nEther: In-guest Detection of Out-of-the-guest Malware Analyzers

Gábor Pék
Laboratory of Cryptography and System Security (CrySyS)
Budapest University of Technology and Economics
pek@crysys.hu

Boldizsár Bencsáth
Laboratory of Cryptography and System Security (CrySyS)
Budapest University of Technology and Economics
bencsath@crysys.hu

Levente Buttyán
Laboratory of Cryptography and System Security (CrySyS)
Budapest University of Technology and Economics
buttyan@crysys.hu

ABSTRACT

Malware analysis can be an efficient way to combat malicious code, however, miscreants are constructing heavily armoured samples in order to stymie the observation of their artefacts. Security practitioners make heavy use of various virtualization techniques to create sandboxing environments that provide a certain level of isolation between the host and the code being analysed. However, most of these are easy to be detected and evaded. The introduction of hardware assisted virtualization (Intel VT and AMD-V) made the creation of novel, *out-of-the-guest* malware analysis platforms possible. These allow for a high level of transparency by residing completely outside the guest operating system being examined, thus conventional *in-memory* detection scans are ineffective. Furthermore, such analyzers resolve the shortcomings that stem from inaccurate system emulation, in-guest timings, privileged operations and so on.

In this paper, we introduce novel approaches that make the detection of *hardware assisted* virtualization platforms and *out-of-the-guest* malware analysis frameworks possible. To demonstrate our concepts, we implemented an application framework called *nEther* that is capable of detecting the *out-of-the-guest* malware analysis framework *Ether* [6].

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEM**]: Security and Protection

General Terms

Security

Keywords

Malware Analysis, Virtualization

1. INTRODUCTION

Due to the ever-growing intelligence of malicious programs security experts are in trouble with the exact understanding of their operation. That is the reason why many steps have been taken over the years to analyse and combat them. However, miscreants are in for constructing heavily armoured pieces of malware in order to stymie the observation of their creations. Most of the in-the-wild instances use a wide range of anti-debugging and anti-unpacking techniques that are capable of detecting or evading the majority of current analyzers.

This fact stimulated the research community to come up with sandboxing tools which aim at emulating real runtime environments. The bulk of these approaches build upon pure software virtualization solutions, thus mimicking the low level instruction set of CPUs. However, numerous opcodes have different side-effects in emulated and in real platforms, thus malicious artefacts can realize that and deny execution. More refined solutions are based on hardware virtualization extensions which apply complete and accurate system emulation, so detection becomes much harder.

Such analyzers are executed in a higher privilege level software component called *Virtual Machine Monitor* (VMM) or *hypervisor* which resides completely outside the environment being analysed. This provides a high level of transparency that could be guaranteed by the rich feature set of hardware assisted virtualization. The first implementation of this approach is the *Ether* [6] malware analysis framework that is deployed over Xen and Intel VT. Later, a similar analyzer called *Azure* [12] was implemented over KVM.

In this paper we introduce new detection attacks against the *out-of-the-guest* malware analysis framework *Ether*. More specifically, we present an in-guest timing attack which was supposed to be prevented by Ether, but for various reasons, it actually does not prevent it, and an attack based on the detection of some specific settings that Ether makes in the system configuration. We also propose a technique to detect hardware virtualization platforms based on the verification of the presence of CPU specific design defects; this approach does not detect Ether itself, but identifies the hardware assisted virtual environment, which can be an indicator of the presence of a malware analysis platform such as Ether. In order to demonstrate our concepts, we implemented a scalable and flexible application framework over Windows XP called *nEther* which can practically disclose the presence of both Ether and Intel VT by executing multiple *feature tests*.

The organization of the paper is the following: Section 2 introduces the related technologies and platforms: Xen, Intel VT and Ether. In Section 3, a few possible feature tests

EUROSEC '11, 10-APR-2011, Salzburg, Austria

are defined that could be eligible for detecting Ether. The weaknesses of Ether and *out-of-the-guest* malware analyzers are discussed in Section 4 as well as the implementation of the application framework that holds the aforementioned feature tests. Related works are summarized in Section 5, while a short conclusion is given in Section 6.

2. PREREQUISITES

2.1 Introducing Xen

In Xen terminology, each virtual machine is called as *Domain (Dom)*, where there is one driver domain (*Dom0*) and can be multiple user domains (domUs) for guest OSes. The VMM is called as *hypervisor* that runs at the highest privilege level. Dom0 has privileged and direct access to all I/O devices (e.g., disks, network devices, PCI devices, etc.). It has two key roles: it administers other domains (creating, destroying, saving and so on), and makes an abstraction layer atop the hardware devices to the hypervisor and hence to virtual machines. This is feasible as the Dom0 kernel has conventional device drivers that communicate with the hardware directly.

2.2 Introducing Intel VT-x

With the introduction of *Virtual Machine Extensions* (VMX), Intel VT-x has extended the capabilities of IA-32 CPUs with two new forms of operations: *VMX root mode* and *VMX non-root mode*. The former refers to the operation form of VMM and higher privileges (ring -1), while the latter belongs to guests (ring 0) and indicates lower privileges, however, each of them supports all the conventional protection rings (0-3). There have been two transitions (*VMX transitions*) defined between root mode and non-root mode: *VMExits* refer to transitions from non-root mode to root mode, while transitions from root-mode to non-root mode are called as *VMEntries*.

The operation of the processor in VMX root mode is very similar to its normal operation, however, a set of new instructions (VMX instructions) are available and the modifications of control registers are limited. In VMX non-root mode the effects of certain sensitive instructions and events are limited, since they cause VMExits and handled by the VMM discretionally. By doing so, the VMM can retain control of the processor resources, furthermore, the guest does not know if its logical processor is running in VMX non-root mode.

2.3 Introducing Ether

Ether [6] is a malware analysis platform that utilizes Intel VT's hardware virtualization extensions, and theoretically it has no presence in the guest operating system. It uses native CPU instructions, thus does not suffer from incomplete or inaccurate system emulation such as hardware emulators do. Ether also comes with a rich feature set, thus it can monitor all the memory write attempts of the guest, trace the instructions and system calls of in-guest processes, and unpack a wide range of protected binaries. The transparency of Ether is guaranteed by the following facts [6].

- *Higher Privilege.* The malware analyzer must be at a higher privilege level than the maximal privilege level a malcode can escalate. As a consequence, all the exception handler and virtual memory mapping routines reside at a higher privilege level. In this way, isolation can be guaranteed, thus no information is leaked from the analyzer to lower privilege levels.
- *No non-privileged side effects.* All the side-effects that the analyzer causes should be accessed only through exception handler routines residing at a higher privilege level than the hostile code. Thus, any alteration of memory, CPU registers, timing, etc. should induce exceptions that can be caught only by privileged handlers.
- *Identical Instruction Execution Semantics.* Every single instruction being executed should have the same effects both in an environment where the analyzer is present (call it analysed environment) and another where it is not (call it non-analysed environment).
- *Transparent Exception Handling.* First of all, if there is no exception handler for the faulting instruction in a non-analysed environment, the privileged exception handler must guarantee the same execution semantics. Secondly, the exception handler should cause only privileged side-effects in the state of the memory and the CPU. Finally, if there is an appropriate exception handler for the faulting instruction in a non-analysed environment, then the privileged exception handler that replaces it in an analysed environment should make identical changes to the state of the memory and CPU.
- *Identical Measurement of Time.* All the timing information retrieved by a non-analysed environment should be identical to the one returned by its analysed counterpart. The analyzer should provide a falsified time to the hostile code that can be guaranteed by a *privileged logical clock* and adjusted whenever an exception (including timing request) has occurred.

Therefore, transparency is achieved in a novel way that tightens the range of possible detection attacks. Practically, Ether mitigates the risk of the following detection vectors:

- *Single-step trap.* Ether can hide its presence by intercepting the requests that directly or indirectly read the value of `TF` in the `EFLAGS` register.
- *Memory Modifications.* Every intended memory write of the observed code that generates a page fault is intercepted by Ether. Moreover, Ether modifies only shadow page tables (actual page tables used for address translation), which are invisible to the analysis target, so it cannot be detected in this way.
- *In-memory presence.* Ether cannot be detected through normal in-guest detection methods, since none of its components resides in the guest's memory.
- *CPU Registers.* All the CPU register alterations made by Ether are concealed from the guest, thus it remains indistinguishable from native (hardware assisted) environments.
- *Privileged Instruction Execution.* Ether captures certain privileged instructions and exceptions invoked by the analysis target as they induce VMExit. After handling them, these are forwarded to the guest with the same effects as in an environment without Ether.

- *Instruction Emulation.* As Ether is based on hardware assisted virtualization, all of its instructions are executed on the native CPU, thus it does not suffer from the weaknesses of hardware emulators such as QEMU.

- *Timing Attacks.* Ether mitigates timing attacks by controlling the in-guest view of internal timers, e.g., `RDTSC` instruction. It makes use of the `TSC_OFFSET` field provided by the Intel VT architecture which is added automatically to any timing request. Ether theoretically sets the `TSC_OFFSET` appropriately, thus the TSC difference between two `RDTSC` instructions should not deviate from the one measured in non-analyzed environments. That is, Ether theoretically mitigates *in-guest* timing attacks, however, the use of external timing sources may make Ether visible.

3. PROPOSED FEATURE TESTS

3.1 Timing Information

The use of timers is a well-known technique to detect the presence of traditional debuggers. The simplest way to get information about timing is the use of *internal timers* and *periodic interrupt sources* such as the *Time-Stamp Counter* (TSC), *Periodic Interrupt Timer* (PIT), *Advanced Configuration and Power Interface Timer* (ACPI timer) and the local *Advanced Programmable Interrupt Controller Timer* (APIC timer). Due to the fact that *out-of-the-guest* malware analyzers aim at providing transparency they falsify internal timers in order to fool a timing test. However, the correct manipulation of timing cannot be guaranteed in all the circumstances. A conventional detector application measures the clock cycles taken by a given operation, e.g., measures the Time-Stamp Counter difference by means of `RDTSC` (Read Time-Stamp Counter) instructions, and if this value is higher than a preset threshold the presence of an analyzer is presumable.

The operation of Ether is different from the aforementioned behaviour so as to avoid traditional detectors, however, we designed a *in-guest* timing feature test that disclosed this specific solution. The feature test builds upon a contradiction raised by clock cycle manipulation where the analyzer returns an adjusted cycle difference to hide its presence. In case of Ether, every `RDTSC` forces the guest OS, where the detector resides, to exit (VMExit) and transfer its control to the VMM to adjust the time. However, the current implementation of Ether uses a very simple logical timer that can be exploited by our feature test. More details about our implementation of this approach is presented in Section 4.3.1.

3.2 CPUID Information

The `CPUID` instruction returns processor identification and feature information depending on the content of general registers `EAX` and `ECX`. These could serve as good candidates for analyzer detectors as various modifications of the system can be retrieved. Ether states that it has neither *in-memory* nor *in-register* presence, thus it stays transparent for the guest OS. However, it does alter a few bits of information that could be returned by the correct configuration of `CPUID`. The specific implementation of this feature test is presented in section 4.3.2.

3.3 CPU Errata

CPU errata refer to the collection of design defects or errors that may induce the CPU to behave differently from the published specifications. An application containing an instruction sequence that gives rise to a CPU erratum may modify certain registers (typically MSRs), causes unexpected exceptions, generates/not generates interrupts, etc. These errata are CPU type specific, and in a lot of cases even not planed to be fixed. Since these design deficiencies are not implemented by emulators and virtualization extensions (neither software nor hardware), efficient tools are given for anyone who are about to detect the presence of a non-native runtime environment. While we cannot detect the presence of Ether itself with this approach, it can be used to reveal the hardware assisted virtualized environment. See section 4.3.2 for implementation.

4. IMPLEMENTATION

4.1 Application Framework

First of all, an application framework is implemented in C++ that provides all the preconditions that are required by the feature tests. Since detectors require several privileged instructions to be executed, the framework builds upon a Windows kernel driver as Figure 1 depicts it.

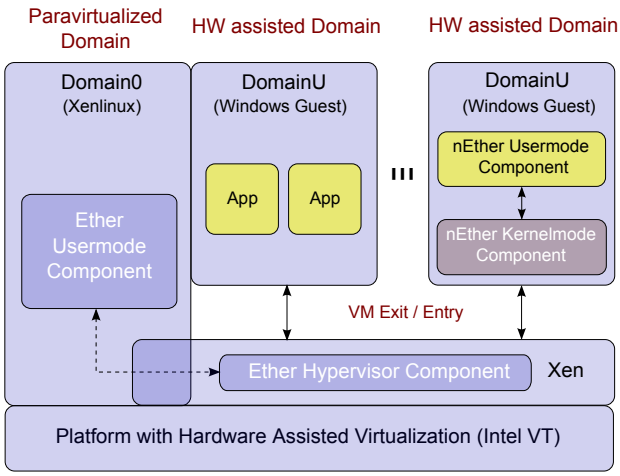

Figure 1: System Overview

Basically there are two types of feature tests: those that do not use the kernel driver, and those that do. The operation of the former is straightforward, thus only the latter is detailed in the following. Once the kernel driver had been loaded, the messaging with it can be initiated. Therefore, whenever a detection test requires a privileged instruction, it uses the `DeviceIOControl` WinAPI function which is capable of notifying the corresponding kernel driver about a communication attempt. The implemented device driver called *DetectorDriver.sys* plays an important role in the detection mechanism as the access and manipulation of privileged data structures, registers, memory locations are feasible only from kernel mode.

4.2 Execution and Verification

The implemented feature tests had to be executed on several reliable environments to make our measurements ac-

ceptable. Since Ether is implemented for an Intel VT CPU with 64-bit support, we also had to choose such a platform. Intel VT has to be enabled explicitly in BIOS in order to operate. The latest compilation of Ether runs on Debian Lenny, and is tested only with Windows XP SP2 guest operating system. That is the reason why we set up the same host environment (Domain 0) with Debian Lenny (Linux kernel 2.6.26) and Xen support. After patching Xen with Ether only the guest operating system had to be installed. Thus, feature tests are executed on 3 different guest operating systems: Windows XP over Ether, Windows XP over only Xen and native Windows XP.

4.3 Implementation of Feature Tests

4.3.1 In-guest Timing

We propose an *in-guest* timing feature test that can detect Ether by exploiting a practical weakness. To understand how our feature test works we introduce the operation of RDTSC instruction in VMX non-root mode. Its value depends on the "RDTSC exiting", "use TSC offsetting" VM-execution control fields and the TSC_OFFSET value. VM-execution control fields allow the VMM to control the operation of certain instructions like the aforementioned RDTSC, while TSC_OFFSET is a special field of the Intel VT architecture that is added to the value of the returned Time-Stamp Counter in defined cases. The RDTSC instruction in VMX non-root mode works in one of the following three options. The first option is that if both the "RDTSC exiting" and "use TSC offsetting" control fields are zero, an in-guest RDTSC can operate normally, thus it returns the 64-bit Time-Stamp Counter value. If the "RDTSC offsetting" field is zero and the "use TSC offsetting" is 1, then it responds with the sum of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC_OFFSET field. The last option is that if the "RDTSC exiting" control field is 1, then it causes a VMExit, which can be intercepted by the VMM to adjust the Time-Stamp Counter value. This is the technique that Ether applies, however, the 64-bit TSC value returned to the guest is adjusted in a deterministic manner as the code snippet (pasted from ether_lenny.patch source file) below demonstrates it.

```
+/* maintain a monotonic fake TSC counter */
+u64 ether_get_faketime(struct vcpu *v)
+{
+ return ++v->domain->arch.hvm_domain.
                .ether_controls.faketime;
+}
```

As the above mentioned code snippet is only responsible for increasing the timer monotonically, the authors of Ether proposed to exactly adjust the logical timer by the use of the TSC_OFFSET field of the Intel VT architecture. Significant to emphasize that this routine is called only at the time when the guest exited with an RDTSC, thus nowhere else the TSC is being incremented. In theory, the task of TSC_OFFSET would be to adjust the returned Time-Stamp Counter value conforming to a native system. Contrary to the original paper [6], the TSC_OFFSET field is not updated by the Ether implementation and does not contain the difference value between the execution time of Ether's exception handler and that of the native handler routine. The "TSC offsetting" VM-execution control field is being disabled while "RDTSC exiting" is being enabled in the minimum configuration, as the following source code taken from the Ether-patched Xen source file vmcs.c shows.

```
void vmx_init_vmcs_config(void)
{
  ...
    min = (CPU_BASED_HLT_EXITING |
           CPU_BASED_INVDPG_EXITING |
           CPU_BASED_MWAIT_EXITING |
           CPU_BASED_MOV_DR_EXITING |
           CPU_BASED_ACTIVATE_IO_BITMAP |
           //CPU_BASED_USE_TSC_OFFSETING |
           CPU_BASED_RDTSC_EXITING);

    _vmx_cpu_based_exec_control =
    adjust_vmx_controls( min, opt,
    MSR_IA32_VMX_PROCBASED_CTLS_MSR);
 ...
}
```

Due to the fact that TSC_OFFSET is neglected and the TSC is increased by one for one in-guest RDTSC call, the TSC difference between two RDTSC instructions is 1. More precisely, this is the case when the code being executed in Ether is not analysed. According to our observations whenever a code is under analysis (e.g., instruction traced) there are CPU time slots for other guest processes to invoke additional RDTSCs. Due to that fact there is a varying TSC difference (∼9-171) value between any two RDTSCs of the analysed code we introduce below. This difference is still so tiny that a general timing attack found in various malware samples (measuring the TSC difference before and after the execution of an instruction) will not be successful. However, this approach still does not enable Ether to stay completely transparent as this kind of operation deviates much from the normal operation of RDTSC.

In the following, we give a practical feature test against Ether which a bit different from a conventional *in-guest* timing attack as it builds upon the fact that each instruction of an instruction sequence increments the TSC with a predefined value depending on the CPU family. For example, in case of Core 2 Duo processors the TSC is incremented at a constant rate by either the maximum core-clock to bus-clock ratio of the processor or by the maximum resolved boot-time frequency [4]. Supposing that each value is bigger than 1, a program that estimates the extent of increment can be used as a feature test. The code sample below initializes a NOP loop that increments the TSC at a constant rate in each iteration, thus the resulting difference between the initial and the final execution of RDTSC should be at least equal to the length of the loop (2000).

```
      mov ecx, 2000
      cpuid
      rdtsc
      xchg ebx, eax
lb:   nop
      loop lb
      cpuid
      rdtsc
      sub eax, ebx
      cmp eax, 2000
      jbe det
```

As RDTSC is a non-serializing instruction, it may be executed after the subsequent or before the previous instructions if their execution takes a while. That is, a serializing instruction, e.g., CPUID, is put before reading the counter.

Furthermore, the loop must contain a non-privileged instruction which does not cause a VMExit so as to prevent its execution from being manipulated in the VMM. Note that in the code sample we examine only the lower 32 bits of `RDTSC` as the faked TSC has never exceeded this interval for our tests. Since `RDTSC` has been discussed above the behaviour of it can be used in other two modes. If it works normally, the presence of a debugger can be detected with conventional timing methods. If it returns the sum of the `IA32_TIME_STAMP_COUNTER` MSR and the value of the `TSC_OFFSET` field the case is the same as with normal operation since the VMM cannot manipulate the TSC value conforming to the length of the loop. Thus, these two other options could not help to adjust the *in-guest* timer. However, in our opinion the implementation of correct faked timing could have been solved with some efforts, thus it is not a theoretical problem.

4.3.2 CPUID Bits

This feature test checks a few bits returned by a `CPUID` that are specific to Ether. The Time-Stamp Counter bit (`TSC`) returned by the `CPUID` instruction on the first leaf (`EAX`=1) shows whether `RDTSC` is supported, including the `CR4.TSD` bit which restricts the execution of the `RDTSC` instruction. By default the `TSC` bit is set on both native and virtualized operating systems. However, Ether explicitly disables it, thus breaching a preset condition that no modifications are made on the state of the guest. The code sample below first executes a `CPUID` and checks whether the VMX bit is set by Xen, thus the environment is hardware assisted virtualized. Finally, the state of the `TSC` bit is evaluated and the result is returned (`res=0` initially) accordingly.

```
  mov eax, 1
  cpuid
  bt ecx, 5  // Checking the VMX bit
  jc exit
             // Xen is detected
  bt edx, 4  // Checking TSC bit if set
  jc exit
  mov res, 1
exit:
```

The current implementation of Ether disables Physical Address Extensions (PAE) and Page Size Extensions (PSE) in order to make memory writes easier as it has been explicitly stated by the authors. At the same time, these changes give rise to another bit of information that can be useful in the detection.

4.3.3 CPU Errata

These type of feature tests could not reveal the presence of Ether, but could reveal the hardware-assisted-virtualized runtime environment. Note that the detection of system monitors through CPU errata has already been discussed for CPU identification in [11], however, here only the QEMU hardware emulator was tested with this technique. Most of the errata require kernel mode instructions, thus must have been implemented in a Windows driver component. On the other hand, these CPU deficiencies or bugs are strongly bound to CPU models, thus the feature tests can be effective only on certain CPU families (e.g., Intel Core 2 Solo). Since our test environment is built atop a machine with Intel Core 2 Duo E6600 CPU the following erratum exploits a vulnerability of the Core 2 Duo family [5].

AH4 Erratum.

The AH4 Erratum states that "*VERW/VERR/LSL/LAR Instructions May Unexpectedly Update the Last Exception Record (LER) MSR* " and there is no planned fix for it. The problem is that the `LER MSR` is updated in certain cases for unknown reasons, if the resultant value of the `Zero Flag` (`ZF` flag of `EFLAGS` register) equals zero after the execution of the instructions above. The *Last Exception Record MSR* comprises two registers called `MSR_LER_FROM_LIP` and `MSR_LER_TO_LIP` situating at register addresses `0x1dd` and `0x1de`, respectively. The former is the abbreviation of *Last Exception Record From Linear Instruction Pointer* which points to the last branch instruction (conditional/ unconditional jumps, call, etc.) that had been taken by the processor before the last exception occurred or the last interrupt was handled. The latter refers to *Last Exception Record To Linear Instruction Pointer* which stores the address of the target of the last branch instruction that had been executed by the processor before the last exception occurred or the last interrupt was handled.

The "*Verify a Segment for Reading or Writing*" instructions verify if a code or data segment is readable/writeable from the current privilege level (CPL) according to [4]. This erratum requires kernel mode operation as it reads privileged resources (LER MSR), thus it is implemented as a part of our device driver. By providing the cleared `AX` and `CX` registers for `VERR` and `VERW` instructions the resultant `ZF` flag is zero for sure as invalid segment pointers (NULL) are given to the source operands. Last, but not least the value of the *Last Exception Record To Liner IP* MSR is read with the privileged `__readmsr(MSR address)` Visual C++ command at register address `0x1de`.

```
__asm{
  xor eax, eax
  xor ecx, ecx
  verr cx
  verw ax
}
ret = __readmsr(0x1de);
```

The concept behind this erratum is that an *out-of-the-guest* analyzer might unintendedly modify the Last Exception Record MSR as it contains information about the last generated exception. Due to the fact that under Ether any memory write attempt of the analysis target induces a page fault which updates the LER value (since the last branch instruction varies as well), this erratum might influence the number of changes. However, the access of LER MSR can be restricted by the MSR bitmap of the CPU [4], thus the execution of the erratum does not affect the LER MSR of the host. A processor erratum is a design fault so its existence is unintended. Consequently, hardware assisted virtualization solutions (e.g., Xen) will not implement them in the exposed virtual CPUs of guests because it would take too much effort and make no sense to mimic unexpected system behaviours, however, a higher level of transparency could be provided. According to the figures, a guest environment supported by Intel VT-x and Xen does not suffer from this weakness. Table 1 demonstrates the execution of this feature test. First of all, the CPU erratum was executed 100, 1000, 10000 and 100000 times under the corresponding environments. As the results show, the erratum has occurred only in native environment, thus it is an evident detector for hardware assisted virtualization.

	Number of updates		
N	*Native*	*Xen*	*Xen with Ether*
100	59	0	0
1000	650	0	0
10000	4232	0	0
100000	20870	0	0

Table 1: The number of updates

5. RELATED WORK

The fingerprinting of virtualized environments is discussed in previous work [3] where a remote network-based fingerprinting method was introduced. In order to hide the fingerprints of system emulators more stealthy debuggers were built [11]. There are several various malware analysis platforms that make use of virtualization extensions, however, the detection of them is not challenging as being in-guest approaches (e.g., VAMPiRE [14]) or running at the same privilege level (e.g., Cobra [14]) as the hostile code. Furthermore, there are also *out-of-the-guest* systems (e.g., TTAnalyze [1]) that build upon QEMU [2] for which there are thousands of detectors, red-pills [10]. Sandboxing environments such as Norman Sandbox [9] or CWSandbox [8] use either Windows API hooking or Windows API virtualization that are detectable as operating at the same privilege level as the observed code [7] does. The work of *Monirul Sharif et al.* is an in-guest VM monitoring tool called SIM [13] which allegedly guarantees the same security level as an *out-of-the-guest* approach. Beside Ether, there are other supposedly transparent *out-of-the-guest* dynamic malware analysis platforms such as the so-called *Azure* [12]. Note that centerpiece of this work is Ether, however, it involves general considerations as well that might be applicable for other *out-of-the-guest* analyzers.

6. CONCLUSIONS

In this paper we introduced novel *in-guest* methods for virtualized environments which are capable of detecting the presence of the *out-of-the-guest* malware analysis platform Ether. Moreover, a feature test could explicitly detect the Intel VT-x hardware virtualized environment. In order to demonstrate our approaches we implemented an application framework. Ether formalized transparency with high accuracy, however, we highlighted a few weaknesses practically.

In the future, we aim at updating the theoretical model and practical implementation of Ether so as to successfully evade current and some possible future attacks. As a consequence we are about to examine various aspects that highlight the theoretical weaknesses of *out-of-the-guest* approaches. One such a problem relates to timing, where we identified three different classes: Firstly, the user interface interaction is extremely slow under the guest if a code is being analysed by Ether, e.g., the move of the mouse arrow is intermittent in that case. Secondly, there can be other relative timing sources found which could be used as reference points. One such an idea is the fluctuation of CPU fan RPM (Rotation Per Minute) as according to our observations, this RPM value is changing periodically under a native environment. Finally, hardware related timing sources including DMA data transfer, reading speed of PCI bus can also be potential candidates. However, even more detection methods can be figured out, thus in our opinion the provision of perfect transparency cannot be guaranteed neither theoretically nor practically.

7. ACKNOWLEDGEMENTS

This work was carried out within the scientific program called "Development of quality-oriented and cooperative R+D+I strategy and functional model at BME", which is part of the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

8. REFERENCES

[1] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware, 2006.

[2] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[3] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*, pages 177–186, Anchorage, Alaska, USA, June 2008.

[4] I. Corporation. Intel®64 and IA-32 Architectures Software Developer's Manual, June 2009.

[5] I. Corporation. Intel®Core™2 Duo Processor for Intel®Centrino®Duo Processor Technology Specification Update. `http://download.intel.com/design/mobile/SPECUPDT/31407918.pdf`, September 2010.

[6] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[7] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, 2006.

[8] Malware Analysis System, CWSandbox :: Behaviour-based Malware Analysis. `http://mwanalysis.org/`.

[9] Norman Sandbox Whitepaper. `http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf`, 2003.

[10] R. Paleari, L. Martignoni, G. Fresi, and R. D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *In Proceedings of the USENIX Workshop on Offensive Technologies (WOOT*, 2009.

[11] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Information Security Conference (ISC 2007)*, Oct 2007.

[12] P. Royal. Alternative Medicine: The Malware Analyst's Blue Pill. In *Black Hat USA*, Aug 2008.

[13] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 477–487, New York, NY, USA, 2009. ACM.

[14] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *21st Annual Computer Security Applications Conference, 2005*, pages 381–392, 2005.