Get address of array of forged EIP_SRC addresses
mov eax, currentIndex	Get index of current block
push [edx+eax*4]	Push the EIP_SRC that the original call instruction would have pushed
xor edx, edx	Clear edx for use in upcoming mul
mov eax, esi	Move PRN to eax
and eax, 7	Keep the bottom 3 bits of PRN in eax
mov currentIndex, eax	Store this as the next index where the code will be executing
mov dl, BLOCK_SIZE	Move size of block to dl
mul dl	This will perform ax = al * dl
mov ecx, cleanBlockZero	Get address of start of clean blocks
add ecx, eax	Ecx = base + sizeofblock * (PRN & 7) This sets ecx to the expected EIP_DST
lea edx, attackEipDstArray	Now the attacker prepares to jump to his own next block
mov eax, currentIndex	Get the index of the next block
jmp [edx+eax*4]	Jump to the address of the next block

point of Pioneer originally was to propose a mechanism to create a dynamic root of trust on ”legacy systems” that did not have a TPM. However we believe the system is worth investigating given what experiments tell us about the prohibitively expensive amount of time that a software-only timing-based attestation system would need to run to overcome network jitter on WANs. When we asked the authors for their prototype implementation, we had found they hadn’t actually implemented or tested it. Therefore we created an implementation, and share the surprising results in Section IV-F.

IV. EXPERIMENTS & RESULTS

A. *Configuration*

The server system that requested self-check measurements was running 64 bit Windows 2008 SP2 Server on an IBM x3650 M2 with 8GB RAM, an Intel Xeon X5570 CPU at 2.93GHz, and a Broadcom BCM5709C Gigabit NIC. Our server software is implemented in C++ and uses WinPcap in order to determine the round trip times. We set a filter so that WinPcap calls back to the server whenever it sees one of our measurements outbound or inbound. When an outbound measurement request is seen, the software updates a pending measurement database entry with the timestamp in microseconds according to WinPcap. When an inbound measurement is seen, the software looks up the pending record, subtracts the sent from received time, and stores the RTT in the database.

The client systems were all 32 bit Windows XP SP3 running on Dell Optiplex 960 systems with 4GB of RAM, an Intel Core 2 Quad CPU Q9650 at 3.00GHz, and Intel 82567LM-3 Gigabit NIC.

The network topology for the experiments is shown in Figure 4. The switches are a mix of Cisco 3750/3750Gs, and the routers are Cisco 6500s. For the multi-hop experiments, we physically moved the same two hosts to each of the 1, 2, 3, 8, and 10 link distances from the server and measured at each location. For the experiments involving 31 systems, the lab where the systems resided was at the 10 link location.

For our self-checksum loop we used 2.5 million iterations, for no reason other than that is how many the original Pioneer used, and also because on the test machines this yielded a time around 100ms. Generally speaking it is desirable to keep the self-checksum around 100ms because that is the commonly accepted threshold beneath which human eyes cannot perceive changes, and it is desirable to not lock the user’s system long enough for them to detect

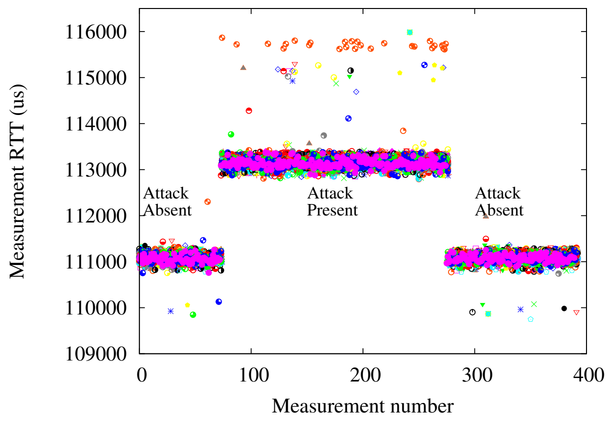

Figure 5. Network round trip time for 31 Dell Optiplex 960s over 10 links. 2 second measurement frequency.

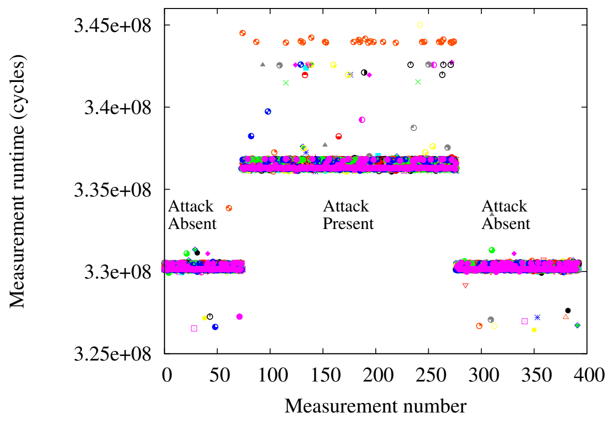

Figure 6. Rdtsc-computed runtime for 31 Dell Optiplex 960s. 2 second measurement frequency.

UI jitter. That said, the subsequent measurement which is performed by the CMM component of Checkmate will add to the time that the system is locked up. For instance hashing a small kernel module might take 1ms, but hashing the largest module, *nt* itself, can take around 50ms.

B. *Timing results for 31 systems over 10 links*

We have been running various versions of the self-check function on many different hosts for around 2 years, and this is our 6th iteration. During that time we had anecdotally confirmed that different hosts of the same hardware type exhibited the same timing characteristics. However in order to more rigorously confirm homogenous timing behavior on homogenous hardware, we were temporarily granted access to a lab of Optiplex 960s.

In Figure 5, we can see that all of the hosts' measurement times cluster very tightly. The maximum standard deviation for a host's clean measurements was 436μs, but the second highest was only 139μs. There is a one measurement partial overlap visible at the beginning and end of the attack data. This is because we did not wait until all hosts had the exact same number of measurements before pausing and toggling the attack.

In order to validate the overhead of the attacker, and to be able to confirm in our data exactly when the attack was installed, we also collected data about the self-reported runtime of the checksum. Our experimental branch of the code includes the use of *rdtsc* (read timestamp counter) instructions so that the client can calculate and report to the server its own perspective on the self-check code runtime. This would never be used in the real system, because an attacker could trivially forge the value, however it is useful during experimentation to provide insight into timing variation due to host effects vs. network effects. The data is shown in Figure 6.

Being able to contrast these graphs helps clarify that most of the outliers are due to variations in runtime, rather than any other host or network effects. From a comparison of network RTT to CPU-computed runtime we can see clear correlation in most of the outliers. We are continuing to analyze our self-check function to understand what could cause outliers to exhibit faster and slower runtimes. However the faster outliers are only around 600 to 800μs faster that their hosts' averages. So even if they were exhibited by the attack data, they would not be low enough to fall within the expected runtime bounds. Overall this data clearly indicates the ability to discriminate attacker timing overhead over 10 network links, a result that has not been previously shown.

C. *Analyzing variation of timing behavior between hosts*

Another aspect we wanted to evaluate within the all-host data set was whether it was realistic to automatically set alerting limits from the measurements collected from one host, and apply those limits to all other hosts of the same hardware type. For increased practicality of deployment in commercial software there must be some way to generate the expected baseline timing for hosts. This could take the form of a software vendor keeping a master list of expected runtimes on a per cpu/frequency basis. It could be profiled and set on the first run, under the assumption that attackers are not sitting in wait to attack a self-checksum system when it is installed. Or it could be done by requiring the customer to install and generate a baseline client timing on a known-clean machine one time per hardware configuration deployed in their environment. In this latter case, there is the need to understand how many false positives would be incurred by collecting baseline timing limits from a single host and applying those limits as the alerting threshold on all other hosts.

We generate a baseline a system by taking about 200 measurements, and then generate upper and lower control limits (average $\pm$ 3 standard deviation) for use in a control chart. Our server then has the ability to send an alert to an analyst when a pre-specified number of consecutive data points are out of control. Therefore we want to know what this threshold should be set to.

Table V
COMPARISON OF DATA FROM HOST SPECIFIED IN THE ROW AGAINST LIMITS DERIVED FROM HOST SPECIFIED IN THE COLUMN

	host 17	host 18	host 19	host 20	host 21
host 15	1/190,1	6/190,1	9/190,1	9/190,1	9/190,1
host 16	1/190,1	2/190,1	4/190,1	2/190,1	2/190,1
host 17	1/190,1	3/190,1	4/190,2	4/190,2	3/190,2
host 18	0/190,0	4/190,1	6/190,1	5/190,1	4/190,1
host 19	0/190,0	0/190,0	0/190,0	0/190,0	0/190,0
host 20	0/189,0	4/189,1	4/189,1	2/189,1	3/189,1
host 21	0/189,0	0/189,0	2/189,1	1/189,1	1/189,1
host 22	1/190,1	1/190,1	1/190,1	1/190,1	1/190,1
host 23	0/190,0	1/190,1	3/190,1	1/190,1	1/190,1
host 24	0/189,0	3/189,1	2/189,1	2/189,1	2/189,1
host 25	0/190,0	4/190,1	8/190,1	6/190,1	3/190,1

Due to a lack of space, we show only a subset of the comparison of all hosts to each other in Table V. The hosts in a row has each of its data points evaluated to determine whether it falls within the control limits generated from the data for the host given in the column. Entries are of the form X/Y,Z. X/Y is the ratio of total number of out of control measurements to total number of measurements. Z is the maximum number of *consecutive* out of control measurements out of X. The maximum Z value for the entire table suggests a threshold that could be set by the server as the number of out of control measurements it should see before it alerts, in order for it to have had no false positives in this training set. So for instance, if we generated a baseline and applied host 20's limits to host 17, we would have seen 4/190 measurements that fell outside of the limits, with a maximum of 2 of those 4 data points being consecutive. The maximum consecutive out of control data points across all comparisons was 2. This is in stark contrast to the behavior in the presence of an attacker, where there will be many out of control data points. The control limits for all hosts are show in Table VI.

D. Measurement of two hosts at different network locations

We expect the number of hops that the client is away from the server will affect the measured RTT. Therefore we tested to see how much this affected the time by measuring at vantage points from 1 direct link (connected via Ethernet crossover cable) to 10 links (the maximum link count on the testing campus.) Figure 7 shows the results of measuring two hosts when moved to different link counts.[3]

[3]There were nine outliers below the clean measurements that were cropped to provide better visibility of the gap in timing between traffic over 8 links vs 10 links.

Table VI
PER HOST CONTROL LIMITS IN MICROSECONDS

	lower limit μs	upper limit μs		lower limit μs	upper limit μs
host1	110847	111317	host2	110738	111403
host3	110635	111472	host4	110850	111315
host5	110728	111453	host6	110789	111407
host7	110733	111405	host8	110773	111410
host9	110844	111356	host10	110750	111473
host11	109786	112403	host12	110825	111308
host13	110855	111340	host14	110661	111396
host15	110740	111395	host16	110705	111415
host17	110814	111282	host18	110876	111302
host19	110853	111304	host20	110841	111295
host21	110845	111321	host22	110818	111260
host23	110849	111330	host24	110827	111348
host25	110746	111451	host26	110755	111417
host27	110839	111314	host28	110824	111330
host29	110714	111407	host30	110758	111362
host31	110837	111295			

An interesting observation is that there is very little difference between the measurements that traversed one router vs. those that traversed none. However once the traffic traverses 3 routers, the RTT increases. In the future we will test the hypothesis that the jump is primarily due to traversing the site's core router.

The procedure to set the bounds for a particular hardware type is to measure it at 1 link and at the maximum number of links on the LAN. From our previous section we know that using the limits from any host of the particular hardware type will work for the other hosts. So we can use the upper limit from a host at 11 links, and the lower limit from the host at 1 link, to obtain a very tightly bound for expected response times for this hardware type at any location on the LAN.

E. Performance measurement

Because timing-based attestation mechanisms should have exclusive control of the system while they are performing their self-check, this lockup of the system could lead to decreased performance. We wanted to know what performance effects would be caused by taking control of a system during attestation. We tested this with 2 measurement frequencies. The first was a measurement every 2 seconds, which is the fastest we allow our server to request measurements. The second was measurement every 2 minutes, which is the maximum frequency we have ever used on volunteer end users, due to previously not having had performance measurements. The data indicate there is negligible performance effects for all tests except CPU performance when measuring every 2 seconds.

AOGenMark [6] tests the CPU by performing a series of complex calculations on a polygon mesh; the higher the

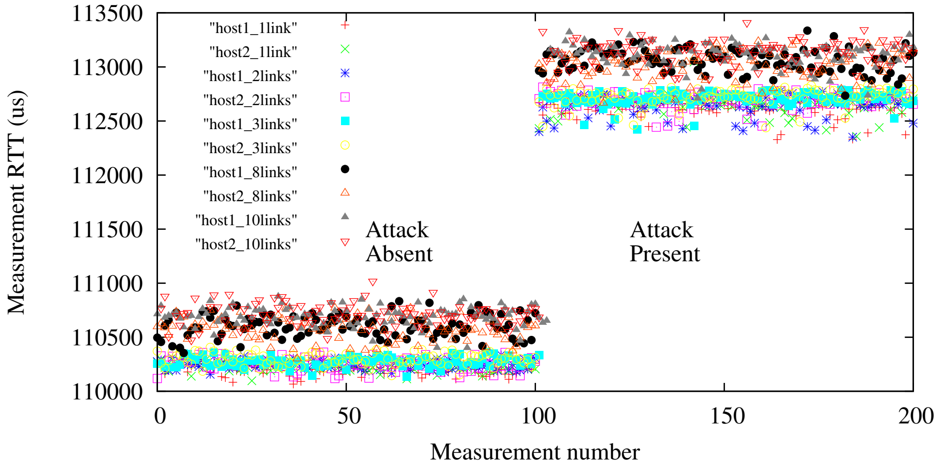

Figure 7. Network RTT for two Dell Optiplex 960s. 2 second measurement frequency, 2.5 million loops, varying network link counts.

benchmark, the faster the CPU. Additionally, the user can increase the computational load by increasing the number of samples, although the term samples is not well defined in the documentation. AOGenMark was run with 4 threads each simultaneously executing on a different core. Each thread had its number of samples set to 16. Ten tests were performed, and the mean for these tests is shown in Figure 8. This shows that for 2 minute measurement the CPU performance overhead is negligible (it was less than one standard deviation, .15 benchmark units), and with 2 second measurements the overhead is 4.8%. This is statistically signficant, but is in the expected range based on how much time the self-check is excluding all other code access, because the 111ms measurement duration is 5.55% of a 2 second measurement period.

For network throughput testing, we used iperf [25]. We set up an iperf server application on a non-target machine that communicated with a client iperf program running on the test machine. The iperf server communicated with the client machine every two seconds to determine the throughput capacity, in Megabits/sec, between the host and the client over a 180-second interval. Ten 180-second interval tests were measured. The mean and standard deviation of the throughput capacities was calculated for each 180-second run. The mean of the means was calculated over all ten of the 180-second runs. Because the means for the 2-minute measurements and 2-second measurements fall well within one standard deviation (6.32 Mbps) of the test run without measurement, there was negligible network throughput impact. The normalized network performance is shown in Figure 8.

Iozone [13] tests the time taken to write to a new file and read from a file. The test file size and memory record size used by system memory were set to 3,072 bytes. Two thousand measurements were taken for each of the tests, and

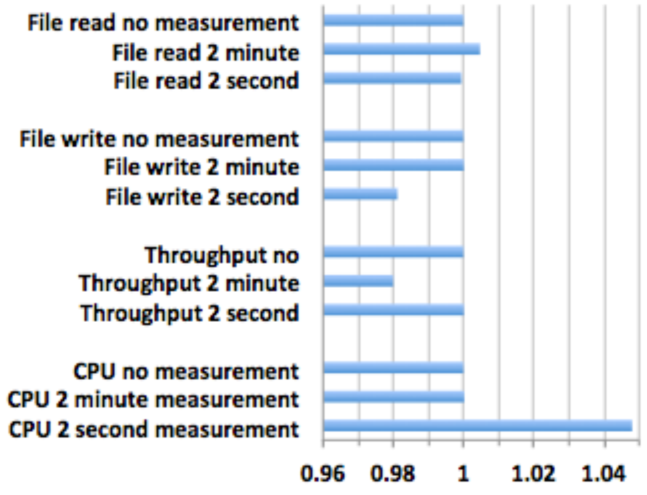

Figure 8. Normalized performance overhead for file system, network throughput, and CPU benchmarks. Values >1 indicate decreased performance, values $<$ 1indicate improved performance.

the mean was calculated over the two thousand samples. Because the means for the 2-minute measurements and 2-second measurements fall well within one standard deviation (7.07 seconds) of the test run without measurement, there was negligible read or write performance impact. The normalized file IO performance is shown in Figure 8.

Overall, measuring with either 2 minute or 2 second frequency did not significantly affect these performance measures. Although these measurements were meant to determine the impact on measurement on the endpoint, they also provide data about the effect of the endpoint's load on the observed RTT. We confirmed that the timing baseline for the benchmarked host under load differed less than a standard deviation from the host not being benchmarked.

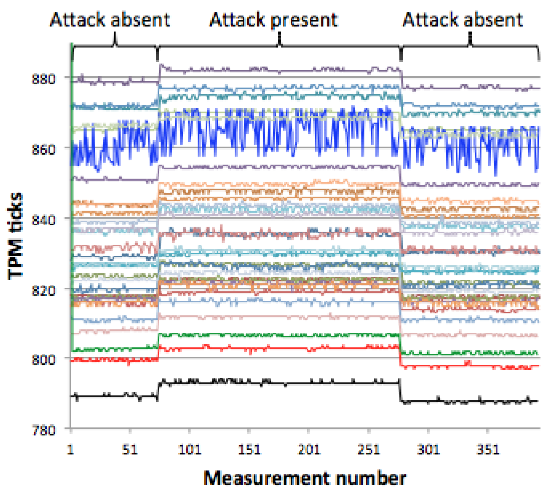

Figure 9. TPM tickstamp time for 32 Dell Optiplex 960s. 2 second measurement frequency, 2.5 Million self-check loops.

F. TPM tickstamp-based timing measurement for 32 hosts

All of the hosts for these experiments had a TPM vendor ID that corresponded to STMicroelectronics, and the chip was labeled on the motherboard as N18FPVLR. There are 32 hosts rather than 31 for the previous experiment, because one host had the wrong software version for the timing tests, but the software still was correct for TPM tests.

We requested TPM tickstamp measurements from the same machines used in the timing measurements. Note that the 5th line from the top in Figure 9 corresponds to a host for which the TPM tickstamp times are far more variable. We had used this host extensively in previous experiments, and the times that were previously reported had much lower variation. Therefore the only explanation that we can offer at this time is that we may have "worn out" this TPM through overuse.

From this data we can conclude that while the attacker is potentially detectable, contrary to expectations, each host's TPM has slightly different timing despite being the same hardware. Thus we cannot set a single baseline for the expected number of TPM ticks for the self-checksum across different hosts. We are not aware of this behavior having been previously reported, and this is one of the key results derived from actually implementing the proposed TPM tickstamp protocol. However we still believe there is room for utilizing the tickstamp timing on real systems. This is because in the ideal case a TPM is provisioned when it first enters an organization. A conservative provisioning process will also boot into a dedicated environment such as a Linux boot CD in order to communicate with the TPM with a significantly reduced possibility of a man-in-the-middle attack occurring where an attacker reports a public key for a private key that he, rather than the TPM controls. It is reasonable to expect that if the TPM requires a single initial "hands-on" provisioning process, then another tool can create a baseline for the number of TPM ticks to compute the self-check. This initial baseline can be used to bootstrap any future updates that occur to the self-check function. It is only necessary to include a special "update tickstamp" action in the client. This would invoke the previous version tickstamp-wrapped self-checksum, send the results, then invoke the new tickstamp-wrapped checksum, and send the results. If the server validates that the previous version tickstamp timing is within the expected range, then it will accept in the integrity of the new version, and can use multiple results to set a new TPM ticks baseline.

There is one further complication for practical use. Due to latency with the communication and processing performed by the TPM, the absolute time it takes to perform a tickstamp-based attestation is approximately 1.3 seconds for these TPMs and these hosts. From the previous experiment we know the absolute runtime for this self-check function with this number of iterations on these systems to be approximately .1 seconds. Therefore we are left to conclude that there is at least 1 second of overhead for performing the two tickstamp operations on these TPMs. In which case there is no way that we can decrease the number of iterations of the checksum loop in order to make the time significantly less perceptible to the user.

Thus, while the TPM tickstamp method gives us higher confidence in the detectability of an attacker than simply hashing our code and extending it into a Platform Control Register, it is constrained in the situations in which it can be used. Past work that had attestation times on the order of seconds [21] [7] have suggested use cases where the attestation is only invoked in response to special events such as an authentication attempt, where a user may be more willing to wait a short duration of the processing.

G. Variation in TPM behavior per manufacturer

When trying to understand the reason for the variation in timings between hosts with the same TPMs, we reasoned that one of the main differences between each of the hosts is that they did not have the same keys. Therefore we wanted to see if the timing behavior was in any way dependent on the signing key used by the TPM. To test this we scripted tools to rekey the TPM, perform 10 tickstamp timing measurements, average the resulting delta ticks, and then repeat. Due to the latency incurred from rekeying the TPMs and performing the tickstamp operations, and the fact that we had only temporary access to the lab machines, we were only able to do this for 50 keys on 10 hosts. Overall the results did not seem to indicate per-key variability, and an example distribution is shown in Figure 10. However, we also tested this theory on some of our Dell Latitude laptops and found anecdotally that the Broadcom TPM on these machines exhibited differing tickstamp times depending on the key. An example for a Latitude D820 is shown in Figure 11.

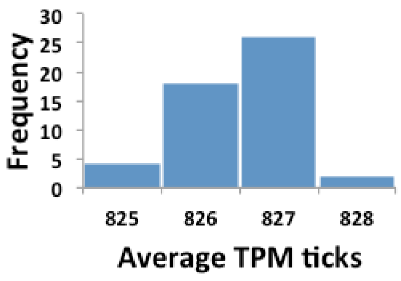

Figure 10. TPM tickstamp frequency distribution for 50 keys. STMicroelectronics TPM in Dell Optiplex 960.

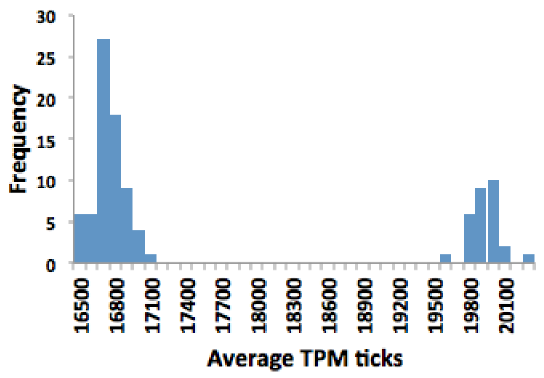

Figure 11. TPM tickstamp frequency distribution for 100 keys. Broadcom TPM in Dell Latitude D820.

We believe more work should be done to understand why different TPMs are exhibiting different timing behavior.

H. Proxy attacks against timing-based attestation

Proxy attacks are the second most effective attacks against timing-based attestation systems, behind TOCTOU attacks. This is because it is difficult to achieve message origin authentication when it is assumed that an attacker has full access to any cryptographic keys in memory. We have implemented a proxy attack to understand how much latency is incurred by a host resending its measurement request to another host. We performed these tests for the best-case situation for the attacker, where the clean host and the proxying host are on the same ethernet segment. The results are shown in Table VII. There is about a 1.5ms overhead visible when the attacker is proxying to a host of the same speed. However when the attacker proxies to a faster host, this overhead is easily negated. The results of Table VIII indicate that the granularity of TPM ticks for these TPMs is too coarse to effectively detect the 1.5ms overhead. And again, when the attacker forwards to a significantly faster machine, the timing decreases, allowing an attacker to transparently forge results.

We believe there is potential for applying VIPER's [10] system for detecting the latency inherent in proxying communications to our work. However, as was mentioned in the paper, when the latency is on the order of 1ms as it is in our results, application of their system as-is will be difficult, so it will require modification. We also believe that proxy attacks are currently detectable on LANs with host-to-host network visibility. Because the attesting system should be locked up during attestation, any communication seen destined to and returning from another host during the attestation time window is immediately suspicious.

Table VII
AVERAGE RTTS IN MICROSECONDS, WITH AND WITHOUT PROXY ATTACKS, 100 MEASUREMENTS EACH

	Average RTT No attack	Average RTT, proxied to Host 3 (Optiplex 960) Q9650 at 3GHz
Host 1, Optiplex 960 Q9650 at 3GHz	109897μs	111455μs
Host 2, Latitude D630 T8300 @ 2.4GHz	128282μs	111320μs

Table VIII
AVERAGE RUNTIME IN TPM TICKS, WITH AND WITHOUT PROXY ATTACKS, 100 MEASUREMENTS EACH

	Average ticks No attack	Average ticks, proxied to Host 3, Optiplex 960 Q9650 @ 3GHz
Host 1, Optiplex 960 Q9650 @ 3GHz	844.25 tpm ticks	844.87 tpm ticks
Host 2, Latitude D630 T8300 @ 2.4GHz	16141 tpm ticks	15720 tpm ticks

V. "THE ONLY WINNING MOVE IS NOT TO PLAY" OR "ET TU TOCTOU?"

As we have shown above, Checkmate can provide practical timing-based attestation of code integrity for COTS OSes like Windows. The ability to say "This code ran unmodified." is a powerful capability not found in today's commercial security software that we think needs to be added. Unfortunately this is not enough to make the software trustworthy.

The problem of Time Of Check, Time Of Use (TOCTOU) attacks has been briefly mentioned implicitly or explicitly in numerous trusted computing papers [20] [17] [21] [7] [9] [10] [28], but it is not often addressed head on. The only paper we are aware of which did explicitly try to tackle one facet of this problem was "TOCTOU, Traps, and Trust" [2], however it was concerned only with the gap between load-time and runtime measurement and how load time measurement was insufficient. It did not deal with the more subtle attacks that can target runtime attestation. We believe that the effects of TOCTOU attacks are currently under-stated when dealing with remote attestation systems where the attacker is assumed to be at the same privilege as the defender. Attestation always requires some amount of measurement, and all measurement systems can potentially fall victim to TOCTOU attacks due to control flow violation which occurs at a strategic point before or during the attestation code execution.

There are 3 requirements for a TOCTOU attack to be performed against a trusted computing system:

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.

These requirements are a useful conceptual framework for examining trusted computing systems, because if any one of these conditions is broken, the TOCTOU attack will not succeed with 100% probability. We differentiate guaranteed success TOCTOU attacks vs. probabilistic TOCTOU attacks because real world guaranteed attacks more devastatingly undercut the trustworthiness of trusted computing. And when the attacker can be forced into a probabilistic TOCTOU attack, defenders are forcing a race condition back on the attacker. Because whatever capability the attacker temporarily removes degrades his control, and the possibility of detection increases. For instance an attacker who was capturing keystrokes may miss important characters; an attacker who was hiding files may have them detected by 3rd party on-access scanning; or an attacker who was denying execution to security programs by terminating them before they launch could have an execution slip through.

A. Countering requirement 1

Virtualized security systems [8] [26] [19] or those using System Management Mode (SMM) [17] [1] typically counter requirement 1 only by their assumptions. They assume the attacker cannot reside at the same privilege level as the defender (that is the hypervisor layer). They utilize the opportunity for the VM to be frozen in place and measured at intervals unknown to the attacker. On the other hand, when measurement takes place on demand in response to actions performed inside the virtualized environment, it may be possible for the attacker to remove modifications before the event is triggered. Tools like Copilot [15], which perform measurement from outside of the CPU and have direct memory access can also measure memory in a way where the attacker cannot know when the measurement is about to take place.

For self-checking systems where the attacker is assumed to be at the same privilege as the defender, we do not believe it is possible to fully counter requirement 1 with just the techniques proposed to date. This is because it is seemingly always possible for an attacker to know when and what type of measurement is about to begin. This can be achieved by placing an inline hook into the self-check code at a location on the path immediately prior to the self-check reading any of its own memory. The problem is one of obvious and deterministic control flow paths to the self-check code. Even if the code's capability could be expanded to guarantee runtime control flow integrity, as we have started to do, it can no more guarantee control flow integrity *before* it runs than it can guarantee code integrity before it runs.

An example of future work to counter this requirement would be to augment control flow with a system like TEAS [5]. By injecting agents on the fly, an attacker cannot automatically analyze the code fast enough to recognize that they are providing new control flow to the existing self-check mechanism. These agents could perform the prolog of a self-check, and incorporate an initial measurement of the existing self-check agent before jumping into a random block of the existing self-check function and allowing it to run to completion. In this way the agent which is pushed to a system just in time would be able to detect the code integrity modifications that the existing self-check function cannot detect itself.

B. Countering requirement 2

Some approaches have implicitly attempted to counter requirement 2 by measuring or proposing to measure all of memory [21] [4] [7]. It was also suggested [7] to page out and overwrite all memory that is not used for the verification function. On the face of it, this would seem to counter requirement 2. However that assumes that every single page of memory that is subsequently checked when it is read back in can be validated. In practice we do not believe it will be possible to apply whitelisting to dynamically allocated memory pages, which can contain attacker code, and we do not think it is likely that a blacklist would exist for malware sophisticated enough to be targeting self-check functions.

For attestation of desktop systems this would not require abandoning the kernel, but may require augmenting a kernel agent with another smaller root of trust. Systems like PioneerNG are implemented in SMM. While conceptually SMRAM is meant to be used as a small, isolated memory region, it is only isolated from the outside. An attacker inside can access all system memory. That would mean even self-checking code in these locations would still be vulnerable to an inline hook placed at their start, followed by the attacker removing himself to a safe location in physical memory. A single un-measured function pointer used by 3rd party code would then allow the attacker back into SMM.

Therefore approaches which attempt to measure all of memory to prevent TOCTOU attacks would seem to only work when all memory under measurement can be isolated and controlled. Systems like Flicker [11] which use Intel Trusted Execution Technology, or SecVisor [19] which uses hardware support for virtualization plus an I/O Memory Management Unit, may be required to counter the attacker having an unmeasured location to hide in.

C. Countering requirement 3

Because kernel-mode self-checksumming systems have difficulty with countering the previous two requirements without significant assumption changes, we have made some improvements for countering requirement 3. We focused on removing as many generic, deterministic, TOCTOU reinstallation avenues as possible. The ability for an attacker to corrupt return addresses and have our code return to attacker code undetected was one area we mitigated. Because of our use of imported functions, an attacker could gain control soon after the self-check is done by placing an inline hook or an IAT hook into code we call. Our extension of the minichecksum mechanism to cover arbitrary ranges helped mitigate this. And the existing technique of placing checksum data onto the stack so that interrupts destroy it is another mechanism that tries to maintain control flow integrity in addition to the existing code integrity. Although software-based attestation constructions are built primarily for code integrity [14], if they do not make these inclusions of control flow integrity, they remain vulnerable to TOCTOU attacks.

However there still remain other mechanisms for the attacker to perform a TOCTOU and regain control soon after our code releases control of the processor. For instance we have implemented a TOCTOU attack which uses a Windows DPC to schedule attacker code to run. The attacker places himself as first to be removed from the DPC queue, which begins emptying very soon after control is released by our kernel module. When the attacker code runs, it reinstalls the hook that allows him to gain control when attestation is about to begin. This is a simple and effective way for the attacker to never have his modifications detected by the attestation mechanism. It is of course a losing game to engage in an arms race and have the attestation mechanism measure the DPC queue and every other way the attacker can reinstall himself.

Future work can combat this by making it difficult for the attacker to know the true end time of the self-check function. Such an approach could be achieved by having multiple CPUs invoking the same self-check function in parallel. Existing approaches such as PioneerNG & MT-SRoT [28] try to have multiple CPUs finish their checksum as close to the same time as possible. Instead checksum completion could be displaced in time so that when one self-check completes, others are still running on other CPUs. If an attacker has set himself to reinstall as soon as possible after the self-check is done on one CPU, the other CPUs may end up reading his reinstallation of his modifications. It would then also be desirable to randomize the order in which CPUs are invoked, so that the attacker cannot know which CPU will finish last, and simply schedule himself to only reinstall after that CPU's self-check finishes. While we believe there will always be the possibility for attacks to use TOCTOU attacks with course granularity reinstall (e.g. a conceptual "sleep(10)"), if the defender can force the attacker to temporarily relinquish control, it is a higher measure of success than is achieved today. The defender has then opened a window of time in which the attacker is more vulnerable to detection and removal.

VI. Conclusion

In this paper we have shown the results of independent implementations of both software and hardware timing-based attestation systems. We have shown that an attestation system for a commodity OS can compensate for ASLR effects, does not need to know how to talk directly to NIC hardware, and has attacker overhead which is detectable over 10 network links. We have also shown that contrary to expectations, TPMs of the same model and manufacturer return a different number of ticks for computing the same function. This means that unlike software timing-based attestation, no single "expected" baseline can be set for different hosts; each host must be baselined independently. We have also clarified that the majority of generic attacks against timing-based attestation systems to date are in fact TOCTOU attacks. We described the conditions necessary for an attacker to achieve a TOCTOU attack, as well as areas of future work to create generic countermeasures against TOCTOU attacks.

We currently consider timing-based remote attestation to be in its infancy, and it looks much like the early days of cryptography, where heuristics abounded. Even though cryptography has been formalized to the point of having provably secure systems, these are not actually the systems being used on a day to day basis. Instead, algorithms like Rijndael were accepted for the AES standard based on its resistance to established attack techniques, as well as tradeoffs such as performance. In the same way we believe that timing-based attestation mechanisms can be made more robust through increased research and implementation in this area. To this end we will be making our current reference implementation openly available[4], in the hope that others will help further improve the state of the art for remote timing-based attestation.

References

[1] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the ACM conference on Computer and Communications Security*, CCS, pages 38–49, 2010.

[2] S. Bratus, N. D'Cunha, E. Sparks, and S. W. Smith. TOCTOU, traps, and trusted computing. In *Proceedings of the International Conference on Trusted Computing and Trust in Information Technologies*, Trust, pages 14–32, 2008.

[4]http://code.google.com/p/timing-attestation

[3] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the ACM conference on Computer and Communications Security*, CCS, pages 400–409, 2009.

[4] Y. Choi, J. Kang, and D. Nyang. Proactive code verification protocol in wireless sensor network. In *Proceedings of the International Conference on Computational Science and its Applications - Volume Part II*, ICCSA, pages 1085–1096, 2007.

[5] J. A. Garay and L. Huelsbergen. Software integrity protection using timed executable agents. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS, pages 189–200, 2006.

[6] Geeks3D. Aogenmark 1.3.0: Simple multi-core cpu benchmark. 2011, http://www.geeks3d.com/20110513/aogenmark-1-3-0-simple-multi-core-cpu-benchmark. Accessed: 10/21/2011.

[7] M. Jakobsson and K.-A. Johansson. Practical and secure software-based attestation. In *Workshop on Lightweight Security Privacy: Devices, Protocols and Applications (LightSec)*, pages 1 –9, March 2011.

[8] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the ACM conference on Computer and Communications Security*, CCS, pages 128–138, 2007.

[9] Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing (Trust)*, June 2010.

[10] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2011.

[11] J. M. McCune, B. Parno, A. Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, April 2008.

[12] Microsoft. Ndis drivers (ndis 5.1) (windows driver kit). September 7th 2011, http://msdn.microsoft.com/en-us/Library/ff556938(v=VS.85).aspx. Accessed: 11/01/2011.

[13] W. D. Norcott and D. Capps. Iozone filesystem benchmark. 20106, http://www.iozone.org/. Accessed: 10/21/2011.

[14] A. Perrig and L. van Doorn. Refutation of on the difficulty of software-based attestation of embedded devices. 2010, http://sparrow.ece.cmu.edu/group/pub/perrig-vandoorn-refutation.pdf. Accessed: 11/01/2011.

[15] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM, pages 13–13, 2004.

[16] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Electron. Notes Theor. Comput. Sci.*, 197:59–72, February 2008.

[17] A. Seshadri. *A Software Primitive for Externally-verifiable Untampered Execution and its Applications to Securing Computing Systems*. PhD thesis, Carnegie Mellon University, 2009.

[18] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure code update by attestation in sensor networks. In *Proceedings of the ACM workshop on Wireless security*, WiSe, pages 85–94, 2006.

[19] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of ACM SIGOPS symposium on Operating systems principles*, SOSP, pages 335–350, 2007.

[20] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the ACM symposium on Operating systems principles*, SOSP, pages 1–16, 2005.

[21] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[22] A. Shah, A. Perrig, and B. Sinopoli. Mechanisms to provide integrity in SCADA and PCS devices. In *Proceedings of the International Workshop on Cyber-Physical Systems - Challenges and Applications (CPS-CA)*, June 2008.

[23] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *ESAS*, pages 27–41, 2005.

[24] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the conference on USENIX Security Symposium*, SSYM, pages 7–7, 2004.

[25] L. Ventura. Iperf on windows. 2010, http://linhost.info/2010/02/iperf-on-windows/. Accessed: 10/21/2011.

[26] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[27] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 127–138, 2005.

[28] Q. Yan, J. Han, Y. Li, R. H. Deng, and T. Li. A software-based root-of-trust primitive on multicore platforms. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, ASIACCS, pages 334–343, 2011.