Remote Attestation of Software and Execution-Environment in Modern Machines

Michael Kiperberg
Deparment of Mathematical IT
University of Jyväskylä
Finland
Email: mikiperb@student.jyu.fi

Amit Resh
Deparment of Mathematical IT
University of Jyväskylä
Finland
Email: amit.resh@jyu.fi

Nezer J. Zaidenberg
Deparment of Mathematical IT
University of Jyväskylä
Finland
Email: nezer.j.zaidenberg@jyu.fi

Abstract—**The research on network security concentrates mainly on securing the communication channels between two endpoints, which is insufficient if the authenticity of one of the endpoints cannot be determined with certainty. Previously [1], [2] presented methods that allow one endpoint, the authentication authority, to authenticate another remote machine. These methods are inadequate for modern machines that have multiple processors, introduce virtualization extensions, have a greater variety of side effects, and suffer from nondeterminism. This paper addresses the advances of modern machines with respect to the method presented in [1]. The authors describe how a remote attestation procedure, involving a challenge, needs to be structured in order to provide correct attestation of a remote modern target system.**

I. Introduction

Software, hardware and hybrid solutions for computer-systems security are facing modern cyber-breaches, viruses, worms, rootkits and other malicious executables. These security solutions may be implemented successfully by remotely creating a trusted application container. The application container is executed on a, possibly, already compromised system. One possible embodiment to manage this security paradigm involves an authentication authority, which can administer an authentication procedure to a remote machine.

The problem of remote software authentication, determining whether a remote computer system is running the correct version of a software, is well known [1]–[8]. Equipped with a remote authentication method, a service provider can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only authenticated gaming consoles can be allowed to connect to the gaming networks [9]–[11] and only authenticated bank terminals can be allowed to fetch records from the bank database [12].

The research in this area can be divided into two major branches: hardware assisted authentication and software-only authentication. While in theory, hardware assisted authentication may provide more conclusive results regarding the authenticity of a remote machine, in practice the hardware fails to provide additional security due to inappropriate designs of currently available operating systems [5].

Hardware assisted authentication uses an external hardware component, such as Trusted Platform Module (TPM) to compute a cryptographic hash of the computers hardware and software configuration and attest it.

Usually [13]–[15] the TPM is used as the root of the chain of trust. The TPM measures the authenticity of the BIOS. The BIOS then measures the authenticity of the boot loader and so on. Unfortunately, all common modern operating systems (e.g. Linux, Windows, OS X) allow the user to load drivers for execution with the same privileges as the operating system itself, i.e. ring 0 on x86 and x64 hardware. Malicious or buggy drivers, which are executed with high privileges, allow random code execution that makes it possible to circumvent the authenticity measurements of the TPM.

Software-only authentication usually targets a specific instruction set architecture that varies from ATMega [3], through Pentium [1] to Intel Core [16]. The authentication entails simultaneously authenticating some software component(s) or memory region, as well as verifying that the remote machine is not running in virtual or emulation mode. Software-only authentication methods may also involve a challenge code, that is sent by the authentication authority, and executed on the remote system. The challenge code computes a result that is then transmitted back to the authority. The authority deems the entity to be authenticated if the result is correct and was received within a predefined time-frame. The underlying assumption, which is shared by all such authentication methods, is that only an authentic system can compute the correct result within the predefined time-frame. The methods differ in the means by which (and if) they satisfy this underlying assumption.

A hybrid approach, in which the TPM is used as an external source of time, was described in [5]. According to this approach the TPM is used to time-stamp the beginning and the completion of the challenge execution, thus reducing the unpredictable deviations in time caused by network delays. The TPM needs to be built on tamper proof hardware, an assumption that is not always true as was shown by Tarnovsky [17], [18].

Pioneer [2] is a software-only component designed to provide execution of a remotely authenticated executable on an untrusted and possibly compromised legacy host system. Pioneer is composed of a dispatcher system that is used to manage a challenge-response protocol with the untrusted platform, where an authenticated executable is to be run. The methodology of Pioneer is based on a verification utility, which first establishes itself as a root of trust, by executing code that both checksums itself and verifies that it is running. The verification utility is randomized by receiving a challenge seed from the dispatcher. Once trusted, the verification utility proceeds to authenticate the executable in question.

Pioneer is based on two assumptions on the untrusted platform: it has a single logical processor, and it does not contain a virtualization extension. Logical processors multiplicity, which was introduced in modern CPUs, violates the assumptions of Pioneer. The authors propose a remedy for this vulnerability by introducing a data dependency between the different parts of the challenge [16], thus preventing its parallel execution. Pioneer execution on processors with a virtualization extension is discussed in [19]. The authors describe a modification to the original method which allows not only to achieve consistent results on all processors but also to employ intermediate variations to detect virtualized environments.

The method proposed in [1] produces the result by computing a cryptographic hash of a specified memory region. Any computation on a complex instruction set architecture (Pentium in this case) produces side effects. These side effects are incorporated into the result after each iteration of the hashing function. Therefore, an adversary, trying to compute the correct result on a non-authentic system, would be forced to build a complete emulator for the instruction set architecture to compute the correct side effects of every instruction. Since such an emulator performs tens and hundreds of native instructions for every simulated instruction, the authors of [1] conclude that it will not be able to compute the correct result within the predefined time-frame.

The conclusion is probably true for the instruction set architecture that was considered in [1]. Modern instruction set architectures allow to construct an "emulator" that performs a single native instruction for every simulated instruction. This construction is provided through a virtualization extension of modern Intel and AMD processors. In short, the virtualization extension allows the software to specify a set of events to be intercepted and a function to be called on their interception. The events can be privileged instruction execution, memory access, interrupts, etc. Since the software can alter this set at any point, it can disable interception of the events that can occur during the execution of the challenge and re-enable their interception when the challenge completes.

Another feature of modern processors that was not discussed in [1], is multi-processing. During the execution of the challenge by one logical processor, another logical processor may affect the caches, which will lead to an incorrect result. Moreover, the additional computational power of other logical processors can potentially be utilized to deceive the authentication authority.

In this paper we discuss the deficiencies of the method described in [1] on modern machines that arise from the virtualisation extension and multiplicity of logical processors. We present solutions to the discussed problems.

1) R $\rightarrow$ A: R's configuration
2) A $\rightarrow$ R: Authentication challenge
3) R $\rightarrow$ A: Result and random value
4) A $\rightarrow$ R: Sensitive information

Fig. 1. This figure depicts a possible communication protocol between the authentication authority (A) and the remote machine (R).

This paper is organized as follows. Section II presents methods of software-only remote authentication described in [1], [2]. In section III we discuss the implications of the virtualization extensions on these authentication methods. Complications that arise from logical processors multiplicity are discussed in section IV. Section V shows how side-effects information on modern processors can be obtained and fully utilized. Non-deterministic behavior of modern processors is described in section VI. In section VII we discuss the conditions that can allow an adversary to deceive the authentication method described in this paper. Section VIII concludes our results.

II. Remote Authentication of Legacy Machines

Remote authentication methods define a protocol between the authentication (attestation) authority and the remote machine. The protocol enables the authentication authority to determine whether the remote machine is authentic.

Figure II depicts a possible structure of the authentication protocol.

The initial messages of the protocol carry information about the current configuration of the remote machine (transmitted by the remote machine). Following this exchange, the authentication authority transmits a message containing the challenge code to be executed on the remote machine. The remote machine executes the challenge, which computes a result, that is a cryptographic hash of some memory region, possibly with some additional information, and transmits it back to the authentication authority. The authentication authority verifies that the result is correct and was received within a predefined time-frame. If both are true the remote machine is considered authentic.

For practical reasons, the remote machine can generate a random number, concatenate it to the result, and encrypt both values before sending the reply to the authentication authority to avoid replay attacks. The remote machine can then use this random value, called the session key, as a proof of its authenticity.

For example, this value can be used as an encryption key to securely transmit some sensitive information from the authentication authority back to the authenticated remote machine. Clearly, an unauthenticated machine will not be sent this sensitive information.

The structure of the challenges and the hardness assumptions vary between authentication methods. Some methods [2], [3] choose the code of the challenge carefully and guarantee that the challenge constitutes the most efficient computation of the desired result. Other methods [1], incorporate side effects into the computed result, thus, in some sense, utilizing the entire processor circuitry in result computation. The goal of both types of methods is to make it impossible to emulate execution of the challenge on a non-authentic machine within the predefined time-frame.

Another similarity between the structures of the challenges produced by both types of methods is the division of the challenge into blocks and the unpredictable control flow between the blocks. The control flow depends on the intermediate values of the result. An invalid intermediate value produces a different control flow, which in turn naturally leads to an invalid final result.

Following each authentication request, a pseudo-random challenge is transmitted, to eliminate replay attacks. The authentication authority generates the challenges and computes their results ahead of time. The blocks, their relative order and the control flow are chosen pseudo-randomly during the generation phase.

The blocks are constructed for a specific architecture. Advances in instruction set architectures can potentially render the current blocks obsolete, by allowing new types of attacks that are not prevented by the current variety of block types. The most significant advances that require special consideration are multi-core architectures and virtualization extensions.

The methods described in [2], [3] are subject to attacks on multi-core processors [16]. The additional computational resources can be utilized to deceive the authentication authority. The authors of [16] propose to mitigate this attack by widening the variety of blocks. The effect of virtualization extensions on the methods described in [2], [3] were studied in [19]. Some of the operations performed by the challenge blocks produce different results in presence of an active virtual machine monitor, thus producing an invalid final result. The authors explain not only how to accommodate this diversion, but also how this diversion can be incorporated into the computed result, thus providing the authentication authority with information regarding the configuration of the remote machine.

The structure of blocks is discussed in [1]. The blocks can be one of two types: blocks incorporating memory content and blocks incorporating side-effects. Blocks of the first type read content of memory from some pseudo-random location and incorporate it into the accumulated result. Blocks of the latter type fetch some information regarding side-effects from the processor or the environment and incorporate it into the computed result using a non-commutative calculation (with regard to blocks of the first type). For example, if blocks of the first type use addition, blocks of the second type can use exclusive-or or rotate.

Every instruction that is executed by a processor modifies its internal state. Some modifications result from the definition of the instruction operations; others — are performed by the processor to improve performance, e.g. cache population, or for debugging and profiling purposes, e.g. L3 cache miss count. Previously, processors were allowed to observe the state of side-effects directly. Current versions of processors provide a different mechanism: performance counters. The processor defines pairs of registers: an event selection register, which allows the software to specify the execution event to be counted, and a monitoring counter register, which is increased on each occurrence of the event specified by the first register. The values of the counter registers can be considered the state of the side-effect and as such can be incorporated into the result.

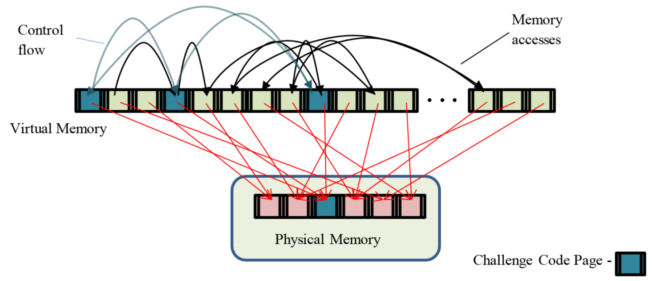

Fig. 2. The figure depicts the mapping between the virtual and the physical memories. The challenge is stored only in one page of the physical memory (the blue square) but can be accessed via many virtual pages. During the execution of the challenge the control flow is transferred between blocks in different virtual pages to utilize the ITLB. The memory is accessed by the challenge in pseudo-random order. The same data can be read multiple times via different virtual pages.

It is desirable to construct the challenge in a way that maximizes the side-effects produced by its execution. One of the side-effects that were considered in [1] is the TLB management system. TLBs store translations of virtual addresses to physical addresses of pages that were recently accessed. Modern processors contain separate TLBs for instructions and data as well as a shared TLB of a higher level, which is larger but slower. When a new translation needs to be stored in a TLB with no free slots, one of the slots is evicted according to some policy, which varies between processors. In order to achieve high utilization of the TLBs the authors of [1] propose to map a large virtual memory region that maps a smaller physical memory region that is to be authenticated (Fig. II). The challenge then can compute the hash by reading the contents of the physical memory region through different pages of the virtual memory region, thus fully utilizing the TLBs and inducing more side-effects. A typical layout of the physical memory region that is mapped by the virtual memory is depicted in Fig. II.

III. Virtualization Extension

Virtualization extension instructions are an extension to the x86 instruction set architecture that allows isolation of multiple operating systems efficiently, thus providing means to construct virtual machine monitors [20], [21]. Previously, construction of virtual machine monitors involved binary instrumentation and required modification in the code of the hosted operating systems.

There are slight differences between Intel's and AMD's implementation of the x86 virtualization extension. In this

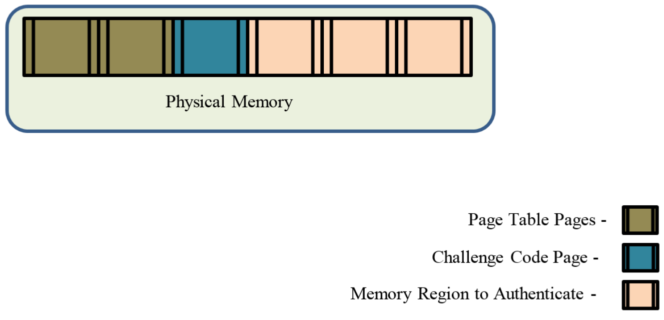

Fig. 3. The figure depicts the layout of the physical memory that is mapped by the virtual memory. The brown squares correspond the pages occupied by the virtual page tables. The challenge is stored in single blue square, which represents a single page. Other squares represent the physical memory region that is authenticated by the challenge. Note that the challenge simultaneously authenticates the contents of the entire range of physical pages: page-table, challenge-code and memory region to authenticate.

paper we will discuss only Intel's implementation and mention the differences where they are important for the discussion.

A virtual machine control structure (VMCS) is defined for each virtual environment managed by a virtual machine monitor (VMM). This structure defines the values of privileged registers, the location of the interrupt descriptors table, and additional values that constitute the internal state of the virtual environment. In addition, this structure defines the events that the VMM is configured to intercept, and the address of the function that should handle the interception. The act of control transfer from the virtual environment to a predefined function is called vm-exit and the act of control transfer from the function back to the virtual environment is called vm-entry. Upon vm-exit the function can determine the reason of the vm-exit by examining the fields of the VMCS and altering them, thus altering the state of the virtual environment, as it wishes. Finally, the VMCS can define a mapping between the physical memory as it is perceived by the virtual environment and the actual physical memory. As a consequence, the VMM can prevent access to some physical pages by the virtual environment. Moreover, the virtual environment will be unaware of this situation.

Interception of some events cannot be disabled, while interception of others cannot be enabled. For example, execution of the CPUID instruction always causes a vm-exit, while execution of the SYSCALL instruction never causes a vm-exit. Processors produced by AMD allow disabling interception of all events.

The existence of the VMM, does not affect the internal state of the processor. Therefore, the authentication method described in [1] can succeed in presence of a VMM. However the VMM can intercept execution of privileged instructions and modify their behavior, thus acting as malicious code. The code itself can be hidden using the physical mapping as described above.

We suggest the following method for VMM detection. Since Intel processors do not allow to disable interception of the CPUID instruction, execution of this instruction forces a vm-exit. On vm-exit, the processor loads the first instruction of the function whose address is specified in VMCS. This behavior alone will affect some of the caches, regardless of the actual implementation of the function. The lookup of the address modifies at least one entry of the ITLB and the higher level TLB (STLB). Fetching the first instruction modifies at least one entry in the instruction cache, L2 cache and L3 cache. In addition, execution of such an instruction takes much more time when a VMM is active. Therefore, we propose to widen the variety of blocks by adding blocks that produce events whose interception cannot be disabled. An example of such a block is a block that contains a CPUID instruction.

Unfortunately, on processors produced by AMD there are no such events that are guaranteed to be intercepted. However, the virtualization extension is enabled on AMD by setting a bit of a model specific register called EFER. We propose to add a block that reads the value of this model specific register and incorporates it into the result. If the bit is set in the value of this register, as perceived by the challenge, then the authentication will fail. If the bit is not set, then either a VMM is not active or it is active but it intercepts accesses to the EFER register and alters its behavior. In the latter case, however, the interception will modify the internal structure of the processor and will be detected as a result of side-effects. Thus, we force an adversary VMM to intercept accesses to the EFER register by incorporating its value in the result.

IV. Accommodating Multiple Processors

Modern processors consist of several execution units, called logical processors, each of which contains separate execution units (instruction decoder, branch predictor, arithmetic logic unit (ALU), floating point unit, etc.). The processor has multiple units of cache memory: translation lookaside buffer (TLB) caches, instruction cache and data caches. TLB caches are used to store information regarding virtual-to-physical address translations. The caches are separated into instruction TLB and data TLB and a unified, larger TLB. The instruction and data TLBs are separate for each logical processor while the unified TLB is shared by a group of logical processors, called a core. Similarly, some caches, like the instruction cache and the L1 cache, are separate for each logical processor while the L3 is shared between all the logical processors. In addition, for simplicity, the caches obey the inclusion policy, by which the higher level caches include all the information contained in the lower level caches. This policy implies that if a line is evicted from a higher level cache it has to be evicted from all the caches beneath. Therefore any logical processor can cause eviction of data from a cache that is owned solely by another logical processor.

We can conclude from the above discussion that in order to preserve determinism of the cache memories state it is required to "freeze" all logical processors but the one executing the challenge. Unfortunately, simple solutions, like idle-loops are not sufficient, since they affect the instruction TLBs and the instruction caches.

```
01. static step = 0;

02. WaitFor(value):
03.   forever:
04.     MONITOR step
05.     if step=cur_proc break
06.    MWAIT step

07. ScheduledFunc(cur_proc,
              total_procs, challenges):
08.   step <- step + 1
09.   wait_for(total_procs + cur_proc)
10.   execute challenges[cur_proc]
11.   encrypt the result
12.   step <- step + 1
13.   WaitFor(2 * total_procs)
```

Fig. 4. Pseudo-code of the challenge execution for multiple logical processors

We suggest to use the MONITOR/MWAIT pair of instructions to “freeze” other logical processors during challenge execution. These instructions take a specified memory range and put the processor in an idle state until the contents of that specified memory region is modified. Since no instructions are executed, the caches are not affected by the idle logical processors.

The pseudocode of our solution is given in Fig. IV. The protocol is executed by one of the logical processors, which receives the challenge and schedules the routine *ScheduledFunc()* (line 07) on all logical processors. The logical processors use the static variable *step* (line 01) for synchronization. Only two manipulations are performed on this static variable: increment and comparison. The comparison is performed by the routine *WaitFor()*, which loops until the value of the static variable *step* becomes *value*.

The loop uses the MONITOR and MWAIT instructions that block until the specified memory region is written by some other logical processor. Consider the execution of *ScheduledFunc()* by the Nth logical processor. Line 09 blocks until all processors reach line 08 and logical processors $0, 1, \ldots, N-1$ reach line 12, i.e. complete the challenge execution. Then the Nth logical processor continues by executing the challenge and encrypting its result with the public key of the authority. Finally the Nth logical processor awaits completion of the challenge execution by all other logical processors. Forcing each logical processor to execute its own challenge, prevents an un-authenticated logical processor from copying the result of the challenge calculated by an authenticated logical processor.

We note that during the execution of the challenge by a logical processor, all other logical processors execute the MWAIT instruction. The MWAIT instruction does not affect the internal structures of the processor but prevent other core to affect the tested core.

V. Performance Counter Chaining

Modern processors manufactured by Intel and AMD provide a facility to count occurrences of side-effect events, internal to the CPU circuitry, called performance events. The main goal behind this feature is to support CPU performance monitoring. Performance events are defined as internal CPU-circuitry state changes resulting from instruction execution, but not linked directly to the instruction results. For example: cache hit or cache miss events on specific cache memories, such as L1/L2/L3 or the translation lookaside buffer (TLB). The number of possible performance events greatly outnumber the available hardware counter circuits. Therefore, it is possible to dynamically link an available hardware counter (called a performance counter) to a specific performance event. Once linked, the performance counter counts the number of events that occurred.

In processors manufactured by Intel and AMD, performance counters are realized by a set of model-specific registers. Performance monitoring mechanisms were introduced with the Pentium processor and later evolved with the introduction of the P6 family, Pentium 4, core and all later processors. In general, some performance mechanisms are architectural. These performance counters are uniformly defined for all processors, while others are non-architectural, meaning they are specific to the micro-architecture and vary between the different processor families.

Most processor models are restricted to 2-4 individual performance counters, while the different Xeon-family processors are an exception in their capability to support 9-25 performance counters, depending on the exact model.

One of the challenge’s goals is to determine if the remote machine is executing under emulation or not. Two factors are measured to determine this: the challenge result and the challenge’s elapsed execution time. The underlying assumptions are that an emulating system shall evoke different performance events as compared to a non-emulating system. Therefore, in order to calculate a result that is correct for a non-emulating system, its performance-measured environment must also be emulated by an emulator attempting to masquerade itself. Taking the previous example, where TLB side-effects are measured, such an emulator would be required to maintain an emulated TLB in order to provide the same results a normally-running system would. A task that is not entirely impossible, but would surely introduce a detectable elapsed-execution time differentiation.

Even assuming that such a feat is possible with regard to one of the side-effect modules, referencing several modules in a single challenge would necessarily amplify the elapsed-execution time differences, since these emulations are mostly orthogonal.

Full emulation of Pentium processors accrue a speed toll estimated at 2-3 orders of magnitude without maintaining side-effects, which do not contribute directly to the software flow or results.

Adding side-effect emulation would increase the execution time by an additional factor. Hence, designing challenges that utilizes a larger variety of performance measurements would not interfere with non-emulated system performance, while ensuring emulated-system differentiation.

A clear deficiency with respect to this is the low ration of available performance counters to possible performance events that can be measured. The novelty presented in this paper, designed to overcome this deficiency, is the use of "chained performance-counters". The idea is to monitor many side-effect inducing modules with a much smaller number of available performance counters, by shifting the counters from one module to the next according to a set of deterministic rules. All CPU components that generate side-effects are initialized to a known state before the challenge execution begins. When challenge execution flow reaches a determinable point, the contents of each side-effect inducing module is deterministic and repeatable regardless of our measurement i.e whether a performance counter was used to monitor its side-effects or not. It follows that a performance counter can be connected to the module to count new events. The new events will occur deterministically for the active challenge given the new determinable state.

As a result, monitoring performance events on multiple modules, using a single performance counter to measure the performance events of these modules, during several separate time intervals, will require a masquerading emulator to emulate all side-effect inducing modules to achieve the correct result.

VI. Nondeterminism

Recent generations of modern processors have seen great advancement in pipeline optimizations, to gain significant improvements in throughput. In Intel and AMD processors, driving most of the worlds laptop, desktop and server systems, these include elaborate cache structures, branch-prediction circuits and prefetcher units. As a result, a lot is going on "behind the scenes" while the processor executes the main program thread. Statistically these predictive actions have a positive effect on performance — effectively increasing overall throughput. However, side-effect event counters are affected as well, leading to seemingly non-deterministic count results.

For example, consider counting L1 data-cache hits. When a load operation causes a new cache-line to be filled it is normally not counted as a hit. However, if that cache-line happened to be previously prefetched, the said load operation will be counted as a hit. Several prefetcher logic circuits exist that account for predictive loads to cache lines.

As discussed earlier, the challenge result calculation incorporates side-effect counter values while iteratively calculating a checksum. Since prefetchers are not directly controllable, indirect means need to be employed to defeat the prefetch logic and achieve deterministic challenge results.

VII. Discussion

We have seen three advances in modern processors: virtualization extension, multiplicity of logical processors and a richer variety of performance events. Some advances improve the reliability of the authentication methods by increasing the resources required for a precise simulation. Other advances, on the other hand, allow un-authenticated remote machines to deceive the authority.

We have described how all the modern advances can be used to strengthen the authentication methods and mitigate deception attempts.

The main source of concern is the virtualization extension. While an active VMM can be easily detected on modern x86/x64 processors manufactured by Intel by issuing CPUID. The same task is much tougher on modern processors manufactured by AMD. A legitimate question to be asked is: whether, in theory, a slightly modified version of a virtualization extension can render an active VMM undetectable?

Consider an alternative implementation of the Intel virtualization extension in which the interception of the CPUID instruction can be disabled. Clearly, detection of an active virtual machine monitor is still possible and is similar to our proposal regarding AMD virtualization extension. Namely, we propose reading the register whose bits define whether a virtual machine monitor is currently active. On Intel, those bits are defined in control register 4 (CR4). Unfortunately, a virtual machine monitor can define the values of all control registers in the VMCS, rendering our detection method ineffective.

Another approach to detection of an active VMM is activation of another VMM. The second VMM is executed either directly by the hardware or with partial support of the currently active VMM. The latter form of execution, called nested virtualization, was implemented by some VMMs [22]. Since the hardware does not support nested virtualization, the VMM must react to certain virtualization extension instructions. By executing these instructions, we can differentiate between nested and regular virtualization.

We believe that it is possible, in theory, to implement a virtualization extension that will allow a virtual machine monitor to make itself undetectable. On the other hand hardware implemented nested virtualization. cannot be detected. The VMM can disable the interception of all events and schedule a delayed vm-exit. If the delay is longer than the execution time of the challenge then such a VMM cannot be detected during the challenge, but can establish full control over the processor during the delayed vm-exit.

VIII. Conclusion

We have seen that the recent advances in instruction set architecture requires the authentication authority to introduce modifications to the currently existing attestation methods. Without accommodating the problems that arise from those advancements the attestation procedure will fail on modern hardware.

The problems attesting modern hosts vary from lack of predictability, caused by logical processor multiplicity, to unreliability, caused by virtualization extensions. While every problem requires a unique solution, we note that only paradigm shifting modifications of the instruction set architecture require redesigning the authentication challenges. We have shownthat even those modifications do not prevent the establishment of authenticity of a remote machine.

References

[1] R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251374

[2] 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 Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095812

[3] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "Swatt: software-based attestation for embedded devices," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, May 2004, pp. 272–282.

[4] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 400–409. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653711

[5] D. Schellekens, B. Wyseur, and B. Preneel, "Remote attestation on legacy operating systems with trusted platform modules," *Sci. Comput. Program.*, vol. 74, no. 1-2, pp. 13–22, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2008.09.005

[6] 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 5th ACM Workshop on Wireless Security*, ser. WiSe '06. New York, NY, USA: ACM, 2006, pp. 85–94. [Online]. Available: http://doi.acm.org/10.1145/1161289.1161306

[7] Y. Yang, X. Wang, S. Zhu, and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 219–230. [Online]. Available: http://dl.acm.org/citation.cfm?id=1308172.1308237

[8] A. Seshadri, M. Luk, and A. Perrig, "Sake: Software attestation for key establishment in sensor networks," *Ad Hoc Netw.*, vol. 9, no. 6, pp. 1059–1067, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.adhoc.2010.08.011

[9] D. Ionescu, "Microsoft bans up to one million users from xbox live," PC World, Tech. Rep., 2009. [Online]. Available: http://www.pcworld.com/article/182010/xbox_users_banned.html

[10] S. consumer electronics, "Information on banned accounts and consoles," Sony consumer electronics, Tech. Rep., accessed on may 2015. [Online]. Available: https://support.us.playstation.com/app/answers/detail/a_id/1260/~/information-on-banned-accounts-and-consoles

[11] Brian, "Nintendo starting to ban pirates from online services on 3ds," Nintendo everything, Tech. Rep., 2015. [Online]. Available: http://nintendoeverything.com/nintendo-starting-to-ban-pirates-from-online-services-on-3ds

[12] Wikipedia, "An analysis of proposed attacks against genuinity tests," Tech. Rep., accessed on May 2015. [Online]. Available: http://en.wikipedia.org/wiki/Warden_(software)

[13] S. Pearson, *Trusted Computing Platforms: TCPA Technology in Context*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[14] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A trusted open platform," *Computer*, vol. 36, no. 7, pp. 55–62, Jul. 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1212691

[15] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251375.1251391

[16] 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 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 334–343. [Online]. Available: http://doi.acm.org/10.1145/1966913.1966957

[17] C. Tarnovsky, "Semiconductor security awareness today and yesterday," in *Blackhat 2010*, 2010. [Online]. Available: https://www.youtube.com/watch?v=WXX00tRKOlw

[18] ——, "Attacking tpm part two," in *Defcon 2012*, 2012. [Online]. Available: https://www.youtube.com/watch?v=Ed_9p7E4jIE

[19] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn, "Remote detection of virtual machine monitors with fuzzy benchmarking," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 3, pp. 83–92, Apr. 2008. [Online]. Available: http://doi.acm.org/10.1145/1368506.1368518

[20] I. Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3*, Intel Corporation, August 2007.

[21] "AMD64 architecture programmer's manual volume 2: System programming," http://support.amd.com/us/Processor_TechDocs/24593.pdf, 2010.

[22] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924973