HyperDbg: Reinventing Hardware-Assisted Debugging

Mohammad Sina Karvandi[1,2], MohammadHossein Gholamrezaei[1,2], Saleh Khalaj Monfared[1,2], Suorush Medi[2], Behrooz Abbassi[2], Ali Amini[2], Reza Mortazavi[3], Saeid Gorgin[1], Dara Rahmati[1], Michael Schwarz[4]

[1] Institute For Research in Fundamental Sciences (IPM), [2] HyperDbg Organization, [3] Damghan University, [4] CISPA Helmholtz Center for Information Security

ABSTRACT

Software analysis, debugging, and reverse engineering have a crucial impact in today's software industry. Efficient and stealthy debuggers are especially relevant for malware analysis. However, existing debugging platforms fail to address a transparent, effective, and high-performance low-level debugger due to their detectable fingerprints, complexity, and implementation restrictions.

In this paper, we present HyperDbg, a new hypervisor-assisted debugger for high-performance and stealthy debugging of user and kernel applications. To accomplish this, HyperDbg relies on state-of-the-art hardware features available in today's CPUs, such as VT-x and extended page tables. In contrast to other widely used existing debuggers, we design HyperDbg using a custom hypervisor, making it independent of OS functionality or API. We propose hardware-based instruction-level emulation and OS-level API hooking via extended page tables to increase the stealthiness. Our results of the dynamic analysis of 10,853 malware samples show that HyperDbg's stealthiness allows debugging on average 22% and 26% more samples than *WinDbg* and *x64dbg*, respectively. Moreover, in contrast to existing debuggers, HyperDbg is not detected by any of the 13 tested packers and protectors. We improve the performance over other debuggers by deploying a VMX-compatible script engine, eliminating unnecessary context switches. Our experiment on three concrete debugging scenarios shows that compared to *WinDbg* as the only kernel debugger, HyperDbg performs step-in, conditional breaks, and syscall recording, 2.98x, 1319x, and 2018x faster, respectively. We finally show real-world applications, such as a 0-day analysis, structure reconstruction for reverse engineering, software performance analysis, and code-coverage analysis.

KEYWORDS

Hypervisor, Debugging, Kernel-debugger, Fuzzing, Malware-analysis

Reference Format:
Mohammad Sina Karvandi[1,2], MohammadHossein Gholamrezaei[1,2], Saleh Khalaj Monfared[1,2], Suorush Medi[2], Behrooz Abbassi[2], Ali Amini[2], Reza Mortazavi[3], Saeid Gorgin[1], Dara Rahmati[1], Michael Schwarz[4]. 2022. **HyperDbg: Reinventing Hardware-Assisted Debugging.** ***(HyperDbg).*** **, 18 pages.**

1 INTRODUCTION

Debuggers are an essential element in software development and analysis that are actively employed by computer engineers to improve efficiency, detect security flaws, and fix bugs in software programs. Additionally, debuggers are also utilized as a valuable tool for software reverse engineering and malware analysis purposes. There has been a series of commercial and open-source debugging software offering convenient features to address such needs [16, 19, 71, 84, 93]. Given the outstanding growth in the sophistication and complexity of evasion and obfuscation methods, it is necessary to facilitate powerful debuggers to analyze, detect, and understand malware.

Modern binary executables, armed with packing [96], evasion [27], and hardware-assisted techniques [48, 69], employ a series of methods that involve anti-virtualization [4], anti-emulation[54], as well as side effects and footprint detection [1] (e.g., call to specific OS APIs) to impede debugging. Despite many valuable efforts for development of transparent and effective analysis methods in the community [23, 25, 68, 77, 95, 97], currently available debugging tools struggle to encounter modern protected programs and malware. These tools lack elaborate kernel-side components to offer deep scrutiny for reverse-engineering purposes. A comprehensive analysis of 4 million malware samples shows that 88% are equipped with anti-reversing, and 81% with anti-debugging or virtualization techniques [8]. Utilizing OS APIs [9] or leveraging ring-0 options [34] leads to artifacts and leakages that high-privilege malware can detect.

All these complications have recently attracted the attention of researchers to integrate the debugging infrastructure deeper into the hardware. As a result, solutions based on bare metal [49, 91, 97], hypervisor level (VT-x) [23, 29, 52, 92], System Management Mode (SMM) [97], or even Intel Memory Management Engine (MME) [28] are used to minimize the leakage of the debugger's presence. This increases the transparency of the debugger and thus its stealthiness. While these lower-level realization of debugging mechanisms increase the transparency surface, they suffer from huge performance degradation. Although sub-kernel deployment [98] of debugging, monitoring and software analysis tools can offer a powerful platform for such use cases such as analyzing evasive malware, previously-proposed sub-kernel debuggers fail to provide rich debugging functionality as they have been either discontinued [17, 29], developed for pure academic purposes [29], or have not been through thorough development and testing required for dealing with real-world applications and scenarios [98]. Moreover, the availability of the source code for such tools is still known to be a requirement in the community.

In this paper, we propose HyperDbg, a hypervisor-based (ring -1) debugger designed to use modern hardware technologies to provide new features to the reverse-engineering community. It operates on top of Windows by virtualizing an already running system using Intel VT-x. As a primary goal, HyperDbg strives to be

HyperDbg, May, 2022, ArXiv
2022 HyperDbg

as stealthy and OS-independent as possible. HyperDbg avoids using any operating-system APIs and software debugging mechanisms. Instead, it extensively uses processor features such as Second Layer Page Table, i.e., Extended Page Tables (EPT), to monitor both the kernel and the user executions.

Avoiding OS-based debugging APIs increases the transparency against classic anti-debugging methods. Moreover, by directly relying on hardware feature, HyperDbg is invisible for time-delta methods that detect the presence of hypervisors, e.g., by detecting the overhead of traps into the hypervisor [64, 72]. Such hardware-enabled features also allows HyperDbg to offer various state-of-the-art functions such as hidden hooks, which are as fast as current inline hooks but also offer stealth debugging. HyperDbg supports *Hardware Debug Registers* simulation to break on read and write accesses to a specific location while remaining entirely invisible to both the OS kernel and the programs. Moreover, such hardware-assisted features make it possible for HyperDbg to eliminate all limitations previously imposed by *Hardware Debug Registers* in size and count [97]. We evaluate the transparency by extensive evaluation against anti-debugging, anti-virtualization, anti-hypervisor methods, and packer software. HyperDbg was not detected by any of the 13 tested packers and protectors. No other existing debugger achieves this level of stealthiness, with debuggers being detected on average by 44% of packers and protectors, with no debugger detected by less than 3. We demonstrate the applicability of transparent debugging on 10,853 malware samples. Our results show that HyperDbg successfully analyzes 22% and 26% more malware samples compared to *WinDbg* and *x64dbg* respectively. We also describe an existing 0-day vulnerability in Windows 10 kernel successfully analyzed by HyperDbg's transparent mode, rediscovered during our experiments.

For high-performance debugging, HyperDbg uses a VMX-root-compatible script engine that executes the entire debugging functionality in the kernel mode, enabling complex debugging functionality. Our script engine eliminates any user to kernel-mode interaction, making any OS-level API obsolete while providing a huge debugging performance. We evaluate the improved debugging performance in three concrete debugging scenarios: stepping, conditional breaks, and syscall recording. Compared to the state-of-the-art debugger *WinDbg*, HyperDbg is 2.98, 1319, and 2018 times faster, respectively.

We show that the unique design of HyperDbg enables use cases beyond classical debugging scenarios. We describe how the proposed debugger enables transparent debugging of I/O devices, analyses performance of software, and provides means for code coverage usable for (kernel) fuzzing. Finally, our analysis of a Windows 10 0-day in a kernel-mode bootkit malware shows that HyperDbg is mature enough for real-world malware analysis.

Contributions. The contributions of this paper are as follows.

(1) We present HyperDbg, a hypervisor-assisted debugger specialized for deep software analysis, reverse engineering, and fuzzing with a focus on stealthiness.
(2) We introduce a VMX-root-compatible script engine within HyperDbg that is orders of magnitude faster than state-of-the-art debuggers for common tasks.
(3) We demonstrate transparent debugging on 10,853 malware samples, showing that HyperDbg can analyze 22%-26% more malware samples than state-of-the-art debuggers.
(4) We describe multiple applications of HyperDbg, such as large-scale and fast malware analysis including a Windows 0-day analysis, code coverage in fuzzing, debugging of I/O devices, and software-performance measurements.

Availability. HyperDbg is fully open source and is available to foster the security research and software engineering: https://github.com/HyperDbg/HyperDbg.

Outline. The remainder of this paper is organized as follows. In Section 2, we provide required background information. Section 3 presents the design, and Section 4 the architecture of HyperDbg. Section 5.2 introduces the script engine, and Section 6 the transparency-mode of HyperDbg. Section 7 provides the transparency and performance evaluations. Section 8 describes additional use cases. Section 9 discusses related work, and Section 10 concludes the paper.

2 TECHNICAL BACKGROUND

In this section, we survey the technical background knowledge to describe the design of the proposed debugger. We briefly review the structure and features of modern debuggers, hypervisors, and the main hardware capabilities provided by Intel, on top of which HyperDbg is implemented.

2.1 Modern Debuggers

Debugging is fundamentally defined as the process of examination and analysis of a software program to understand or locate the unsatisfying code snippets in terms of functionality, performance, or security flaw [2, 51]. To address the desired functionalities, a debugger should facilitate multiple mechanisms. Stepping through the source code or assembly, memory inspection and modification, as well as breakpoint definition are vital features in commodity debuggers. From the reverse engineering and malware analysis perspective, debuggers generally fall into two categories of user-mode and kernel-mode debuggers [34]. User-mode debuggers provide the basic functionality to analyze a user-mode process. They are simply implemented and easy to use. User-mode debuggers give a convenient and isolated environment for the user. *x64dbg* [93], *Ollydbg* [71], and Immunity Debugger [43] are well-known examples of user-mode debuggers. Kernel-level debuggers run in kernel mode, which grants them higher privileges in terms of register and memory access during the program's execution. *WinDbg* [16] and *GDB* [19] are famous examples of kernel debuggers that are widely used for reverse engineering and malware analysis [1]. With advances in malware evasion techniques [4], researchers have been showing interest towards virtualization, simulation, and hardware-assisted debugging methods [53] that can offer a more transparent environment for code analysis and low-level modification of the execution flow [68, 97].

2.2 Hypervisor

A hypervisor (also known as a virtual machine monitor or VMM) is a software that makes virtualization possible by virtually sharing the resources, such as memory and processor [13, 18, 74]. It abstracts guest machines and the operating system they run from the actual hardware and runs virtual machines (VMs). Generally, hypervisors can be divided into the two following categories:

Type 1 - Hypervisor. Type 1 (also known as “bare metal”) hypervisor runs directly on the top of the hardware, with no operating system running below it [33]. In these hypervisors, the VMM is responsible for allocating and scheduling system resources to guest OSs. Well-known examples of this class of hypervisors include VMware ESXi and Xen.

Type 2 - Hypervisor. Type 2 hypervisor (also known as “hosted VMM”) run as an application in a host operating system and performs I/O operations on behalf of the guest OS. As such, after the guest OS issues an I/O request, it gets trapped by the host OS and sent to a device driver, and the completed I/O request is again routed back to the guest OS via the host OS [24, 33]. Examples of this category are VMware Workstation, VMware Player, Microsoft Hyper-V, Oracle VirtualBox, and Parallels Desktop.

2.3 Instruction Set Architecture (ISA) Extensions

In this section, we briefly describe Intel VT-x, Intel EPT, and Intel TSX ISA extensions employed in the proposed hypervisor-level debugger. Note that HyperDbg in its current format only supports Intel processors and is built based on Intel technologies and terminology. However, similar hardware features exist both for AMD and ARM processors that can be exploited likewise. Further description is provided in Appendix F.

Intel Virtualization Technology (VT-x). Intel VT-x (formerly known as Vanderpool) is the hardware virtualization technology provided by Intel for IA-32 processors to simplify virtualization and increase the performance of VMMs [66]. VT-x introduces new data structures and instructions to the ISA [30] and enables processors to act as if there were several independent processors to allow multiple operating systems to run simultaneously on the same machine.

Intel Extended Page Table (EPT). Intel VT-x technology comes with a hardware-assisted Memory Management Unit (MMU) and the implementation of Second Level Address Translation (SLAT), known as Extended Page Table (EPT). By translating the Guest Physical Address (GPA) to Host Physical Address (HPA) on the CPU level [87], EPT eliminates the overhead associated with software-managed shadow page tables [42]. In Intel’s design, each CPU core can use a separate EPT Table, which allows for multiple independent accesses from different OSs concurrently.

Intel Transactional Synchronization Extensions (TSX). Intel TSX is the product name for two x86 instruction set extensions, called Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) [45, 83]. In this paper, by using the term Intel TSX, we refer to RTM specifically. RTM is an extension that adds instructions to ISA that allow declaring hardware transactions. Instructions within a transaction appear atomic, i.e., either all succeed or the architectural state is rolled back to before the transaction. Such a rollback happens, e.g., if an interrupt occurs during a transaction.

2.4 Terminology

As our implementation here is based on Intel processors, we describe the low-level design of our system uses Intel Terminologies. Appendix C gives a complete descriptions of the used terms featured by Intel which are also briefly tabulated in Table 4.

3 HIGH-LEVEL OVERVIEW

This section provides a brief high-level description of the design of HyperDbg and its building blocks. Here, we describe how the proposed debugging functionalities are implemented by a high-level abstraction and propose three debugging operations modes.

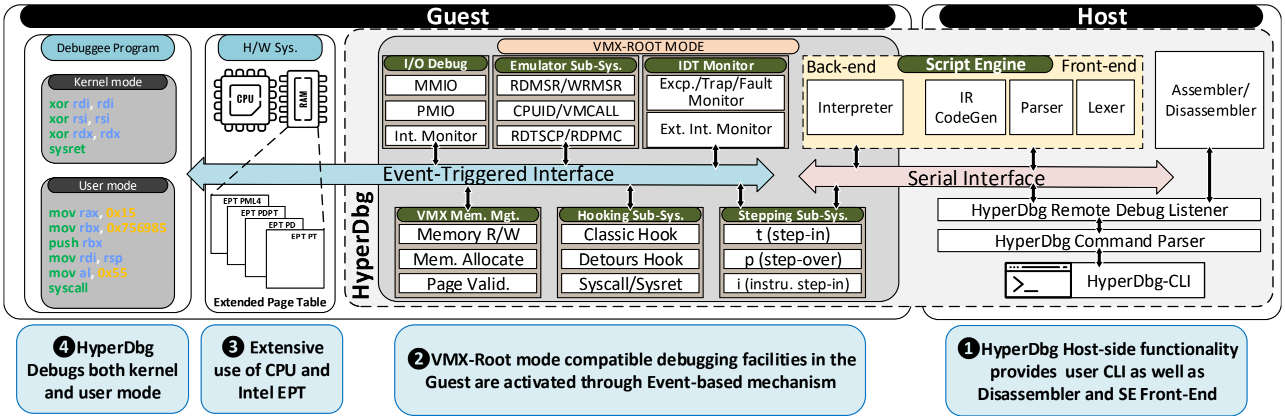
3.1 High-level Debugging Flow

On the high level, like other debuggers, HyperDbg is design to performs a level of analysis within a target system referred as the *Guest*. The source debugging instructions are usually sent from an external system known as the debugger *Host*. Figure 1, illustrates a high-level overview of HyperDbg’s sub-systems and execution flow. As shown, the debugger lies as an end-to-end framework, connecting the guest and the host systems by a communication interface (e.g., Serial). While the core building blocks are all deployed within the hypervisor level on the guest side, the host side provides a CLI interface with the user and deploys an assembler/dissembler as well as a front-end engine for the debugging functionalities. Multiple debugging sub-systems are deployed in the VMX-root mode of the guest system, which directly utilizes hardware features (e.g., EPT) for their functionality. As shown in Figure 1, the debugging commands are taken by the host where are (dis)assembled and parsed through the script engine in ❶. Then, the commands are sent via a communication channel to the guest. These commands are interpreted on the script engine’s back-end at the guest’s hypervisor level. Based on the requested debugging routine, any user or kernel-mode debuggee program code can be targeted on the guest side with a direct access to the execution flow as indicated in ❹. The sequence of the commands and functionalities are executed based on an event-triggered routine (Section 3.2) according to each sub-system as depicted in ❷. Finally, The sub-system functionalities utilize correspondent hardware-based features (e.g., EPT) to execute its operation in ❸. We describe the deployment of each sub-system in detail in Section 4. Note that in the practical debugging procedure, bi-directional communication is required between the user at the host and debuggee code at the guest side. However, as shown in the figure, with the use of the script engine, HyperDbg is able to confine the communication in an automated routine within the guest kernel mode if necessary.

3.2 Event-Triggered Interface

To facilitate the debugging routines, we control the usage of the underlying functions and building blocks by an abstracted concept referred as an *Event* in HyperDbg. Subsequently, we define *Conditions* and *Actions* that are used in the sub-system procedures for debugging.

3.2.1 Events. An *Event* is the occurrence of an incident that is of interest to the debugger. This comprises a wide range of activities ranging from a specific system call (*Syscall*) that the debugger is set



```
xor rdi, rdi
xor rsi, rsi
xor rdx, rdx
sysret
```

```
mov rax, 0x15
mov rbx, 0x756985
push rbx
mov rdi, rsp
mov al, 0x55
syscall
```

Figure 1: High-level overview of HyperDbg's sub-systems and execution flow

to monitor, to access to a particular memory address. HyperDbg can be configured to perform arbitrarily defined actions upon the occurrence of each event. A list of the supported events provided by HyperDbg is presented in Table 3 in the Appendix A.

3.2.2 *Actions.* Upon having an event triggered, HyperDbg can evoke specific functionalities known as actions. HyperDbg provides three types of action: *Break*, *Script*, and *Custom Codes*. The Break action is the conventional feature of classic debuggers where all processing cores are paused until the debugger's further permission. The Script action allows viewing and modifying parameters, registers, and memory contents without breaking into the debugger. It also permits creating logs and running codes in the kernel space. The Custom Codes action provides the ability to run custom assembly codes whenever a specific event is triggered.

3.2.3 *Conditions.* Conditions are specific circumstances that can be defined by the user in form of logical expressions to constrain the execution of an event. This, in turn, allows for the definition of conditional events where an event is triggered only upon evaluation of an expression to true.

3.3 Operating Modes

Based on different applicability, HyperDbg provides two modes of operation described as follows.

3.3.1 *VMI Mode.* Virtual Machine Introspection (VMI) Mode is presented for regular user application debugging and kernel-mode local debugging. Although it offers a conventional debugging experience by providing access to all HyperDbg features (including debugging, halting, and stepping user-mode applications) in an out-of-the-box fashion, kernel-mode breaking to the debugger and stepping are limited. VMI mode also allows scripts and custom codes in both user-mode and kernel-mode for local or remote debugging.

3.3.2 *Debugger Mode.* Debugger Mode is a powerful operating mode that allows for connecting to the kernel and halting the system to step-in and step-over through the kernel and user instructions. Here, debugging connectivity is carried out with a serial cable or a virtual serial device.

3.3.3 *Transparent Mode.* Both modes can be used in Transparent Mode, which offers stealth debugging by attempting to conceal HyperDbg's presence on timing and microarchitectural levels. While this does not guarantee 100% transparency, it makes it substantially more challenging for the anti-debugging and anti-hypervisor methods to detect the debugger. The presenting transparency methodology is described in Section 6 and is thoroughly evaluated in Section 7.1.

4 BACK-END ARCHITECTURE

This section explores the architectural design of HyperDbg on a sub-system level. We describe the challenges and shortcomings of the existing methods and debuggers for each sub-system. Then, by describing the underlying detailed implementation of the core sub-systems, we propose HyperDbg's approach to address each of these challenges.

4.1 Stepping Subsystem

The stepping mechanism is an essential capability of a debugger. In this section, we investigate the stepping mechanism used in conventional debuggers and their shortcomings with regards to their capability in delivering a true line-by-line stepping procedure. In the following, we discuss the solutions offered in HyperDbg as a *VMX-root* mode debugger to provide different stepping mechanisms and address these issues.

4.1.1 *Step-in.* Step-in offers the conventional step functionality available in commodity debuggers (e.g., *WinDbg* [16], *GDB* [19]) by setting the *RFLAGS* trap flag to make the system stop after execution of a single instruction. This allows the debugger to read/modify the content of the registers and the memory by following a trap flag in the kernel.

Challenge. Conventional stepping mechanisms cannot guarantee a line-by-line stepping procedure as all other CPU cores and processes may execute their routines, and interrupts can drastically alter a program's execution flow.

Figure 2a shows an example of the step-in where the execution flow is disrupted by a #DB exception interruption. A naive solution would mask all external interrupts by unsetting the Interrupt Flag in *RFLAGS*. However, intercepting/preventing the interrupts can easily break the OS semantics and lead to Blue Screen Of Death (BSOD) (e.g., queuing self-DPC interrupts when IRQL is not properly adjusted). HyperDbg introduces the *instrumental step-in* to provide a guaranteed stepping mechanism in debugging routine.

Approach. Considering the shortcomings of the conventional Step-in mechanism, HyperDbg introduces an instrumentation Step-in mechanism by employing the Monitor Trap Flag (MTF); a feature that works similar to RFLAGS's Trap Flag (TF) but appears transparent to the guest. Moreover, by utilizing Non-Maskable Interrupts (NMIs) to ensure the execution in a single core while other cores are halted. This method entirely overcomes the disruptions by inevitable interrupts.

4.1.2 Instrumentation Step-in. To the best of the authors' knowledge, HyperDbg is the first debugger to address the issue by presenting a guaranteed stepping method. According to Figure 2c, after executing the target instruction, a VM-exit is triggered (as an MTF has been previously set). Doing so, guarantees that only the succeeding instruction is executed in the debugging guest. To do so, HyperDbg continues at only one core and disables interrupts on the same core (ignoring external interrupts by setting external-interrupts exiting bit in VMCS) to offer a fine-grained stepping. This method provides the user with the unique feature to instrument routines from user-mode to kernel-mode and kernel-mode to user-mode that is not possible though other kernel debugger (*WinDbg*). As an example, whenever the user-mode executes a SYSCALL instruction, HyperDbg allows the user to follow the instructions directly into the kernel and execute the next instruction in the kernel-mode (SYSCALL handler). Similarly, if a page-fault occurs in the middle of a user-mode application, the debugger is moved into the kernel-mode's page-fault handler. On the other hand, kernel-mode to user-mode migration is also handled by HyperDbg (e.g., executing a *SYSRET* or *IRET* returns the debugger to user-mode from kernel-mode).

4.1.3 Step-over. The step-over mechanism in HyperDbg is very similar to conventional Step-in, except for the call instruction where the debugger sends the length of the call instruction to the debuggee, and instead of setting the Trap flag, it sets a Hardware Debug Register to the instruction after the call. Therefore, when the call is finished, the Hardware Debug Register is triggered, and the debugger is notified about the next instruction. Since other threads/cores might also trigger the Hardware Debug Register (as all the threads/cores are continued through the stepping), HyperDbg ignores such #DBs from other Thread IDs/Process IDs and re-sets the debug register until reaching the correct execution context and target thread that is supposed to trigger the Hardware Debug Register. Figure 2b shows the overview of the step-over stepping mechanism in HyperDbg, where upon inspection of a call instruction, a debug breakpoint exception (#DB) is thrown for the next instruction.

4.2 Hooking Subsystem

Hooking in the context of debugging is the act of intercepting an arbitrary event (e.g. execution of a breakpoint on a particular address), running specific commands, and turning the execution flow back to the conventional routine at the entry point of the event.

Challenge. Existing hooking systems in commodity debuggers implement direct memory access, which a user-mode software can easily check and detect. The integrity of memory can effortlessly be verified as well. This leaves the possibility of debugging detection for evasive malware. Moreover, *Hardware Debug Registers* used to record memory content in debugging process are fixed in number and size, limiting hooking performance.

4.2.1 SYSCALL and SYSRET Hooks. HyperDbg implements hooking functionality by evoking an undefined opcode exception (#UD) (by unsetting the SCE bit in the Extended Feature Enable Register, i.e., *IA32_EFER*) and checking for the originating cause of the exception. The user can execute any arbitrary script and set hooks for any arbitrary system-call through the OS (*SYSCALL*) or any return of the execution flow from a system-call (*SYSRET*). During a user-to-kernel or kernel-to-user emulation, the debugger can monitor, execute or modify the system context before the actual execution of the instructions. HyperDbg provides the following approach for its novel hooking capabilities.

Approach. HyperDbg allows the user to monitor and manipulate memory accesses while remaining transparent by providing two EPT hooking mechanisms that reveal an unmodified version of the target page to the application. This methodology delivers an entirely transparent memory hook via EPT. Furthermore, we emulate Debug Registers to increase address traceability surpassing the previous limitations.

4.2.2 EPT Hidden Hook. We propose *Hidden Breakpoints* (Classic EPT Hooks) which add a #BP (0xcc) to the target machine's memory to cause a trap upon an attempt from the guest to execute the target memory address. Another way is to utilize *Detours-Style Hooks (Inline EPT Hooks)*, which change the execution path by jumping to the patched instructions and returning the execution flow to the regular routine after the callback.

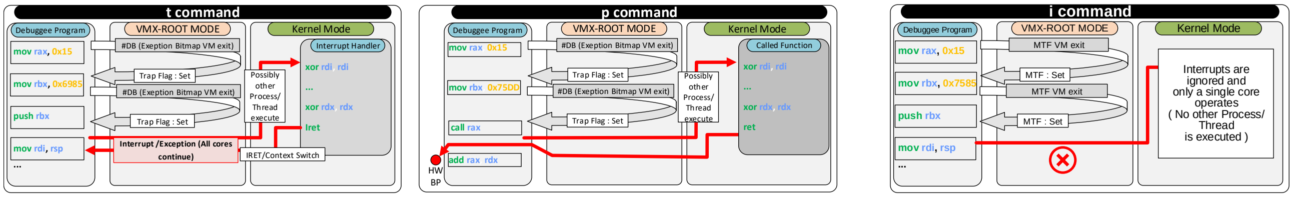
While the latter approach has some flexibility constraints (e.g., limitations with the usage of script engine, the range of hookable addresses, number of hooks in a page table), avoiding the costly VM-exit operation makes for a substantially faster hooking mechanism.

4.2.3 Limitless Simulating of Debug Register (monitor). EPT hooking also allows for monitoring any read/write to any range of addresses by causing an event trigger to emulate *Hardware Debug Registers* capability while eliminating its limitations on the number and lengths of trackable addresses [11].

4.3 Memory Access in VMX-root Mode

Implementation of safe memory access is one of the challenging parts of designing a hypervisor-level debugger, as there are many scenarios that can lead to system halt or an exception (e.g., access to paged-out [39] pages in the VMX-root [46], and access to user-space memory from the VMX-root mode) that cannot be addressed using readily available primitive instructions (e.g., *mov*).

Challenge. Safe memory access through VMX-level is extremely complicated as it is often handled by the OS. This often results in performance overheads and footprints in conventional current



(a) The *t command* Stepping mechanism in HyperDbg

(b) The *p command* Stepping over mechanism in HyperDbg

(c) The *i command* Instrumentation Stepping Approach in HyperDbg

Figure 2: HyperDbg Stepping Commands

debuggers. However, safe and efficient memory access is necessary for many use cases such as malware analysis.

Approach. We propose a series of methodologies to address the complications of VMX memory management, described as follows.

4.3.1 Discovering Page-table Entries. The conventional method in HyperDbg to detect a valid page is checking for the presence of a valid page-table entry (with set present bit) for its target address. This method requires traversing through the page tables to carry out the discovery process. As an alternative method, we make use of Intel Transactional Synchronization Extensions (TSX). TSX suppresses exceptions/faults without any switch between user/kernel modes nor getting involved with exception handlers. This ability is leveraged in HyperDbg to check for the validity of a page by checking the successful execution of a transaction involving the target address. A similar approach has been used by Schwarz et al. [81] to check if an address in SGX is mapped. This method can be carried out using only a few instructions (Listing 1); however, as not all processors support this capability, HyperDbg automatically checks for the processor's support of this feature and switches to the former method if necessary. Our experiments show that a TSX-based page discovery for user-mode debugging is roughly three orders of magnitude faster since normal traversing requires the requests to be forwarded to the user for validity check. However, in kernel-mode applications, the method incurs a 40% slow-down due to the domination of cycles introduced by RTM routines.

```
 ; Use Intel TSX to suppress any
; page-fault in VMX-root mode
 XBEGIN   $+xxx    ; End of TSX
 MOV RAX, Dword PTR:[RCX]
; Access the target memory address,
 XEND     ; End of TSX
 MOV RAX, 1
 JMP Return
 MOV RAX, 0
Return :
 RETN     ; Return the result
```

Listing 1: Using Intel TSX to detect address validity.

4.3.2 Retrieving a Page by Injecting Page Fault (#PF). Upon absence of a page, HyperDbg injects a page-fault to the debuggee (by configuring the *CR2* register to the target virtual address) to request the VMX non-root, to bring the page back from the hard disk to the RAM when it is resumed. While this method is not applicable in some scenarios (e.g., in DISPATCH_LEVEL IRQL level as paging is not available), it can be useful in many others (e.g., upon execution of a SYSCALL or SYSRET where the system is guaranteed to be in PASSIVE_LEVEL).

4.3.3 VMX-root Mode Compatible Message Tracing. Sending a message from VMX-root mode to VMX non-root mode is a challenging part of hypervisor design due to various limitations of accessing paged-pool buffers in VMX-root mode. Notably, most NT functions are not *ANY IRQL* compatible, as they might access buffers that reside in paged pool memory. To send commands and messages from VMX-root mode to the user-mode application or the debugger, HyperDbg provides a custom VMX-root mode compatible message tracing mechanism. This mechanism operates on the non-paged pool, and its memory is visible in VMX-root mode. By deploying specialized messaging buffers, we ensure that the messages are only sent when the paging process is safely accessible on the kernel-mode. The details of this mechanism is thoroughly discussed in [78].

4.3.4 Reading and Writing Memory. Due to the various safety considerations surrounding making direct access to a user-space address from VMX-root mode, HyperDbg is designed not to access the memory directly but to use a virtual addressing method to reserve a Page Table Entry (PTE) and map the desired user-mode physical address to a kernel-mode virtual address to enable safe memory read/write access. Furthermore, the write-enable bit in the PTE eliminates the check for the writability of the target address.

4.3.5 Pre-allocated Pools. Given that most of HyperDbg's routines operate in VMX-root mode, HyperDbg makes use of pre-allocated pools to provide a mechanism for addressing the conventionally impossible [46] issue of allocating memory in the VMX-root mode. These pools (when divided into 4KB granularity) provide the resources necessary for EPT hooks. HyperDbg's memory manager routines periodically check for any deallocation/replacement of memory pools needed in VMX root mode and performs them when the debugee is in VMX non-root mode.

5 FRONT-END ARCHITECTURE

In the following section, we explore the intermediatory components of HyperDbg's connecting back-end VMX-root mode sub-systems with the host machine as well as the user-interface functionality. Specifically, We describe guest-host communication and the kernel-level script engine. Although the core functionality of the proposed script engine operates on the guest side's VMX-root, we regard all non-VMX-root sub-modules in our framework as front-end here.

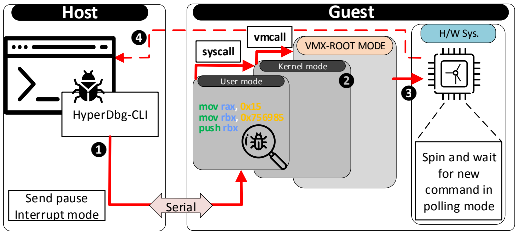


Figure 3: The overall view of the communication in HyperDbg

5.1 Communicating and Task Appliance

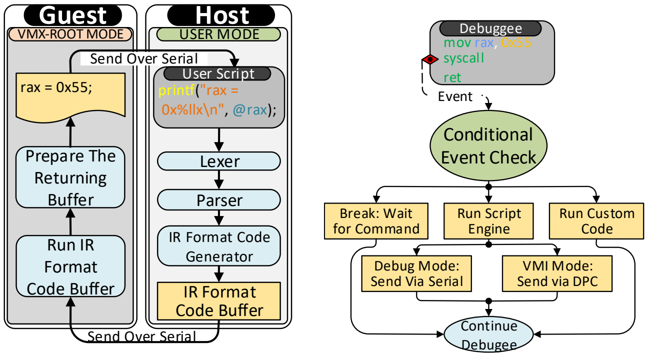
The impracticality of using Windows API for data transmission over network in a debugger can be attributed to the unavailability of interrupts in VMX-root mode (which forces the mode of communication to polling mode) and the need for extra implementation, as Windows uses different device stacks in different IRQL levels for networking. Owing to these challenges, HyperDbg utilizes serial ports for data transfers as it simplifies many aspects of design and usability and enables the use of polling mode. Figure 3 shows the general overview of HyperDbg communication routine.

5.1.1 Sending Data over Serial. Following the connection initialization between a serial device and its co-responding serial port, a connection to the target device can be established by providing the COM argument. HyperDbg supports up to four different serial ports at a time. Furthermore, halting a debuggee is performed by sending an interrupt signal using the interrupt mode of the serial device, which eliminates the need for gritty checks in polling mode when the debuggee is running. The interrupt to the user-mode application of the debuggee is passed down into the kernel-mode, where eventually, a VMCALL is invoked to put the debuggee to the pause state in the VMX-root mode and await further commands (packets) from the debugger.

5.1.2 Communication between Cores. Upon an event getting triggered, HyperDbg checks for a corresponding action and halts every other core in the *VMX-root* mode in case of a break action (by sending Non-Maskable Interrupts (NMIs) [21], which cause the core to spin on a spinlock and invoke a *VM-exit* and await further commands from the debugger), or executing the custom code/script without notifying the other cores, otherwise.

5.2 Kernel-level Script Engine

Modern day debuggers fall short in providing a high-performance and highly customizable scripting framework. Striving to address this gap and faced with the lack of support for direct access to memory in VMX-root mode, we designed a VMX-enabled script engine from the scratch. To the best of authors' knowledge this is the only script-engine solution available in VMX-root mode offering advantageous features like OS spinlock, memory check as well as auxiliary functions (e.g., *printf* and *strlen*). As shown in the overview of the script engine's architecture in Figure 4a, the script engine is comprised of a back-end (that uses *LL(1)* and *LALR(1)* parsers for maximum efficiency) and a front-end that uses a *MASM Style* syntax with C keywords (e.g., if, else, for) and an easily customizable grammar.



(a) HyperDbgs script engine's execution flow

(b) Script Engine invocation as an *event*

Figure 4: Description of Script Engine in HyperDbg

The user-inputted scripts are delivered to the front-end host, scanned via a lexer, and parsed into an Intermediate Representation (IR), which is sent into a buffer over the serial interface into the guest's kernel VMX-root mode for execution. Afterward, a buffer is gradually filled with the execution results and transmitted back to the host. This approach offers substantial performance improvement compared to the conventional bidirectional method used in commodity debuggers (where commands and scripts are sent and parsed line by line) by sending the entirety of the script into the VMX-root mode, and the response back into the user mode, in a unidirectional flow.

As illustrated in Figure 4b, it is also possible to set a script as the *action* of an *event*. In this scenario, the parsed IR script is stored into the VMX-root kernel once, and upon having its corresponding event triggered, the IR is performed locally, thus improving the execution performance of the script engine. A sample script with a detailed description of the example is provided in Appendix E.

6 TRANSPARENCY ANALYSIS

In this section, we investigate the side effects and overhead created by HyperDbg which potentially could be exploited for detection. We further analyze different levels of transparency analysis using malware anti-debugging methods. Furthermore, we propose a statistical approach for immunizing HyperDbg against timing side-channel attacks targeting sub-OS intercepting entities.

6.1 Hypervisor Detection Methods and Mitigations

Detection of sub-OS third-party programs (e.g., hypervisors) is carried out by querying for a set of indicative footprints, such as registry keys, system-calls (e.g., to discover running processes and loaded drivers), and instructions [89] (e.g., CPUID, IDT, LDT). HyperDbg counters these endeavors by intercepting the attempt, forcing a VM-exit, and emulating the corresponding return values with those of a normal, non-virtualized environment in the VM-exit handler. Table 1 provides a comprehensive overview of these methods. More sophisticated hypervisor/VM detection methods exploit timing side channels. The key idea is the fact that certain instructions (e.g., CPUID, GETSEC, INVD, XSETB) cause a VM-exit routine when executed. If the target program is running in a VM, this results in a longer execution time than on bare metal, which can be detected by timing measurements. The listing 2 shows an example of such attacks. In the following we describe the mechanisms in HyperDbg to counter these detection methods.

6.2 Timing Transparency in HyperDbg

HyperDbg's transparent mode offers a solution for hiding the virtualization timing leakage by identifying VM-detecting sequences and replacing the timing values with those of a non-virtualized system. To the best of our knowledge, HyperDbg is the first debugger to offer a practical means to hide the timing footprint used by analyzer software to detect virtualized environments. By offering the means for the construction of a statistical model of the execution time, extensive timing profiling is executed prior to the launch of the VMM module. Hence, a timestamp can be emulated with as much resemblance to the normal operating condition of the guest OS as possible.

```
rdtscp   ; get the current time clock
cpuid    ; Execute a serialization instruction (VM-exit)
rdtscp   ; Delta Timing
```

Listing 2: The timing measurement code by forcing VM-exit

In the initial release of HyperDbg, a proof of concept of this method is implemented using a two-term Gaussian Distribution as a regressor function, as our experiments indicate that it can be a good fit for modeling the execution times of such nature. Figure 5 shows the Probability Distribution Function (PDF) of our measurements by running 10k executions of the sequence in Listing 2, with and without HyperDbg enabled. These values can be derived and the statistical parameters can be recorded for emulation purposes.

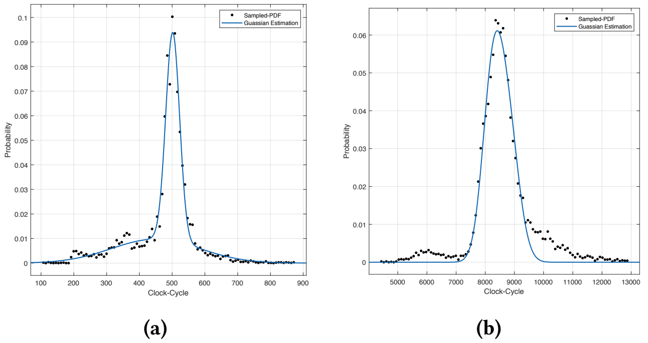
Table 1: Anti-Debugging and Anti-VM exercises and mitigation in HyperDbg

Cat.	Methodology	Example of the Meth.	Example	Mitigation in HyperDbg

Anti-Debugging and Fingerprinting Methods	API-Call
	(System-Call)
		GetCurrentProcessId()
		CreateToolhelp32Snap()
		Process32Next()
		NtQueryInfor.Proc.()
			[8, 59]	Modify results
				via EPT-Hook (hiding process)

	PEB Field	FindWindow()
		IsDebuggerPresent()
		NtGlobalFlags()
			[56]	HyperDbg is not detectable by default

	Heap Structure	HEAP.Flags HEAP.ForceFlags	[47]	HyperDbg is not detectable by default
	#BP Detection	Find BP (0xCC) inst. Read DR (Debug Register)	[60]	!dr to modify and disable unwanted BPs
	Timing Measurement	GetTickCount(), QueryPerf.Counter, GetLocalTime()	[61]	EPT-Hook Modification of results
	Trap-Interrupt	Instruction Prefix, INT 3, 0x2D, Interrupt 0x41	[55]	Set Exception bitmap in VMCS
	Control Flow Manupulation	NtSuspendThread(), NtSetInf.Thread(), CreateThread()	[57]	HyperDbg not detectable by default
Anti-VM/Hypervisor/Emulation	CPU Instructions	CPUID forces a VM-exit certain info in VM	[65]	VM-exit (CPUID result modification)
	Protection Model Instructions	SIDT, SLDT, SGDT STR, SMSW	[8]	VM-exit (emulation and modification)
	Architectural Delta-Timing	RDTSC+CPUID+RDTSC RDTSC(P)+RDTSC(P)	[65]	HyperDbg Trans. Mode (!hide command)
	In/Out Instructions	Magic I/O port in VMware	[22]	VM-exit handled (I/O bitmap)
	Invalid MSR Access	Invalid MSR issues General Protection (#GP)	[98]	Emulate !msrread/!msrwrite command
	Exception Handling	Try-Catch General Protection Excep. (#GP)	[58]	Handled by default Inject routine into user-mode



(a)

(b)

Figure 5: PDF distribution of timing measurement for deactivated HyperDbg (a), with activated HyperDbg (b)

HyperDbg currently enables covering the VM timing leakage by providing two methods that are: 1) Adjusting the MSR register that keeps track of the CPU's time which is referred as `IA32_TIME_STAMP_COUNTER`, and 2) Emulating the results of *RDTSC* and *RDTSCP* instructions that provide the means for user-level applications for accessing the CPU timing values. Each of these methods comes with its own set of advantages and setbacks. The former approach does not require a VM-exit for its function, which simplifies the solution and allows for covering more complex VM-detection sequences, but might increase the instability and interfere with the normal functionality of the system as other applications are also reliant on this mechanism for their timing measurements. In contrast, the latter method (Figure 6) does not cause any interference with the inner workings of the system, but requiring a VM-exit adds a layer of complexity, as emulating timing values expected by an examiner program armed with sophisticated patterns for VM-detection would require extra considerations.

The transparency function can be used on a process or a list of executables, as global emulation of timing instructions would most likely disturb primary functionalities of the system (our experiments show disturbances in the screen driver, as well as audio output performance when a global emulation is implemented).

7 EVALUATION

In this section, we thoroughly evaluate HyperDbg in terms of its transparency and performance in different scenarios.

7.1 Transparency Evaluation

There are multiple debugger detectors and analyzers, both commercial and open-source, to examine against the transparency methods. For our initial evaluation, we have test the implemented methods on the well-known *pafish* [3] software which employs a collection of anti-debugging and protection methods.

In accordance with our expectations, the first method, which involves updating the `IA32_TIME_STAMP_COUNTER`, interferes with

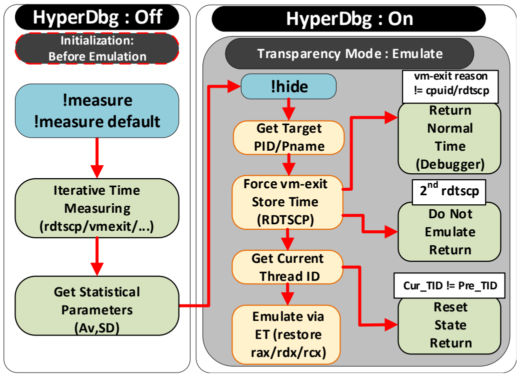

HyperDbg : Off
Initialization: Before Emulation
!measure
!measure default
Iterative Time Measuring (rdtscp/vmexit/...)
Get Statistical Parameters (Av,SD)
HyperDbg : On
Transparency Mode : Emulate
!hide
Get Target PID/Pname
Force vm-exit Store Time (RDTSCP)
Get Current Thread ID
Emulate via ET (restore rax/rdx/rcx)
vm-exit reason != cpuid/rdtscp
Return Normal Time (Debugger)
2nd rdtscp
Do Not Emulate Return
Cur_TID != Pre_TID
Reset State Return

Figure 6: State Diagram Process of *rdtsc/rdtscp* emulation by HyperDbg

the primary functions of the system and causes screen flickering during our experiments. Regardless, the second method (emulation) was able to successfully pass *pafish* analyser with 100% success rate when enabling emulation for the pafish process.

As an extension to our transparency analysis, we separately evaluate HyperDbg against common anti-debugging methods and on commercial off-the-shelf packers/protectors.

7.1.1 *Evaluation by Anti-Debugging, -VM, and -Hypervisor.*

Table 1 describes the common anti-debugging and anti-virtualization methods [1, 97], and HyperDbg's countermeasure to avoid detection. Each of these methods is applied separately in HyperDbg's *Debugger mode* and activates the suitable countermeasure to verify the transparency of the proposed debugger. Furthermore, for an end to end transparency analysis, all the mitigation techniques are activated. We evaluate HyperDbg against a collection of evasive malware which is known to employ a wide range of anti-debugging and VM technologies.

In our experiments, we analyzed 10,853 samples of malware in different categories derived from a malware database [88]. Each of these malware samples are executed in HyperDbg's normal and *Transparent Mode* in the *Debugger Mode* as well as *x64dbg* (user-mode debugger) and *WinDbg* (kernel-mode debugger) in Microsoft Windows 10 20H1 for comparison. We observe that a relatively large percentage of the samples detect the debugging environment in *WinDbg* and *x64dbg* and change their behavior accordingly to conceal their malicious behavior. Considering *WinDbg* is the baseline debugger, Figure 8a reports the percentage of successfully executed malware samples where the debugger is attached. For this experiment, we measure the success rate of the execution of malware samples by carefully logging the syscall sequence in the target system by hooking the syscalls (changing IA32_LSTAR). As shown in the Figure 8a, HyperDbg's Transparent mode increases the transparency surface by 22% compared to *WinDbg*, successfully executing malware samples in all four categories without being detected. This is evident since HyperDbg is performed at the hypervisor level, leaving minimal footprints, which anti-debugging/VM methods in malware cannot determine. Also, it is worthy to note that some malware samples detect user-mode artifacts introduced by *x64dbg*, making it even less transparent than *WinDbg*.

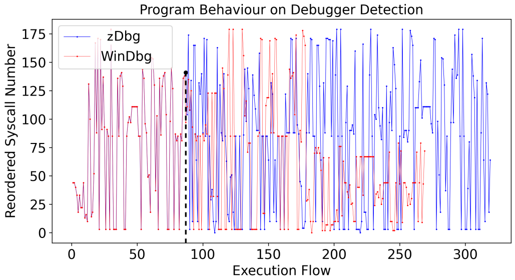


Figure 7: Syscalls executed in a malware using HyperDbg and WinDbg

7.1.2 *Syscall Malware Analysis.*

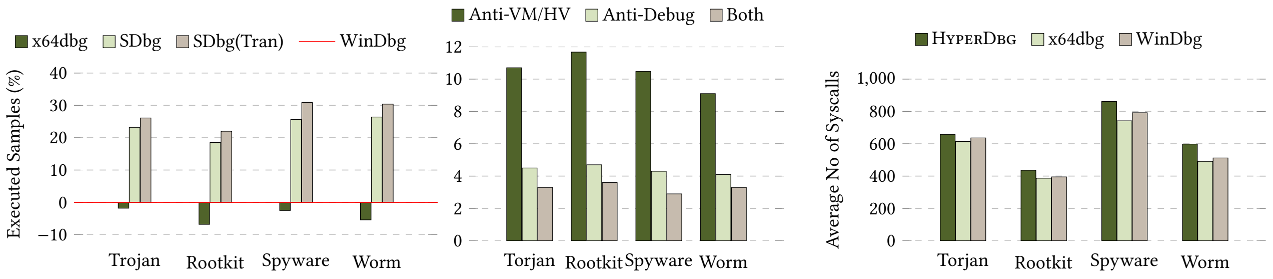
The transparent deployment of HyperDbg has the advantage of stealthily monitoring malware execution. Figure 7 shows the syscall execution flow of a sample malware on HyperDbg. For high-level comparison it is possible to extract execution flow and divergence point of evasive malware here. One can execute the malware on a bare-metal system with no debugging present rather than HyperDbg's transparent mode for monitoring purposes. We attached a kernel-mode debugger (*WinDbg*) to the malware and executed the malware, and recorded the syscall execution flow. We have employed a simple script code using HyperDbg's script engine to trace the *SYSCALLs* in the execution flow. (See Appendix E) As Figure 7 depicts, the execution flow of the malware does not follow a similar behavior in the different environments. As a simple analysis, we could come to the decision that this specific malware uses some level of Anti-Debugging methods. To evade its malicious intention, an entirely different (and most probably safe) execution path is chosen within the malware code when a debugger is detected. The same approach is used to measure whether the malware samples are running or not, shown in Figure 8a.

7.1.3 *Evaluation by Packers and Protectors Testing.*

HyperDbg is tested by binary files that are processed with packers and protectors. These packed/protected binary files are tested on different debuggers as well as HyperDbg in both regular debugging and transparent mode debugging. Table 2 shows the results of attaching and debugging these protected binaries.

Table 2: Evaluation and comparison of HyperDbg for integrated software via packers/protectors

Packer/Protector	File Type	WinDbg	x64dbg	Ollydbg	**HyperDbg**	**HyperDbg's Trans. Mode**
ASPack.V2.42	PE32	Error	✓	✓	✓	✓
Enigma.V4.30	PE64	✗	✗	N/A	✗	✓
Net_Crypto.V5	PE32	✗	✗	✗	✓	✓
Obsidium.V1.7	PE64	✗	✗	N/A	✓	✓
Themida.V3.0.4	PE64	✗	✗	N/A	✓	✓
Upx.V3.96	PE64	✓	✓	✓	✓	✓
Vmprotect.V.2.13	PE64	✗	✗	N/A	✗	✓
MEW11.V1.2	PE32	✓	✓	✓	✓	✓
Pecompact.V3.11	PE32	✓	✓	✓	✓	✓
PELock.V2.0	PE32	✗	✗	✗	✓	✓
Petite.V2.4	PE32	Error	✓	✓	✓	✓
TeLock.V0.98	PE32	✓	✓	✓	✓	✓
YodaCrypter.V1.02	PE32	✗	✗	✗	✓	✓



(a) **Rate of successful executed malware samples in debuggers**

(b) **Anti-Debug/VM/HV methods on Table 1 using HyperDbg**

(c) **Averaged number of executed Syscalls in malware samples in debuggers**

Figure 8: Three simple graphs

7.2 Performance Evaluation

In terms of performance, we analyze HyperDbg in three debugging scenarios which are discussed in the following.

7.2.1 Performance Analysis of Scenario 1: Step-in. Single stepping is one of the most fundamental functionalities of a debugger that has been carefully optimized in HyperDbg to become as fast as possible. To evaluate the performance of this functionality, we considered n = 100 sets of 65,536 predefined instructions (a particular application) to evaluate the performance. HyperDbg was able to instrument the instruction sets on average in 6 minutes and 51 seconds (μ = 411 seconds) with the standard deviation of σ = 28.3 seconds. It took *WinDbg* on average 1,221 seconds with the standard deviation of σ = 118.4 seconds to perform the same function on the same instructions sets. Thus, HyperDbg takes 2.97 less time on average to execute the same analysis compared to *WinDbg*'s instrumenting.

7.2.2 Performance Analysis of Scenario 2: Conditional Breakpoints. An essential part of analyzing binaries are conditional breakpoints, which have been implemented robustly and substantially faster than almost all of the currently available debuggers in HyperDbg.

To evaluate the performance of HyperDbg on conditional breakpoints, we set one on the frequently used *nt!ExAllocatePoolWi*thTag function, checking whether the *RAX* register contains a specific value. As performance metric, we count the number of times the condition was checked within 5 minutes both for HyperDbg and *WinDbg*. To mitigate timing noise on our results, we repeat the experiments for n = 50 times. As the baseline of the performance, *WinDbg* checks on average 6,941 conditions. At the same time, HyperDbg checks on average 9,153,731 conditions with its classic implementation of EPT Hook and checks on average 23,214,792 conditions with its detours style EPT Hook.

These result show that HyperDbg's script engine achieves the astonishing 1,319 and 3,345 average fold speedups compared to *WinDbg* in classic EPT Hook and detours EPT Hooks, respectively.

This significant speed gain comes from the fact that based on the design, HyperDbg checks and evaluates scripts directly in the kernel and VMX-root mode and does not need the assistance of the user-mode for this end. Thus, in contrast to *WinDbg*, nothing is transferred to the debugger during the execution of the script.

This difference is also visible in the system's overall performance during the execution of the benchmarks on the debugger. In *WinDbg*, the system slows down to the point that it seems the system has come to a halt since not even the most basic computations, such as cursor movements are properly processed. While in HyperDbg's case, even though the performance of the system is still slow, it's usable. Therefore, other tasks could still be normally performed on the system, which allows alteration and addition of new conditional breakpoints while the test is performed.

7.2.3 Performance Analysis of Scenario 3: Analyzing Syscalls. Setting breakpoints on syscalls is another scenario that can be used for evaluation of the performance of HyperDbg.

Generally speaking, it is not possible to set a breakpoint on syscall-handler routines in other debuggers like *WinDbg*. However, it is possible to trace system calls by setting breakpoints on functions responsible for dispatching the SYSCALL numbers. In HyperDbg, it is possible to set breakpoints on syscall-handler routines and to emulate system calls. For the performance evaluation, we perform n = 50 experiments each lasting 300 seconds. *WinDbg* executes 2,559 syscalls, while at the same time HyperDbg executes 5,166,430. Hence, HyperDbg is on average 2018x times more efficient than *WinDbg* in tracing syscall routines.

8 APPLICATIONS

With the privileged access level and the newly-presented APIs, HyperDbg can be used in many applications.

8.1 Debugging Devices

HyperDbg supports the general functionality of any other debuggers, i.e., pausing and stepping through the instructions, read/write on memory, read and modify registers, and putting breakpoints anywhere in the program. Plus, it has many other creative events to ease the debugging process.

One of the unique capabilities of HyperDbg is its ability to debug the communications of the system with external devices. The user can monitor each x86 I/O port separately for port mapped I/O (PMIO) devices and use EPT to monitor Memory Mapped I/O (MMIO) devices. Since I/O instructions and EPT modifications are treated as events in HyperDbg, the user is able to monitor the executions of *IN* and *OUT* instructions and create separate logs. Moreover, it is also possible to modify the registers in the script engine and therefore, delivering the modified values to the operating system. In addition to debugging Port Mapped I/O and Memory

Mapped I/O, HyperDbg is also capable of notifying the user about the interrupts from external devices. For example, HyperDbg can be configured to intercept any particular interrupt form an external device (e.g., a PS/2 keyboard) and allow the user to halt Windows to investigate the device in case it occurs, or simply ignore the interrupt and allow the operating system to continue normally.

8.2 Fuzzing

One of the main problems of kernel fuzzing is the fact that every invalid value causes a kernel error and thus a BSOD. HyperDbg can avoid these errors by handling them even before the OS is notified and help fuzzing (e.g., by measuring the code coverage).

8.2.1 Code Coverage. As it is common to access invalid memory location during fuzzing, especially as with a high code coverage, the target program is exhaustively tested with new input vectors. In this scenario, HyperDbg is notified in the case of any invalid access to the memory by using events related to the exceptions and faults. Hence, if the input test parameters cause a system crash due to the invalid access, HyperDbg is notified prior to the OS, and it is possible to efficiently discard the crash before calling the OS error handling routines. This makes it possible to restore the system state and continue the fuzzing process with the target function gaining a huge performance increase compared to a naive code coverage handled through the OS.

HyperDbg's script engine provides the possibility to execute brute-force tests for a target program using simple scripts. For instance, one can re-execute a target function arbitrarily often, each time with different parameters.

The proposed instrumentation step-in procedure in HyperDbg forces the system to only run the specific process without switching to other processes. Consequently, the CPU only executes the targeted codes during the fuzzing process and returns the program flow to the initial state of fuzzing if any crash appears. Using the script engine, it is then possible to prepare the CPU for the next stage of fuzzing with new parameters in an entirely automated mechanism. This method results in a fine-grained approach to fuzz both user and kernel programs with high-performance execution.

8.3 Malware Analysis

As another essential application, HyperDbg features a transparent debugging tool that can be used for evasive malware analysis. Given the unique toolset of HyperDbg, online malware analysis is armed with a high-performance run-time script-engine, which makes the process effective and substantially faster. We describe a simple and transparent syscall malware analysis using HyperDbg in Section 7.1.2. In the following, we survey the applicability of HyperDbg in a Windows vulnerability.

8.3.1 Analysis of a Vulnerability: A Case Study. During our experiments, we rediscover a full-kernel mode Bootkit known as *Pitou* [12], to which the latest Windows versions are still vulnerable. We briefly describe *Pitou* as a case study analyzed by HyperDbg.

Pitou is able to attack the victim system by bypassing the user access control and performing privilege escalation, which enables it to infect the Master Boot Record (MBR). This allows it to inject its kernel payload at the time of Windows startup without facing any resistance from Kernel Mode Code Signing (KMCS) policy. Pitou is then able to take control of the lowest level components of the OS (e.g., Windows network driver - NDIS) and utilize VM-level code that is not executable natively on Windows to obfuscate itself from conventional disassemblers, which makes it much more difficult to analyze it. To the date of writing, it is still able to infect the systems running the latest version of Microsoft Windows with a 0-day local privilege escalation.

Pitou also employs advanced anti-debugging and anti-sandboxing techniques that look for any traces of the execution in a non-native execution environment by performing inspections on Windows registry, kernel modules, disk devices, BIOS memory, and measurement of CPU ticks using RDTSC. These methods have been shown to be updated by the creator of the malware in time. In our tests, the malware detects the debugger environment with some of the most well-known and widely-used debuggers like *WinDbg*, *x64dbg*, and *Ollydbg*[71]. It deviates from its normal behavior on every single debugger. However, with HyperDbg's transparent mode, we successfully execute the malware and perform an extensive dynamic analysis to reverse-engineer its execution flow.

8.3.2 Digital Forensics & Incident Response (DFIR). HyperDbg can be used extensively in the DFIR to detect signs of attacks. For instance, the script engine of HyperDbg can be utilized for developing a pre-built plugin to monitor the top abused APIs/syscalls under user-specified conditions and on any subset of the processes (e.g., critical system processes only), allowing the inspector to adjust between the conciseness and thoroughness of the logs based on their preference. Additionally, HyperDbg is capable of classifying the APIs into different categories of attacks (e.g., code injection, keylogging, or discovery) and transmitting the results over TCP/Named Pipe/File using Event Forwarding.

8.3.3 Attempt to Exploit Detection. HyperDbg can be used to detect many exploitation techniques. Often, exploits modify a special structure as the final payload, such as the token of a process [14, 45].

In the above example, HyperDbg can be used to monitor TOKEN structure and detect any access (or more precisely, any write) to this structure. After that, this abnormal behavior can be traced back to reach the initial phase of exploit and reveal its method.

8.4 Software Performance Analysis

HyperDbg can be used for performance and security analysis in software development and testing. For example, the highly optimized methods available in HyperDbg can be utilized for intercepting events such as page faults, with marginally superior performance compared to alternative means and methods used in an user-mode analysis tool [6, 7, 41].

HyperDbg can detect page faults both the operating system and applications. In previous works, Shadow Paging, Page Tracking, and Pseudo-paging methods were used to detect page-faults [41]. Detecting page-faults is beneficial in the evaluation of applications that opt to improve their performance by minimizing the frequency of page faults. HyperDbg can detect page faults by exploiting exception bitmaps and providing it as an event. Using this method, HyperDbg can provide the exact address of fault area (CR2) for further investigations. This method is transparent to the operating system and does not change its semantics.

9 RELATED WORK

Developing a debugger and low-level software analyzer has been regarded as a crucial topic for the computer community due to its impact and applicability in a wide range of scientific research and industrial products. The implications can be generally categorized into two main groups: 1) Hardware-level malware analysis and 2) System isolation, monitoring, and sandboxing.

Over the past decade, many researchers have proposed several debugging methods based on the ring -1 (sub-OS level) infrastructure to address these issues. However, in terms of transparency level, applicability, performance, and generalization, these tools fail to present a suitable solution for the community. HyperDbg as an open-sourced and general hardware-assisted debugger that aims to provide researchers and computer engineers with a tool to help resolve the aforementioned issues.

Hardware-level malware analysis. Malware developers have managed to develop many strategies and techniques that allow them to escape from almost every form of detection methodology, including virtualization, debugging, and emulation techniques. Anti-debugging and anti-virtualization techniques used in early malware [10] employ numerous evasion methods to hide or reduce malicious activities. These anti-detection methods are analyzed comprehensively in the recent study by Galloro et al. [32] where over 92 classes of evasive techniques executed by modern malware.

Furthermore, hardware-based artifacts such as processor's cache actualization [75], scheduling leakage in simultaneous multithreading (SMT) [62], as well as timing side effects [72] of monitoring facilities can be observed by evasive malware.

As thoroughly discussed by Garfinkel et al. [35], achieving full transparency against malware running in a virtualized environment is extremely challenging. Considering all of the issues, previous work proposed frameworks such as Apate [82] to hide debugging procedures from malware. Likewise, other work proposes resilient malware detectors against evasive malware using hardware features [44, 85]. Leon et al. [53] study the possibility of utilizing hypervisors to detect, deactivate and analyze evasive malware by employing low-level processor features.

Unlike previous solutions, which merely focus on transparency rather than functionality, our method in HyperDbg to approach malware analysis provides a richly equipped debugging facility by pushing the deployment of more complex functionalities deeper into the hypervisor. This approach not only provides transparency but gains significant performance, as well as rich functionality all together in a singular framework making HyperDbg applicable for real-world malware analysis.

System isolation, tracing, and sandboxing. Due to the increasing complexity of the malware evasion techniques, researchers have recently evolved the environment from VM-based sandboxes such as CWSandbox [90] and Cuckoo sandbox [31] to Bare-Metal sandboxes like BareBox [49], and BareCloud [50] to minimize the leakage of the virtualization environment. Pioneered by Ether [25] as the first hypervisor-based analyzer with more transparency level, Malt [97] and Ninja [68] target Intel's SMM and Arm's TrustZone to present hardware-level debugging and process tracing as well as sandboxing primarily aiming at malware debugging. Although transparent to some level, all these works provide simple functionalities and low-speed tracing, making them unsuitable for deep and dynamic code analysis. HyperDbg addresses these shortcomings by providing real-time user-specified debugging functionalities using VMX-based script-engine. Furthermore, even though the hardware overhead is negligible for most previous solutions, the total debugging execution flow is prolonged due to the continuous ring transportation to perform dynamic code analysis. This drawback is fundamentally solved in HyperDbg 's design.

Recently, researchers employ newer hardware-based features (e.g. Intel-PT) for low-level hypervisor fuzzing [79, 80], kernel failure reverse debugging [36] as well as machine learning approaches [5], to discover vulnerabilities and bugs. Similar ideas are deployed for embedded systems arming application tracing [26], debugging[67, 70], unpacking[94] on Arm processors.

Though designed as debugger, HyperDbg delivers a superior level of transparency for low-level sandboxing and isolation. Moreover, its architectural design and VMX-enabled script engine provide an accurate and fast process tracking of arbitrary binaries.

Feature comparison among existing debuggers A Comprehensive feature comparison is given in Table 5 in Appendix D.

10 CONCLUSION

With the expanding hardware support in modern processors, it is now more than ever crucial to employ hardware-assisted techniques in software debugging. Common software debugging solutions rely on traditional OS-dependent APIs for code functionality analysis, vulnerabilities detection, and reverse engineering. Modern packed software and evasive malware employ sophisticated anti-debugging methods to hide their primary functionalities and withstand reverse engineering attempts on the extensively used debugging solutions. This paper presents HyperDbg, an open-source hypervisor-level debugging tool with transparency and performance in mind. HyperDbg exploits Intel VT-x and Intel EPT to present multiple new debugging modules, useful for fuzzing, malware analysis, and reverse engineering. We propose a novel VMX-level script engine in HyperDbg's core which gives an unmatched debugging performance useful for software fuzzing as user-mode to kernel-mode (and vice versa) transfer is entirely avoided. Our evaluation shows a high level of stealth code analysis against malware classes and unprecedented performance in terms of debugging functionality among other available kernel debuggers. Finally, HyperDbg is designed modular and scalable for convenient usage in both academia and industry.

AVAILABILITY

HyperDbg is fully open-source and is available at:
https://github.com/HyperDbg

REFERENCES

[1] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. 2019. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–28.

[2] Sanjeev Kumar Aggarwal and M Sarath Kumar. 2002. Debuggers for Programming Languages.

[3] Ortega Alberto. 2022. pafish. https://github.com/a0rtega/pafish. Accessed: 2022-02-15.

[4] Theodoros Apostolopoulos, Vasilios Katos, Kim-Kwang Raymond Choo, and Constantinos Patsakis. 2021. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems* 116 (2021), 393–405.

[5] Shushan Arakelyan, Sima Arasteh, Christophe Hauser, Erik Kline, and Aram Galstyan. 2021. Bin2vec: learning representations of binary executable programs for security tasks. *Cybersecurity* 4, 1 (2021), 1–14.

[6] Reza Azimi, Michael Stumm, and Robert W Wisniewski. 2005. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*. 101–110.

[7] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. 2013. The {Page-Fault} Weird Machine: Lessons in Instruction-less Computation. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*.

[8] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat* 1 (2012), 1–27.

[9] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. 2016. Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 323–336.

[10] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN)*. IEEE, 177–186.

[11] Tzi-cker Chiueh. 2008. Fast bounds checking using debug register. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 99–113.

[12] Citeseer. 2019. PITOU : Kernel Payload and DGA. (2019).

[13] IBM Co. 2019. Hypervisors. https://www.ibm.com/cloud/learn/hypervisors. Accessed: 2022-02-15.

[14] Mitre Co. 2022. Access Token Manipulation. https://attack.mitre.org/techniques/T1134. Accessed: 2022-02-15.

[15] Microsoft Co. 2022. What is IRQL and why is it important? https://techcommunity.microsoft.com/t5/ask-the-performance-team/what-is-irql-and-why-is-it-important/ba-p/372666. Accessed: 2022-02-15.

[16] Microsoft Co. 2022. Windows Debugger (WinDbg). https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools. Accessed: 2022-02-15.

[17] NuMega Co. 2022. SoftIce. https://www.angelfire.com/bug/ass_1/Readme.htm. Accessed: 2022-02-15.

[18] VMWare Co. 2022. What is a hypervisor? https://www.vmware.com/topics/glossary/content/hypervisor. Accessed: 2022-02-15.

[19] GNU Community. 2022. GDB. https://www.gnu.org/software/gdb/. Accessed: 2022-02-15.

[20] Ghidra Community. 2022. ghidra. https://ghidra-sre.org/

[21] Intel Corporporation. 2018. Intel 64 and ia-32 architectures software developer manuals.

[22] Cyberbit. 2022. Anti-VM and Anti-Sandbox Explained. https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/.

[23] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*. 289–298.

[24] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. 2013. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering* 2, 3 (2013), 222–225.

[25] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. 51–62.

[26] Yunlan Du, Zhenyu Ning, Jun Xu, Zhilong Wang, Yueh-Hsun Lin, Fengwei Zhang, Xinyu Xing, and Bing Mao. 2020. Hart: Hardware-assisted kernel module tracing on arm. In *European Symposium on Research in Computer Security*. Springer, 316–337.

[27] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. 2020. On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security* 15 (2020), 2750–2765.

[28] Mark Ermolov and Maxim Goryachy. 2019. Intel VISA: Through the Rabbit Hole. *Black Hat Asia* (2019).

[29] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. 2010. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 417–426.

[30] Manuela K Ferreira, Henrique C Freitas, and Philippe OA Navaux. 2008. From Intel VT-x to MIPS: An ArchC-based Model to Understanding the Hardware Virtualization Support. In *Workshop on Computer Education Architecture, Beijing, China*. 9–15.

[31] Cuckoo Foundation. 2022. Cuckoosandbox. https://cuckoosandbox.org/. Accessed: 2022-02-15.

[32] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. 2022. A Systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security* 113 (2022), 102550.

[33] Peter Baer Galvin. 2009. VMware vSphere vs. Microsoft Hyper-V: A technical analysis. *Corporate Technologies, CTI Strategy White Paper* (2009), 32.

[34] Shang Gao and Qian Lin. 2012. Debugging classification and anti-debugging strategies. In *Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies*, Vol. 8350. International Society for Optics and Photonics, 83503C.

[35] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility Is Not Transparency: VMM Detection Myths and Realities.. In *HotOS*.

[36] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 281–292.

[37] Part Guide. 2019. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3C: Chapter 24, VIRTUAL MACHINE CONTROL STRUCTURES (Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls)* 3C (2019).

[38] Part Guide. 2019. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3C: Chapter 24, VIRTUAL MACHINE CONTROL STRUCTURES (Table 24-7. Definitions of Secondary Processor-Based VM-Execution Controls)* 3C (2019).

[39] Steven M Hand. 1999. Self-paging in the Nemesis operating system. In *OSDI*, Vol. 99. 73–86.

[40] Hex-Rays. 2022. IDA Pro. https://hex-rays.com/ida-pro/

[41] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review* 43, 3 (2009), 14–26.

[42] Shun-Wen Hsiao, Yeali S. Sun, and Meng Chang Chen. 2020. Hardware-Assisted MMU Redirection for In-Guest Monitoring and API Profiling. *IEEE Transactions on Information Forensics and Security* 15 (2020), 2402–2416. https://doi.org/10.1109/TIFS.2020.2969514

[43] Immunity Inc. 2022. immunitydbg. https://www.immunityinc.com/products/debugger/index.html. Accessed: 2022-02-15.

[44] Md Shohidul Islam, Khaled N Khasawneh, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Lei Yu. 2021. Efficient hardware malware detectors that are resilient to adversarial evasion. *IEEE Trans. Comput.* (2021).

[45] Mohammad Sina Karvandi, Saleh Khalaj Monfared, Mohammad Sina Kiarostami, Dara Rahmati, and Saeid Gorgin. 2022. A TSX-Based KASLR Break: Bypassing UMIP and Descriptor-Table Exiting. In *International Conference on Risks and Security of Internet and Systems*. Springer, 38–54.

[46] Sina Karvandi. 2019. Hypervisor From Scratch – Part 7: Using EPT & Page-Level Monitoring Features. (2019). https://rayanfam.com/topics/hypervisor-from-scratch-part-7/

[47] Jong-Wouk Kim, Jiwon Bang, and Mi-Jung Choi. 2020. Defeating Anti-Debugging Techniques for Malware Analysis Using a Debugger.

[48] Jong-Wouk Kim, Jiwon Bang, Yang-Sae Moon, and Mi-Jung Choi. 2019. Disabling anti-debugging techniques for unpacking system in user-level debugger. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 954–959.

[49] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. BareBox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*. 403–412.

[50] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 287–301.

[51] Andrew Ko and Brad Myers. 2008. Debugging reinvented. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 301–310.

[52] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 386–395.

[53] Roee S Leon, Michael Kiperberg, Anat Anatey Leon Zabag, and Nezer Jacob Zaidenberg. 2021. Hypervisor-assisted dynamic malware analysis. *Cybersecurity* 4, 1 (2021), 1–14.

[54] Cătălin Valeriu Liţă, Doina Cosovan, and Dragoş Gavriluţ. 2018. Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in UPA packers. *Journal of Computer Virology and Hacking Techniques* 14, 2 (2018), 107–126.

[55] Check Point Software Technologies LTD. 2022. Anti-Debug: Assembly instructions. https://anti-debug.checkpoint.com/techniques/assembly.html.

[56] Check Point Software Technologies LTD. 2022. Anti-Debug: Debug Flags. https://anti-debug.checkpoint.com/techniques/debug-flags.html.

[57] Check Point Software Technologies LTD. 2022. Anti-Debug: Direct debugger interaction. https://anti-debug.checkpoint.com/techniques/interactive.html.

[58] Check Point Software Technologies LTD. 2022. Anti-Debug: Exceptions. https://anti-debug.checkpoint.com/techniques/exceptions.html.

[59] Check Point Software Technologies LTD. 2022. Anti-Debug: Misc. https://anti-debug.checkpoint.com/techniques/misc.html.

[60] Check Point Software Technologies LTD. 2022. Anti-Debug: Process Memory. https://anti-debug.checkpoint.com/techniques/process-memory.html#software-breakpoints.

[61] Check Point Software Technologies LTD. 2022. Anti-Debug: Timing. https://anti-debug.checkpoint.com/techniques/timing.html.

[62] Yehonatan Lusky and Avi Mendelson. 2021. Sandbox Detection Using Hardware Side Channels. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 192–197.

[63] Thomas Mandl, Ulrich Bayer, and Florian Nentwich. 2009. ANUBIS ANalyzing unknown BInarieS the automatic way. In *Virus bulletin conference*, Vol. 1. 02.

[64] Gary McGraw and Greg Morrisett. 2000. Attacking malicious code: A report to the infosec research council. *IEEE software* 17, 5 (2000), 33–41.

[65] Christian Rossow Michael Brengel, Michael Backes. 2010. Detecting Hardware -Assisted Virtualization. (2010).

[66] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. 2006. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal* 10, 3 (2006).

[67] Zhenyu Ning, Chenxu Wang, Yinhua Chen, Fengwei Zhang, and Jiannong Cao. 2021. Revisiting ARM Debugging Features: Nailgun and Its Defense. *IEEE Transactions on Dependable and Secure Computing* 01 (2021), 1–1.

[68] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on {ARM}. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 33–49.

[69] Zhenyu Ning and Fengwei Zhang. 2018. Hardware-assisted transparent tracing and debugging on ARM. *IEEE Transactions on Information Forensics and Security* 14, 6 (2018), 1595–1609.

[70] Zhenyu Ning and Fengwei Zhang. 2019. Understanding the security of arm debugging features. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 602–619.

[71] Yuschuk Oleh. 2022. OllyDbg. https://www.ollydbg.de/. Accessed: 2022-02-15.

[72] Yoshihiro Oyama. 2019. How does malware use RDTSC? A study on operations executed by malware with CPU cycle measurement. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 197–218.

[73] Jonas Pfoh, Christian Schneider, and Claudia Eckert. 2011. Nitro: Hardware-based system call tracing for virtual machines. In *international workshop on security*. Springer, 96–112.

[74] phoenixNAP. 2022. What is a Hypervisor? Types of Hypervisors 1 & 2. https://phoenixnap.com/kb/what-is-hypervisor-type-1-2. Accessed: 2022-02-15.

[75] François Plumerault and Baptiste David. 2021. DBI, debuggers, VM: gotta catch them all. *Journal of Computer Virology and Hacking Techniques* 17, 2 (2021), 105–117.

[76] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield, Dan Magenheimer, Jun Nakajima, and Asit Mallick. 2005. Xen 3.0 and the art of virtualization. In *Linux symposium*, Vol. 2. Citeseer, 65–78.

[77] Nguyen Anh Quynh and Kuniyasu Suzaki. 2010. Virt-ICE: Next-generation debugger for malware analysis. *Black Hat USA* (2010).

[78] Rayanfam.Com. 2021. Hypervisor From Scratch – Part 8: How To Do Magic With Hypervisor! https://rayanfam.com/topics/hypervisor-from-scratch-part-8/. Accessed: 2022-02-15.

[79] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing.. In *NDSS*.

[80] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*. 2597–2614.

[81] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical enclave malware with Intel SGX. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 177–196.

[82] Hao Shi and Jelena Mirkovic. 2017. Hiding debuggers from malware with apate. In *Proceedings of the Symposium on Applied Computing*. 1703–1710.

[83] Julian Stecklina and Thomas Prescher. 2018. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).

[84] LLDB Team. 2022. LLDB. https://lldb.llvm.org/. Accessed: 2022-02-15.

[85] Donghai Tian, Qianjin Ying, Xiaoqi Jia, Rui Ma, Changzhen Hu, and Wenmao Liu. 2021. MDCHD: A novel malware detection method in cloud using hardware trace and deep learning. *Computer Networks* 198 (2021), 108394.

[86] Amit Vasudevan and Ramesh Yerraballi. 2005. Stealth breakpoints. In *21st Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 10–pp.

[87] VMware Inc. 2019. Performance Evaluation of Intel EPT Hardware Assist. https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf. Accessed: 2022-02-15.

[88] vx underground. 2021. vx-underground malware collection. *https://www.vx-underground.org/* 1 (2021).

[89] Gary Wang, Zachary J. Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2015. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/wang

[90] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy* 5, 2 (2007), 32–39.

[91] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. 189–198.

[92] Carsten Willems, Ralf Hund, and Thorsten Holz. 2013. Hypervisor-based, hardware-assisted system monitoring. In *Virus Bulletin Conference*.

[93] x64 Debugger. 2022. X64. https://x64dbg.com/

[94] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. 2021. Happer: Unpacking Android apps via a hardware-assisted approach. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1641–1658.

[95] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 227–238.

[96] Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 297–300.

[97] Fengwei Zhang, Kevin Leach, Angelos Stavrou, and Haining Wang. 2016. Towards transparent debugging. *IEEE Transactions on Dependable and Secure Computing* 15, 2 (2016), 321–335.

[98] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using hardware features for increased debugging transparency. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 55–69.

APPENDICES

A EVENT COMMANDS IN HYPERDBG

Table 3: The list of the supported events in HyperDbg

!epthook	Classic EPT hook
!epthook2	EPT hook with detours hooking
!syscall	Hook execution of system-calls
!sysret	Hook execution of sysret instruction
!monitor	Monitors any access (Read/Write) to a region of memory
!cpuid	System-wide CPUID instruction execution detection
!msrread	System-wide RDMSR instruction execution detection
!msrwrite	System-wide WRMSR instruction execution detection
!tsc	System-wide RDTSC/RDTSCP instructions execution detection
!pmc	System-wide RDPMC instruction (performance counter) execution detection
!exception	Monitors and hooks first 32 entries of Interrupt Descriptor Table (IDT)
!interrupt	Monitors and hooks external-interrupts 33 to 256 entries of Interrupt Descriptor Table (IDT)
!dr	Detects any reads or write into hardware debug registers
!ioin	Monitors and ability to modify I/O ports and IN instruction
!ioout	Monitors and ability to modify I/O ports and OUT instruction
!vmcall	System-wide VMCALL instruction (hypercalls) execution detection

B PROCESS/PROCESSOR/EXECUTION MODE SWITCH

In this section, we describe the architecture related to switching between processes, processors, and different modes of execution in HyperDbg.

B.1 Detecting Execution-mode Changes (Kernel-mode to User-mode)

Detecting changes to the operating mode is performed via the same mechanism used in the *i* command in HyperDbg. The proper method for implementing this functionality would be checking the CS register, fetching GDT, and checking the Long Mode flag. However, since the CS for *wow64* and native code is set to a constant value across all versions of Windows, the CS register check is sufficient for the determination of a mode switch.

B.2 Switching to a New Processor/Process

HyperDbg uses a straightforward mechanism to switch between cores. Each core has its own spinlock to wait on VMX-root mode. By unlocking the spinlock assigned to the new core and setting the spinlock of the current core, it is possible to enter a waiting state. Consequently, HyperDbg calls the command handler from the new core, making the new core responsible for getting commands. Note that HyperDbg is designed to have a single core for getting commands at any given time. Moreover, switching to a new process is performed by monitoring changes to the *CR3* register. Each time Windows changes the memory layout of any process, the *CR3* is changed, and HyperDbg checks whether or not Windows has switched to the target process. If the memory layout is changed and the target process is now on the execution stage of the Windows, HyperDbg halts the debuggee again and waits for the commands from the debugger.

B.3 Getting Debugging Events: #BPs and #DBs

HyperDbg uses the exception bitmap of VMCS to get notified of breakpoints (#BP) and Debug Breakpoints (#DB) to halt the other cores. HyperDbg is the first debugger capable of being notified about the debugging event which means that HyperDbg is notified even earlier than the operating system. Consequently, we design the system not to notify user-mode application or kernel-mode (OS) entities regarding the debugging events. So, all the breakpoints events are handled by HyperDbg.

B.4 Spinning on Spinlocks

Spinning the cores in HyperDbg is considered as a primary technique in its functionalities. We study the challenges in this context. Suppose a function requires a spinlock (e.g. it is merely a buffer which is to be accessed) in a single-core processor. The function raises the IRQL to *DISPATCH_LEVEL*. Here, the Windows Scheduler can not interrupt the function until it releases the spinlock and lowers the IRQL to *PASSIVE_LEVEL* or *APC_LEVEL*. If during the execution of the function, a VM-exit occurs, the operation mode is moved into the VMX-root. (It can be interpreted that a VM-exit happens similar to a HIGH_IRQL interrupt.)

In the case where a user accesses the buffer in the VMX-root mode, two scenarios are possible:

- The first scenario is to wait on a spinlock that was previously acquired by a thread in the VMX non-root mode. In such scenario, a deadlock occurs and spins forever.
- Alternatively, it is also possible to enter the function without looking at the lock (while there is another thread that enters the function at the same time), which would result in a corrupted buffer and invalid data. Windows also imposes another limitation, as cores must not wait on a spinlock when IRQL is higher than *DISPATCH_LEVEL*. This lies in fact that Windows raises the IRQL to (*DISPATCH_LEVEL*) 2, when a spinlock is acquired. In this case, Windows performs the workload, releases the spinlock and lowers IRQL back afterwards.

Looking at corresponding Windows functions (e.g *KeAcquireSpinLock* and *KeReleaseSpinLock*), the IRQL arguments are given as input. Windows saves the current IRQL to the parameter supplied by the user in *KeAcquireSpinLock*and then it raises the IRQL to *DISPATCH_LEVEL*. After the function is finished with the shared data, it calls *KeReleaseSpinLock* and passes the old IRQL parameter to the function. Finally, it unsets the bit and restore the old IRQL (lowering the IRQL).

Unfortunately, Windows spinlocks employs IRQLs which do not make sense when VMX-root mode is in action. This makes it very complicated to use such functions in this mode. Hence, to implement spinlock for HyperDbg functionalities such as multi-core message tracing, we design a custom VMX-root compatible spinlock.

B.5 MTFs Disadvantages

By setting the monitor trap flag, it is not necessarily guaranteed that the next instruction is the targeted instruction. In this case, if the upcoming instruction is a sudden interrupt from the CPU, the next targeted instruction in the debugging program would not be executed since the interrupt handler instructions are executed first. One way to address this issue is to set a *VM-exit* on exceptions (Exception Bitmap) and external-interrupts. However, this resolution is not optimal as it might causes system inconsistency by blocking interrupts. HyperDbg is able to resolve this issue using an instrumentation stepping process.

The following Listing illustrates the set/unset of MTF in an execution sequence.

```
/* Set the monitor trap flag */
void HvSetMonitorTrapFlag(BOOLEAN Set)
{
 unsigned long CpuBasedVmExecControls = 0;
 // Read the previous flag
 __vmx_vmread(CPU_BASED_VM_EXEC_CONTROL, &
     CpuBasedVmExecControls);
 if (Set) {
 CpuBasedVmExecControls |= CPU_BASED_MONITOR_TRAP_FLAG;
 }
 else {
 CpuBasedVmExecControls &= ~CPU_BASED_MONITOR_TRAP_FLAG;
 }
 // Set the new value
 __vmx_vmwrite(CPU_BASED_VM_EXEC_CONTROL,
     CpuBasedVmExecControls);
}
```

Listing 3: MTF Set/Unset in an example execution sequence

B.6 Debugger Pausing

Typically, there are two scenarios in which the kernel debugger is paused. A breakpoint is triggered either by a break request from an event or the script engine. In this context, if the user is in the kernel-mode, a *VMCALL* occurs and the future chain of events are handled accordingly. If the user is already in *VMX-root* mode, other cores should be notified to prevent a system-level crash. Operating in VMX-root mode is similar to HIGH_IRQL. In VMX-root mode, all the interrupts are masked because of RFLAGS' IF Bit.

The other scenario is upon the request from the user (for instance, an interruption by pressing CTRL+C). There, a packet is sent to pause the debugger. In this method, the debugger processes the packet in user mode, invoking an *IOCTL*, executing a *VMCALL* which transfers from the kernel-mode to the *VMX-root*.

B.7 Continuation a Single Core

One of the exclusive features of HyperDbg is to keep execution (continuation) on one core while other cores are in a halt-state. We used this mechanism in our instrumentation step-in command to guarantee that no other cores (threads) get the chance to be executed. The fundamental basis of this mechanism is to ensure that the target core is not interrupted during the debugging.

There are two approaches that HyperDbg uses to prevent a target core from getting interrupted (e.g. by clock interrupt or keyboard interrupts).

- First, the *RFLAGS.IF* bit of the guest can be unset so the interrupts are masked.
- Second, the *PIN Based External-Interrupt Exiting bit* can be set so all of the external interrupts would cause VM-exits; thus, allowing the interrupts to be simply ignored in the VM-exit handler

The former method is faster and avoids unnecessary VM-exits. However, for several considerations described in following, the second method is preferred in HyperDbg.

In contrast to the method described in the first approach, it is much safer not to change the guest's registers. As an example, if a page-fault, *SYSCALL*, or an invalid operation such as division-by-zero occurs, the execution is directed to the kernel and guest's *RFLAGS* are saved by the processor. Therefore, extra operations are required to locate the user-mode RFLAGS (search in stack for exceptions and in *R11 Register* for *SYSCALLs*), because the *RFLAGS* that was previously saved in user-mode is with IF bit disabled. If this specific task is ignored, *RFLAGS* are restored without checking for IF bit every time the guest continues and performs a context-switch. In this case, by unsetting this bit from hypervisor, the core becomes uninterruptible as the OS cannot get the execution again (e.g, using clock interrupt). Consequently, after a delayed bug check, Windows realizes the target core behaves abnormally and returns an error. Moreover, changing the guest's *RFLAGS* is also incompatible with instructions like *CLI* and *STI*. More importantly, considering the side effects, the guest is able to detect the tampering of HyperDbg using *PUSHF* function and check for *IF* bit in RFLAGS.

All of the issues investigated regarding *RFLAGS* changing, in addition to the fact that using PIN Based External-Interrupt Exiting bit is completely transparent from the kernel-mode and user-mode, has lead us to employ the second method in HyperDbg.

Table 4: List of the used terms and their brief description.

Term		Description
Abbreviation	Full Form	
VMX-root VMX non-root	Virtual Machine Extensions	Intel's Modes of Operations: A software on VMX-root mode has higher privileges, has access to certain instructions regardless of the privilege level
VM-Entries	-	Transition From VMX-root to VMX non-root
VM-Exits	-	Transition From VMX non-root to VMX-root
VMCS	Virtual Machine Control Structure	A hardware-defined structure to control the settings of each guest VM.
PB VM-Execution Controls	Processor-Based Execution Controls	Part of VMCS which control features and the vital attributes of the hypervisors
MTF	Monitor Trap Flag	At the VMCS, permits operation of a processor in single step mode in VMX-root. following a VMRESUME, a single instruction and then a VM-Exit occurs.
-	Exception Bitmap	An 32-bit field in the VMCS that controls the processor exceptions.
-	Event Injection	Events injected by Hypervisor (Interrupts, Exceptions, NMIs, and SMIs) and these events will be delivered to the guest as if they've arrived normally
NMI	Non-Maskable Interrupts	A hardware-based interrupt that cannot be ignored by the system.
SYSCALL	System Calls	System calls are a programmatic way that a user-mode application requests a service from the operating system's kernel.
IRQL	Interrupt Request Level	A hardware-independent mechanism MS Windows uses to prioritize interrupts and code that is running on the Processor.

C FULL DESCRIPTIONS OF THE USED TERMS

C.1 VM-entries, VM-exits (VMX-root and VMX non-root)

VT-x introduces two new modes of operations: VMX-root operation, and VMX non-root operation. A software running on VMX-root mode has higher privileges and has access to certain instructions that are not available in VMX non-root operation, regardless of the privilege level [76]. The core of HyperDbg runs mainly in the VMX-root mode, while guests (operating system's kernel, and applications) are executed in VMX non-root.

With definition of these two modes of operation, VT-x consequently defines two new transitions: one being VM-entry, which is a transition from the root operation to guest non-root operation, and the other VM-exit, which performs the opposite.

C.2 Virtual Machine Control Structure (VMCS)

To control the guest features, we have to set some properties in the Virtual Machine Control Structure (VMCS). The VMCS is a hardware-defined structure that controls the behavior and settings of each guest virtual machine (VM). Such a data structure exists for each VM in the memory and it is managed by the Virtual Machine Monitor (VMM). With every change of the execution context between different VMs, the VMCS is restored for the current VM, defining the state of the VM's virtual processor and this way, VMM controls the guest software.

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-Exit Control Fields**: Fields that control VM-exits.

- **VM-Entry Control Fields**: Fields that control VM-entries.
- **VM-Exit Information Fields**: Read-only fields to receive information on VM-exits describing the cause and the nature of the VM-exit.

C.3 Primary/Secondary Processor-Based VM-Execution Controls

Primary Processor-Based VM-Execution Controls [37], a part of VMCS, along with Secondary Processor-Based VM-Execution Controls fields [37, 38], control features that can be altered using the VMWRITE instruction.

These fields control some of the vital attributes that determine the behavior of the hypervisors [37, 38] such as Monitor Trap Flags, or Enabling EPT.

C.4 Monitor Trap Flag (MTF)

The Monitor Trap Flag or MTF (located in the VMCS) is a feature provided by Intel that works exactly like the Trap Flag in *RFLAGS*, which is additionally invisible to the guest. By setting this flag, following a VMRESUME, the processor is forced to execute a single instruction and then a VM-exit occurs.

C.5 Exception Bitmap

The exception bitmap is a 32-bit field in the VMCS that controls the exceptions (Exceptions are reserved by Intel on first 32 entries of Interrupt Descriptor Table).

When an exception occurs, its vector is used to select a bit in the exception bitmap. If the bit is 1, the exception will lead to a VM-exit. If the bit is 0, the exception is delivered normally through the IDT.

C.6 Event Injection

Hypervisors are able to inject events (Interrupts, Exceptions, NMIs, and SMIs) and these events are delivered to the guest as if they have arrived normally. Event injection is managed by special fields in VMCS.

C.7 Non-Maskable Interrupts (NMI)

An NMI is a hardware-based interrupt which typically cannot be ignored by the system. An NMI is usually triggered on hardware errors. These errors include non-recoverable internal system chipset errors, system memory corruptions such as parity and ECC errors, and data corruption detection on system and peripheral buses. Nevertheless, it is possible to mask specific NMIs by employing special techniques.

Some OS-level programs use debugging NMIs to diagnose, analyze, and fuzz codes. In such cases an NMI can execute an interrupt handler transferring the control flow to a special monitor program. In this circumstance, the developer can monitor the memory and examine the internal state of the program at the instant of its interruption. This also allows the debugging or diagnosing of computers which appear hung.

On some systems, a computer software could drive an NMI through hardware and software debugging interfaces and system reset buttons. HyperDbg as a hypervisor level software, employs NMI triggering extensively for kernel debugging.

C.8 Syscalls

Syscalls are a programmatic way that a user-mode application requests a service from the operating system's kernel. Syscalls provide an interface between a process and operating system and allow a user-level process to request a privileged service from the operating system.

In modern operating systems, syscalls are implemented using the SYSCALL and SYSRET instructions. These instructions rely on a set of Model Specific Registers (MSRs), namely *IA32_STAR*, *IA32_CSTAR*, and *IA32_LSTAR* MSR [73].

This mechanism can be turned on and off by setting and unsetting the SCE flag in the Extended Feature Enable Register (EFER). Making use of either SYSCALL or SYSRET with the SCE flag not set results in an invalid opcode exception.

C.9 Interrupt Request Level (IRQL)

An Interrupt Request Level (IRQL) is a hardware-independent mechanism that Windows uses to prioritize interrupts and code that is running on the processors. Processes running at a higher IRQL will preempt a thread or interrupt running at a lower IRQL [15].

The below list shows different routines and the corresponding IRQL that these routines are running on:

- **DIRQL** : Interrupt Service Routines (ISRs) of hardware and external devices
- **DISPATCH_LEVEL** : Scheduler, DPCs, and codes protected by a spinlock
- **APC_LEVEL** : Asynchronous Procedure Calls (APC) routines
- **PASSIVE_LEVEL** : User code, dispatch routines, and PnP routines

D FEATURE COMPARISON AMONG EXISTING DEBUGGERS

Table 5 illustrates a comparison among the existing debuggers and HyperDbg.

E HYPERDBG'S SCRIPT ENGINE

```
!syscall 0x55 pid 0x14c0 script {
    if(@rcx == 0x27 && @rdx == 0x47) {
        printf("Syscall triggered : %x in process id : %x
    \nThe third param : %llx\nThe fourth param : %llx\n"
    , @rax, $pid, @r8, @r9);
        pause();     }
        }
```

Listing 4: Script engine examples

In the above example, a syscall event is configured to trigger exclusively for syscalls specific to the process (pid = 0x14c0), which will execute the target script in VMX-root mode.

Considering that Windows uses fastcall calling convention for its syscalls, we know the registers stored in *RCX*, *RDX*, *R8*, *R9*, and stack. In the example script, it is checked if the first parameter to the syscall (*RCX* register) is equal to 0x27 and the second parameter (*RDX* register) is equal to 0x47. If these conditions are met, a message is printed, which generates a log from the 3rd (*R8* register) and 4th (*R9* register) parameters. At last, the `pause()` function is used to pause the debuggee and give the control to the debugger.

Table 5: A comprehensive comparison of HyperDbg with different debuggers

Debugger	Deployment Level	Debugging Mode		Transparency Insurance	Direct Deployment (NO VM/Emulator)	Source Code Available	Hooking	Scripting	Custom Assembly Execution	Notable Features (Currently Working Debuggers)
		User Mode	Kernel Mode							
HyperDbg	Hypervisor (Ring -1)	✓	✓	Hardware Assisted Methods (e.g. TimeStamp Emulation Transparency)	✓	✓	EPT Hidden Hooking	Customized VMX compatible ScriptEngine	✓	Different subsystems Fast script engine I/O debugging
Ghidra[20]	Application (Ring 3)	✓	✗	✗	✗	✓	✗	Python Scripting	✗	Advanced Decompiler Multi-platform Multi-architecture support
WinDbg[16]	Operating System	✓	✓	✗	✓	✗	✗	JavaScript WinDbg Script	✗	Javascript support Advanced scripting language
SoftIce[17]	Operating System	✗	✓	✗	✓	N/A	✗	✗	✗	Local debugging with special GUI
x64dbg[93]	Application (Ring 3)	✓	✗	Software-Based ScyllaHide	✓	✓	✗	Customized Scripting	✗	Flexible and strong GUI Lots of useful features
Ollydbg[71]	Application (Ring 3)	✓	✗	Software-Based ScyllaHide	✓	✗	✗	ODBGScript	✓	Stability
gdb [19]	Operating System	✓	✓	✗	✓	✓	✗	Bash Scripting Automation	✗	Main Linux Debugger Support multiple platforms
Malt[97]	SMM (Ring -2)	N/A	✓	SMM Level Transparency	✓	N/A	N/A	N/A	N/A	N/A
BareBox[49]	Hypervisor (Ring -1)	N/A	✓	Meta OS (Bare Metal) Transparency	✓	N/A	N/A	N/A	N/A	N/A
V2E[95]	Hypervisor (Ring -1)	N/A	✓	Hypervisor Level Transparency	✗	N/A	N/A	N/A	N/A	N/A
Anubis[63]	Hypervisor (Ring -1)	N/A	✓	Limited Software-Based Methods	✗	N/A	N/A	N/A	N/A	N/A
Virt-ICE[77]	Hypervisor (Ring -1)	N/A	✓	Emulating Software-Based Methods	✗	N/A	N/A	N/A	N/A	N/A
HyperDbg (deprecated)	Hypervisor (Ring -1)	✓	✓	N/A	✗	✓	✗	✗	✗	N/A
Ether[25]	Hypervisor (Ring -1)	✓	✓	Hypervisor Level Transparency	✗	N/A	N/A	N/A	N/A	N/A
VAMPiRE[86]	Hypervisor (Ring -1)	N/A	✓	Hypervisor Level Transparency	✓	N/A	N/A	N/A	N/A	N/A
SPIDER[23]	Hypervisor (Ring -1)	N/A	✓	Hypervisor Level Transparency	✗	N/A	N/A	N/A	N/A	N/A
IDAPro[40]	Application (Ring 3)	✓	✗	✗	✓	✗	✗	Built-in Scripting Engine (IDC / Python)	✗	Advanced decompiler Multi-architecture Multi-platform support

F HARDWARE FEATURES ON ARM AND AMD

ARM processors also contain virtualization extensions that provide hardware means for hypervisors to virtualize the CPU, permitting multiple OSes to be run on the same machine. ARM processors support the Second Level Address Translation (SLAT), which is known as Stage-2 page tables provided by a Stage-2 MMU.

Similarly, AMD supports virtualization through AMD-v technology. SLAT implementation in AMD processors is through the Rapid Virtualization Indexing (RVI), or Nested Page Tables (NPT) technology.

With the similar approach it is most likely possible to investigate and implement the same methodologies presented in this work for AMD or ARM-based hardware-assisted debuggers.

G APPLICATION: REVERSE ENGINEERING

G.1 Automatic Symbol Reconstruction

One of the main goals of HyperDbg is to provide a reverse engineering and dynamic binary analysing tool. Here, we describe a simple example of reverse-engineering an application. HyperDbg provides a functionality that maps a virtual memory to a C/C++ data type (e.g., enums, and structures). This is extremely useful, as OS-level PDB linkers can dynamically be translated to structures for testing and reverse engineering. For instance, a user can exactly detect the location of values in an specific function used by OS process and modify it accordingly. This is easily done using simple scripting. The following listing shows how memory content of target process can be easily delivered by structured representation. For instance, here _PROCESS_CREATION_INFO is de-referenced and could be modified.

```
kHyperDbg>   dt nt!_EPROCESS ffff948cc2393080
_EPROCESS
 ...
+0x05b7   uint8_t PriorityClass : 0x2
+0x05b8   void* SecurityPort : (null)
+0x05c0  _SE_AUDIT_PROCESS_CREATION_INFO ... : ffff948c‘c25fa2a0
+0x05c8  _LIST_ENTRY JobLinks ... : [ 00000000‘00000000 -
     00000000‘00000000 ]
```

Listing 5: Converting PDB references to structures for reverse engineering