

HYPERDBG DEBUGGER

A DEBUGGER DESIGNED FOR ANALYZING, FUZZING AND REVERSING

VERSION 0.1.0.0

VM-exit Transparency in HyperDbg

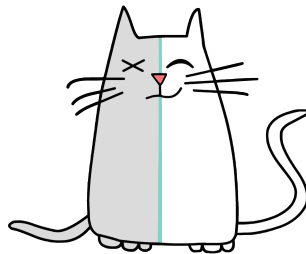
Website

<https://hyperdbg.org>

Research

<https://research.hyperdbg.org>

July 25, 2022



HyperDbg Progress: Transparency

In this short article we describe the progress on developing the *transparency* feature of the HYPERDBG. We have developed a statistical-based mitigation for HYPERDBG to be immunized against timing side-channel attacks targeting sub-OS intercepting entities.

1 Timing Measurement to Detect HyperDbg

The existence of a third party sub-*ring 0* programs and debuggers are often detected by a set of timing measurement instructions in the OS, forcing a *vm-exit* procedure to detect any abnormality.

1.1 VM-exit

Generally *VM-exits* are transitions from VMX non-root operation to VMX-root operation. Virtual machine control structure (VMCS) is a data structure in memory that exists exactly once per VM (or more precisely, one per each logical CPU) while the VMM manages it. With every change in the execution context between different VMs, the VMCS is restored for the current VM, defining the state of the VM's virtual processor and VMM control Guest software using VMCS. The VMCS consists of six logical groups.

- **Guest-state area:** Processor state saved into the guest state area on *VM-exits* and loaded on *VM-entries*.
- **Host-state area:** Processor state loaded from the host state area on *VM-exits*.
- **VM-execution control fields:** Fields controlling processor operation in VMX non-root operation.
- **VM-exits control fields:** Fields that control *VM-exits*.
- **VM-entry control fields:** Fields that control *VM-entries*.
- **VM-exits information fields:** Read-only fields to receive information on VM exits and describing the cause and the nature of the *VM-exits*.

The *VM-exits* and its corresponding cause are significantly crucial when handling timing measurements if transparency is considered.

1.2 Time Difference due to VM-exit

In the presence of the HYPERDBG, multiple instructions cause unconditional *VM-exit*, which reveals the presence of a lower-level inspector in the system. Particularly, detectors employ *CPUID* between the *RDTSCL* to measure the elapsed time, as shown in the following listing.

```
2  rdtscp    ; get the current time clock of processor
3  ...      ; save the rdtsc results somewhere (e.g registers)
4  cpuid     ;Execute a serialization instruction (forcing VM-exit)
5  ...
6  rdtscp    ; Compute the core clock timing again in order to see how many
7           ; clocks are spent
```

Listing 1: The timing measurement code by forcing VM-exit

Moreover, there are other instructions such as *GETSEC*, *INVD*, *XSETB*, *INVEPT*, *XSETBV*, *INVVPID*, *INVVPID*, *HLT*, *INVLPG*, *RDPMC* I/O IN/OUT, *WBINVD* as well as all VMX instructions cause *VM-exits*. Also, Exceptions, NMIs, MSRs Read/Write, EPT Violations and Monitor Trap Flags (MTF) would cause *VM-exits* too.

2 Statistical Analysis of Elapsed Time for VM-exit

In this section, we statistically analyze the POC timing measurements used in commodity detector software as well as computer anti-cheat programs. Specifically, the basic **CPUID** along with pair of **RDTSCP** is used in our simple test case.

2.1 Setup

We have used a Skylake i7-6820HQ processor to execute our tests under latest Windows 10 with/without HYPERDBG activated. Note that the method should work for all Intel CPUs architectures with different specifications since the statistical analysis in the transparency function is executed on any system running HYPERDBG, yielding its own specific timing results (which should be verified in practice).

2.2 Modeling in the time in normal condition (Without HyperDbg)

We have executed the code in the Listing 1 10,000 times, and measured the time in normal condition with HYPERDBG deactivated in the system.

Note that the measurement process took under 1 minute in our test-bed.

The initial timing measurement data was not reliable due to the existence of outlier samples. These false samples are measured due to unreliable execution of some CPU instructions, handling numerous processes simultaneously in the timing procedure, and interrupts. Based, on the model of the data in accordance to the *Variance*, *Maximum* and *Minimum* of samples, we have utilized the *Grubbs outlier method*[3] and *Median Absolute Deviation(MAD)*[6]. As could be predicted, the data follows a Gaussian Distribution when the Probability Distribution Function is plotted as in Figure 1.

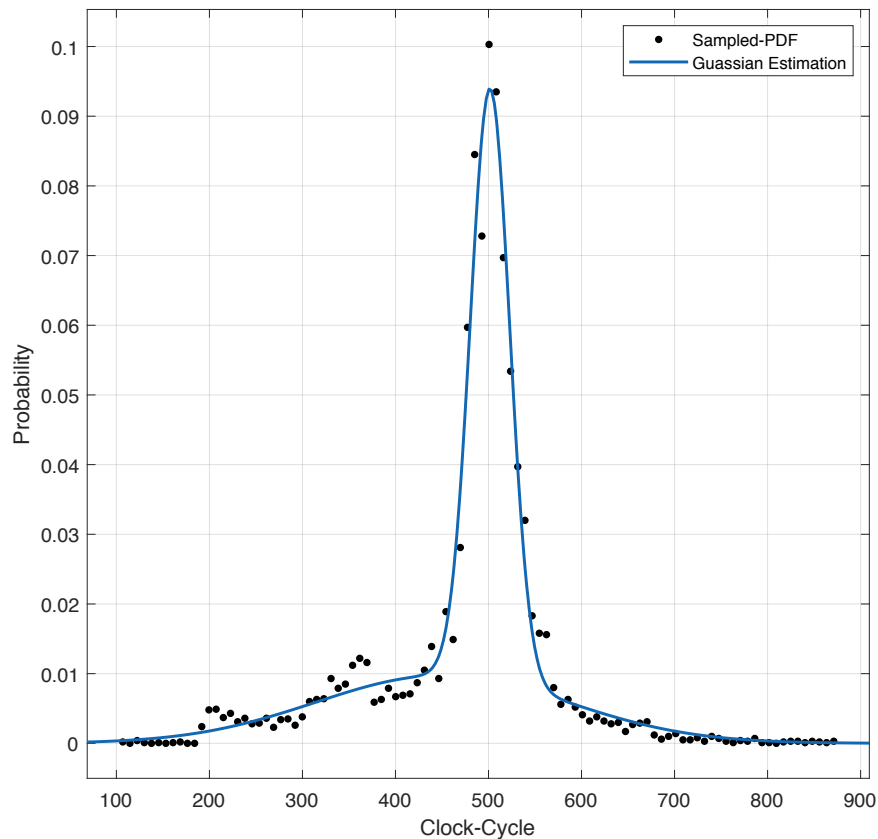


Figure 1: PDF distribution and sampled data of timing measurement without activated HyperDbg

The discrete interval of the bins in order to construct the PDF is set to 100 in our test-case. The sampled data follow a two-term Gaussian form in Equation 2.

$$p(x) = a_1 e^{-\frac{(x-b_1)^2}{2c_1^2}} + a_2 e^{-\frac{(x-b_2)^2}{2c_2^2}} \quad (1)$$

With the help of values of a, b, c , a simple fitting algorithm could be used to reveal the Average (μ) and Variance (σ^2) of a single Gaussian Distribution.

These values then are used in a simple Gaussian Random Generator Algorithm such as Marsaglia Polar Method [5], to generate a timing sample. The usage of this sample will be explained in the methodology.

2.3 Modeling in the time in the presence of HyperDbg

In the presence of HYPERDBG, the **CPUID** instruction will force a *VM-exit* as discussed, and a chain of instructions in the procedure will occur. In this scenario, we have evaluated the timing measurements for the same 10,000 times executing the code in Listing 1. After removing the outlier data based on the similar description in the previous section, the distribution of the data set is extracted as shown in Figure 2.

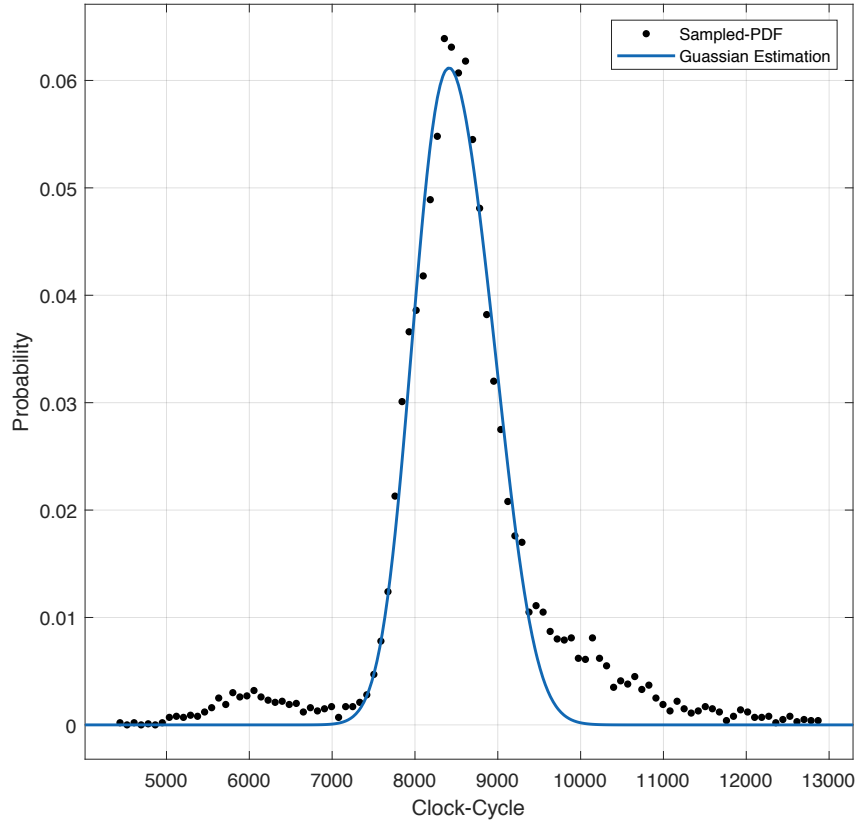


Figure 2: PDF distribution and sampled data of timing measurement with activated HyperDbg

3 Proposed methods for Transparency

The general approach to hide the presence of the low-level debugger here is to transparent the timing leakage caused by the operations executed in HYPERDBG, breaking the routine of the OS and CPU.

We propose two different methods to make the HYPERDBG transparent. The first approach is to change the CPU time stamps carefully (i.e., **IA32_TIME_STAMP_COUNTER**) in the kernel when an analyzer

software or a simple user attempts to measure the elapsed time. By a statistical profiling process, *VM-exit* timing along with other processes in the measurement procedure could be estimated for each computer in accordance with the processor model. Then, this estimated time could be used to update the time stamps to hide the instructions that occurred in the HYPERDBG.

The second method presented here, relies on a systematic approach to emulate the timing instructions in Intel processors. In such approach every time **RD TSC** and **RD TSCP** instructions are executed, a *VM-exit* is forced and a special *transparency* function is invoked to hide the timing leakage due to the extra instructions in the HYPERDBG.

Below, we have discussed each of these approaches in detail.

3.1 Changing Timestamps

In this approach, by the use of the statistical methodology described in the previous section, an accurate timing profiles are constructed and employed in transparency functions. An overview of the approach is shown in Figure 3.

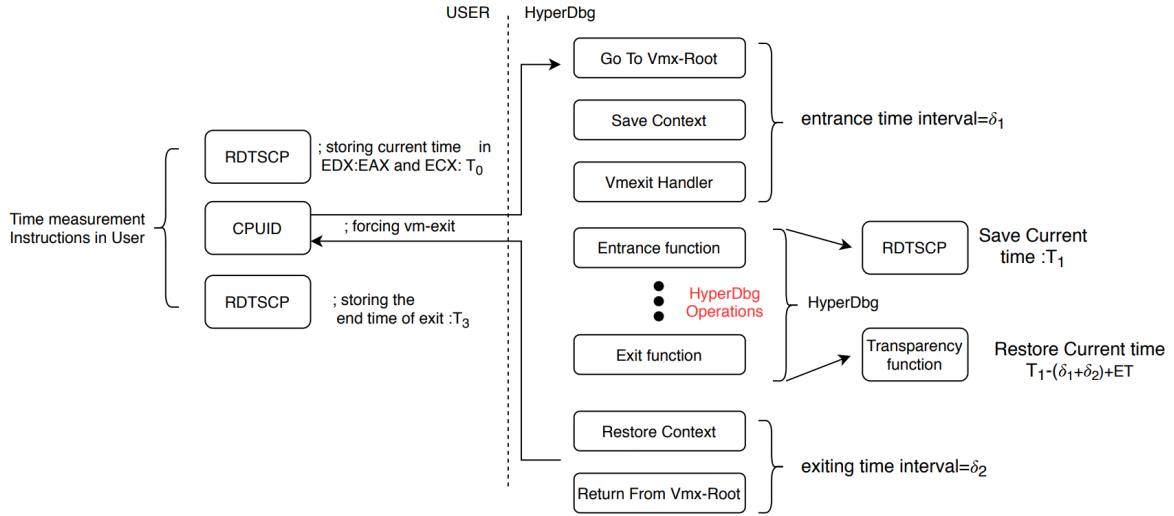


Figure 3: Transparency of HYPERDBG by changing IA32_TIME_STAMP_COUNTER

Assuming the target time measuring instruction set in Listing 1, according to the Figure 3, in the presence of HYPERDBG, **CPUID**, will force out a *VM-exit*. The *VM-exit*, consist of a *entrance* and *exiting* phases as depicted in the figure. *entrance* and *exiting* time intervals represented by δ_1 and δ_2 respectively, are measured iteratively in order to derive statistical characteristics. These timing are also follow a Gaussian Distribution (or two terms Gaussian Dis.). Note that the values of δ_1 and δ_2 are measured initially before the hiding process itself. To ensure the validity of these measurement for the HYPERDBG, a special function in the user-mode is executed by the debugger, storing the timing stamps (T_0) using *Volatile* registers (e.g. EDX, ECX). Moreover, additional hash validation is checked for each time measurement.

After the profiling phase, the hiding process is executed by HYPERDBG as shown in the Figure 3. When an analyzer software is activated to detect any low-level interception, the Entrance Function stores the time by executing **RD TSCP** (T_1), after *VM-exit* entrance. Then, any arbitrary functions in HYPERDBG are executed. At the end of the operations in the hypervisor, a Transparency function is called. Here, the IA32_TIME_STAMP_COUNTER value is replaced with the following value.

$$Time_Stamp = T_1 - (GRG(\delta_1) + GRG(\delta_2)) - GRG(Norm) \quad (2)$$

The GRG is the Gaussian Random Generated number by the use of the Marsaglia Polar Method [5] activated with Standard Deviation (σ) and Mean (μ) values captured in the initial statistical test-cases. $GRG(Norm)$ represents the Gaussian estimated elapsed clock-cycles for the timing instructions when HYPERDBG is not activated in normal condition.

Taking into account, the procedure explained above, anytime a third-party detecting software performs a timing analysis to detect HYPERDBG, the system automatically immune itself by normalizing the timestamp used to reveal the time.

3.2 Emulating RDTSC/RDTSCP

In this method, the timing instructions(**RDTSC** and **RDTSCP**) are entirely emulated by the HYPERDBG, whenever used in a monitoring software, so the analyzer fails to detect any timing abnormality in the system operation. This is essentially done by a chain of events and a procedure that will be described in detail in this section.

The overview of the state diagram of emulation process is shown in the Figure 4.

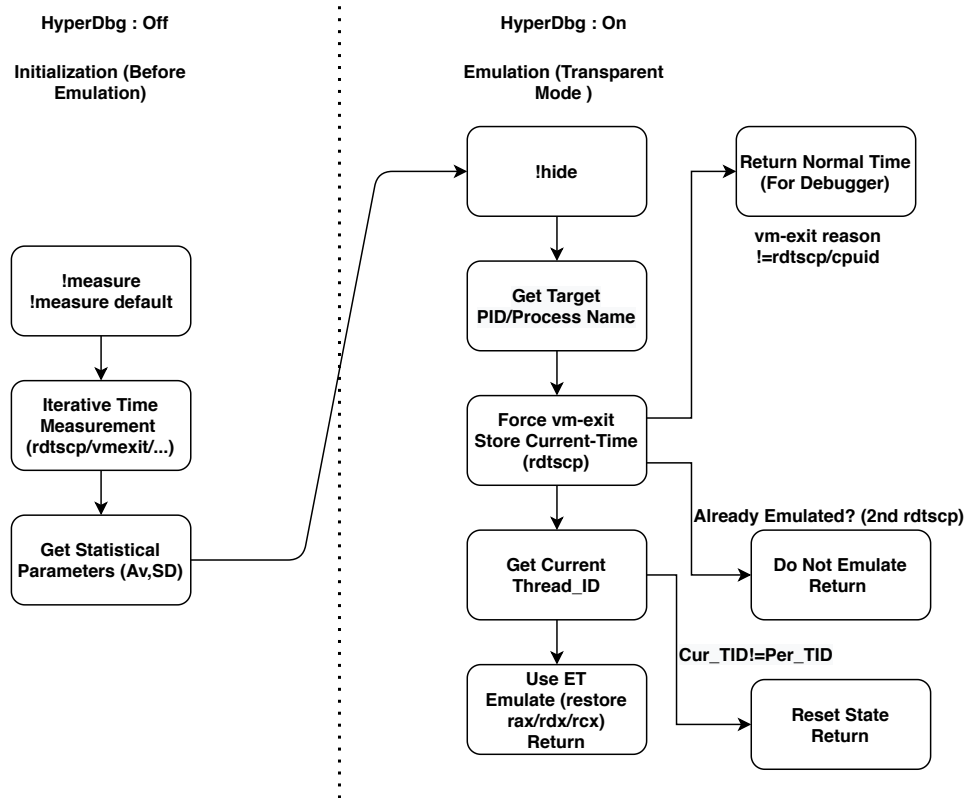


Figure 4: State Diagram Process of *rdtsc/rdtscp* emulation by HYPERDBG

According to Figure 4, this procedure requires an initialization procedure where HYPERDBG is deactivated on the normal operating system. This initiation procedure comprises a statistical analysis of time measurement. The analysis is carried out by **!measure** [2] command design in the HYPERDBG. The main goal here is to derive the Gaussian Parameters of timing intervals for different measurement scenarios that analyzing software could perform. By iteratively measuring the time interval normally, operating system required *Standard Deviation* and *Mean* values for these scenarios which are captured in accordance with the unique characteristics of the system in hand. Alternatively one can use **!measure default** command for the use in virtual machine. This command uses default parameters that have been stored earlier in the system. Although this transparency method works for virtual machines, the VM hypervisor performs other verification methods rather than timing measurement to ensure the integrity to detect the interceptions.

Following the successful measurement of timing instructions, the HYPERDBG is activated and ready to be initiated in the transparent-mode. The process is executed by the command **!hide** [1]. The user is required to pass the targeted PID or process name, which HYPERDBG is supposed to be transparent to it. This could also be carried out by handing over a list of analyzer software to HYPERDBG. Afterwards, the system is ready to emulate and hide any **RDTSC** / **RDTSCP** instructions.

Confining the emulation process to a target PID/process is essential since a global emulation of timing instructions would most likely disturb the primary functionalities of the system. Our experiments show a disturbance in Screen driver as well as audio output performance when a global emulation is implemented.

In order to execute HYPERDBG functionalities for transparency it is required to enforce a *VM-exit* any time timing instructions are called. This enforcement could be activated by setting the 12th bit of *Primary Processor-Based VM Execution Control*, a field in the *VMCS* as described in Intel's user manual (Figure 5 [4]).

Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls

Bit Position(s)	Name	Description
2	Interrupt-window exiting	If this control is 1, a VM exit occurs at the beginning of any instruction if RFLAGS.IF = 1 and there are no other blocking of interrupts (see Section 24.4.2).
3	Use TSC offsetting	This control determines whether executions of RDTSC, executions of RDTSCP, and executions of RDMSR that read from the IA32_TIME_STAMP_COUNTER MSR return a value modified by the TSC offset field (see Section 24.6.5 and Section 25.3).
7	HLT exiting	This control determines whether executions of HLT cause VM exits.
9	INVLPG exiting	This determines whether executions of INVLPG cause VM exits.
10	MWAIT exiting	This control determines whether executions of MWAIT cause VM exits.
11	RDPNC exiting	This control determines whether executions of RDPNC cause VM exits.
12	RDTSC exiting	This control determines whether executions of RDTSC and RDTSCP cause VM exits.
15	CR3-load exiting	In conjunction with the CR3-target controls (see Section 24.6.7), this control determines whether executions of MOV to CR3 cause VM exits. See Section 25.1.3. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.

Figure 5: Primary Processor-Based VM Execution Control [4]

In the case of a *VM-exit*, the program first checks the *VM-exit* reason. If the reason is not due to a *RDTSC/RDTSCP/CPUID*, the previously stored timestamp for the *RDTSC* at the beginning of the *VM-exit* is returned. This is a crucial action since the debugger itself could have halted the system, and many instructions are executed during this interval. This ensures the transparency of the intentional/unintentional activities of the debugging process. Moreover, if the system has already emulated a *RDTSC* as the first timing instruction, it needs to leave the second instruction unemulated and be executed normally.

If the conditions are met, the final constraint is to check the *Thread ID* for the measuring application and verify that the same *thread* is performing a measurement. In the case of any *thread* switch, a context switch has occurred, and every counter and state should be reset as well as all the timing counters. Finally, the exact reason and timing measurement scenario is classified (whatever it is caused by a *CPUID* or a simple *RDTSCP/RDTSCP*) and using the parameters derived by the *!measure* command, A Gaussian Random Number as a normal time interval is generated by a Marsaglia Polar Algorithm [5]. Then the values are replaced in *RAX* and *RDX* in the case of *RDTSC* and *RCX* as well in the case of *RDTSCP*.

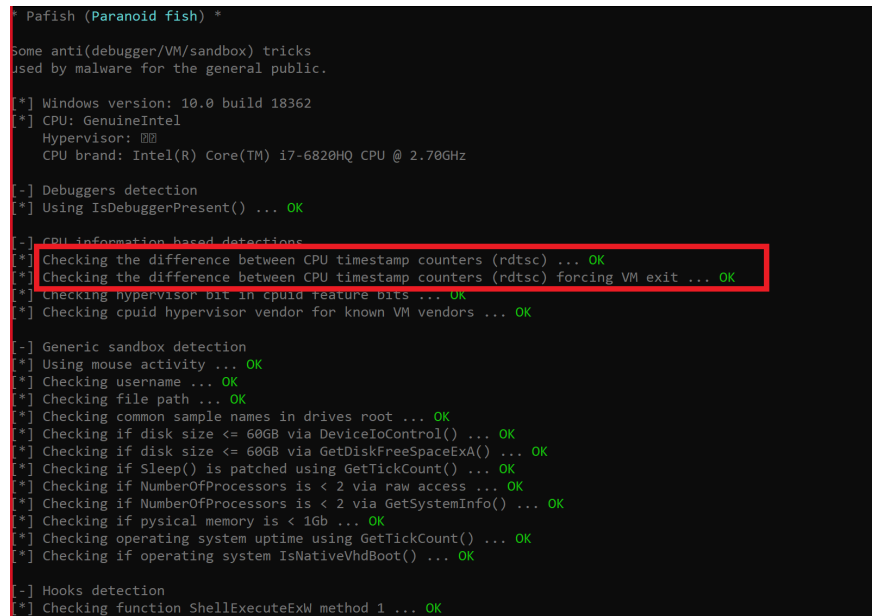
The problem with the second method is that there are different methods to detect *RDTSCP/RDTSCP* emulation by measuring the elapsed time between two or more *RDTSCP/RDTSCP*. However, the first method is immune to these detections because we are not emulating *RDTSCP/RDTSCP* and the execution of these commands represents the real system clock. In order to solve this problem, we saved the results of the first timestamp counter, and in the second *RDTSCP/RDTSCP*, we added our measurements to the emulated counter. It is precisely like *RDTSC+CPUID+RDTSC* method, but in the initial measurements, we measure the time between *RDTSC+RDTSC* and use it in our emulation.

4 Results

There are multiple debugger detectors and analyzers, both commercial and open-source, to be tested for the proposed transparency method here. However, for our initial evaluation, we have tested the implemented methods on the well-known *pafish* [7] software.

Not surprisingly, the first method, which includes updating *IA32_TIME_STAMP_COUNTER*, interferes with the system's primary functions, making the screen flickering during our experiments. Hence, We were unable to execute our experiments for the first method successfully. We are still working on the method to resolve the side effects caused by changing the system timestamps.

Nevertheless, the second method (emulation) was able to successfully pass *pafish* analyser with 100% success rate after executing *!measure* and handling the *!hide* command with the process name of *pafish*. Figure 6 shows the transparent active HYPERDBG in a physical system, where *pafish* fails to detect any abnormal timing leakage.



```

Pafish (Paranoid fish) *
some anti(debugger/VM/sandbox) tricks
used by malware for the general public.

[*] Windows version: 10.0 build 18362
[*] CPU: GenuineIntel
    Hypervisor: 
    CPU brand: Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz

[-] Debuggers detection
[*] Using IsDebuggerPresent() ... OK

[-] CPU information based detections
[*] Checking the difference between CPU timestamp counters (rdtsc) ... OK
[*] Checking the difference between CPU timestamp counters (rdtsc) forcing VM exit ... OK
[*] Checking hypervisor bit in cpuid feature bits ... OK
[*] Checking cpuid hypervisor vendor for known VM vendors ... OK

[-] Generic sandbox detection
[*] Using mouse activity ... OK
[*] Checking username ... OK
[*] Checking file path ... OK
[*] Checking common sample names in drives root ... OK
[*] Checking if disk size <= 60GB via DeviceIoControl() ... OK
[*] Checking if disk size <= 60GB via GetDiskFreeSpaceExA() ... OK
[*] Checking if Sleep() is patched using GetTickCount() ... OK
[*] Checking if NumberOfProcessors is < 2 via raw access ... OK
[*] Checking if NumberOfProcessors is < 2 via GetSystemInfo() ... OK
[*] Checking if physical memory is < 1Gb ... OK
[*] Checking operating system uptime using GetTickCount() ... OK
[*] Checking if operating system IsNativeVhdBoot() ... OK

[-] Hooks detection
[*] Checking function ShellExecuteExW method 1 ... OK

```

Figure 6: Successful Transparency Functionalities of HYPERDBG over *pafish*

5 Extensions and Improvements

We are improving our scenarios where an analyzer executes more complex time measurements by executing multiple *RDTSCLP* or executing pre-defined dummy instructions in the timing interval.

Moreover, we are working on the issues regarding the experiments for the first method. Also, The experiments should be carried out on different analyzing software to showcase the reliability of the proposed method.

References

- [1] *!hide (enable transparent-mode)*. <https://docs.hyperdbg.com/commands/extension-commands/hide>.
- [2] *!measure (measuring and providing details for transparent-mode)*. <https://docs.hyperdbg.com/commands/extension-commands/measure>.
- [3] *Grubbs's test for outliers*. https://en.wikipedia.org/wiki/Grubbs%27s_test_for_outliers. Accessed: 2020-08-01.
- [4] Intel Intel. "and IA-32 architectures software developer's manual". In: *Volume 3A: System Programming Guide, Part 1*.64 (64), p. 64.
- [5] *Marsaglia polar method*. https://en.wikipedia.org/wiki/Marsaglia_polar_method. Accessed: 2020-08-01.
- [6] *Median absolute deviation*. https://en.wikipedia.org/wiki/Median_absolute_deviation. Accessed: 2020-08-01.
- [7] *pafish*. <https://github.com/a0rtega/pafish>. Accessed: 2020-08-13.