

# HYPERDBG: Reinventing Hardware-Assisted Debugging

Mohammad Sina Karvandi  
Institute For Research In Fundamental  
Sciences  
Iran

MohammadHossein  
Gholamrezaei\*  
Chosun University  
Republic of Korea

Saleh Khalaj Monfared  
Worcester Polytechnic Institute  
United States

Soroush Meghdadizanjani  
Stony Brook University  
United States

Behrooz Abbassi, Ali Amini  
HyperDbg Organization  
Republic of Korea

Reza Mortazavi  
Damghan University  
Iran

Saeid Gorgin\*  
Chosun University  
Republic of Korea

Dara Rahmati\*  
Shahid Beheshti University  
Iran

Michael Schwarz  
CISPA Helmholtz Center for  
Information Security  
Germany

## 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,<sup>†</sup> we present 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 Table (EPT). 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.

## CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; **Virtualization and security**; **Software security engineering**; • **Software and its engineering** → *Compilers*.

## KEYWORDS

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

### ACM Reference Format:

Mohammad Sina Karvandi, MohammadHossein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadizanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. 2022. **HYPERDBG: Reinventing Hardware-Assisted Debugging**. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*, November 7–11, 2022, Los Angeles, CA, USA. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3548606.3560649>

## 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 [19, 22, 71, 82, 90]. 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 [93], evasion [28], and hardware-assisted techniques [45, 69], employ a series of methods that involve anti-virtualization [5], anti-emulation[51], 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 [25, 26, 68, 75, 92, 94], currently available debugging

\*Work performed while at Institute For Research In Fundamental Sciences.

<sup>†</sup>For the extended version of this paper which includes additional appendices, refer to: <https://arxiv.org/abs/2207.05676>.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-9450-5/22/11...\$15.00

<https://doi.org/10.1145/3548606.3560649>

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 [9]. Using OS APIs [11] or ring-0 options [34] leads to artifacts 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 [46, 88, 94], hypervisor-level (VT-x) [25, 30, 49, 89], System Management Mode (SMM) [94], or even Intel Memory Management Engine (MME) [29] 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 [95] 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 [20, 30], developed for pure academic purposes [30], or have not been through thorough development and testing required for dealing with real-world applications and scenarios [95]. 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 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 hard to detect with time-delta methods that detect the presence of hypervisors, e.g., by detecting the overhead of traps into the hypervisor [60, 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 limitations previously imposed by *Hardware Debug Registers* in size and count [94]. 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, analyzes 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>.

## 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, 48]. 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* [90], *Ollydbg* [71], and Immunity Debugger [40] 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* [19] and *GDB* [22] are famous examples of kernel debuggers that are widely used for reverse engineering and malware analysis [1]. With advances in malware evasion techniques [5], researchers have been showing interest towards virtualization, simulation, and hardware-assisted debugging methods [50] that can offer a more transparent environment for code analysis and low-level modification of the execution flow [68, 94].

## 2.2 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.

*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 [31] 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 [84], EPT eliminates the overhead associated with software-managed shadow page tables [39]. 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 a set of x86 instruction set extensions, including Restricted Transactional Memory (RTM), which adds support for the declaration of hardware transactions. Instructions inside such a transaction either all succeed or are rolled back altogether, in case any error or abortion occurs during the transaction, e.g., interrupt from OS [42, 81]. In this paper, by using the term Intel TSX, we refer to RTM specifically.

## 2.3 Terminology

As the implementation here is based on *Intel processors* and the target OS is *Microsoft Windows*, we describe the low-level design of our system based on Intel and Windows terminologies.

*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 [15, 21, 73]. It abstracts guest machines and the operating system from the actual hardware and runs virtual machines (VMs).

*Interrupt Request Level (IRQL)*. An Interrupt Request Level (IRQL) is a hardware-independent mechanism that Windows uses to prioritize interrupts and code. Processes running at a higher IRQL preempt a thread or interrupt running at a lower IRQL. [18].

There are different IRQLs used for different things. *DIRQL* is for interrupt service routines (ISRs) of hardware and external devices, *DISPATCH\_LEVEL* is used for the scheduler, DPCs, and code protected by spinlocks, *APC\_LEVEL* is for asynchronous procedure call (APC) routines, and *PASSIVE\_LEVEL* for user code, dispatch routines, and PnP routines.

## 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 designed to perform 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 is 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/disassembler 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 they are (dis)assembled and parsed through the script engine in ①. 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 ④. 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 ②. Finally, The sub-system functionalities utilize hardware-based features (e.g., EPT) to execute their operation in ③. 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 host and guest. However, as shown in the figure, with the use of the script engine, HYPERDBG can confine the communication in an automated routine within the guest kernel mode if necessary.

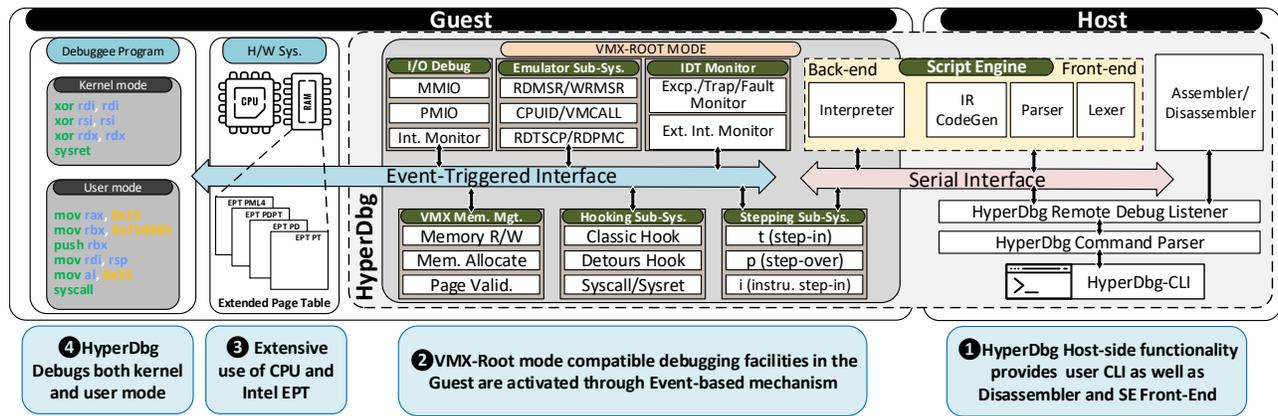


Figure 1: High-level overview of HYPERDBG's sub-systems and execution flow

## 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 to monitor, to access to a particular memory address. HYPERDBG can be configured to perform arbitrarily defined actions upon the occurrence of each event.

**3.2.2 Actions.** Upon having an event triggered, HYPERDBG can invoke 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 micro-architectural levels. While the adversarial dynamic between malware generators and anti-malware producers is a never-ending process and this mode does not guarantee 100% transparency, it makes it substantially more challenging for the anti-debugging and anti-hypervisor methods to detect the debugger. It is noteworthy to mention that HYPERDBG is already immune to high-level anti-debugging methods that rely on API-specific methods to detect debugging environments (e.g., self-debugging binaries). 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

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. 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* [19], *GDB* [22]) 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 clearing the Interrupt Flag in *RFLAGS*. However, intercepting/preventing the interrupts can easily break the OS semantics. 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, Non-Maskable Interrupts (NMIs) are used to ensure that the execution is done on 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 our 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 on only one core and disables interrupts on the same core (ignoring external interrupts by setting the 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 through 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. 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 triggering an undefined opcode exception (#UD) (by clearing 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 arbitrary scripts and set hooks for arbitrary system calls 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 EPT-level hooks that are not visible to the user-mode program or the operating system when attempting to read the hooking address. The first type of hidden breakpoints in HYPERDBG are Classic EPT Hooks, which are achieved by injecting 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. The second variation of hidden hooks utilizes Detours-Style Hooks [10] (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 the 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 [13].

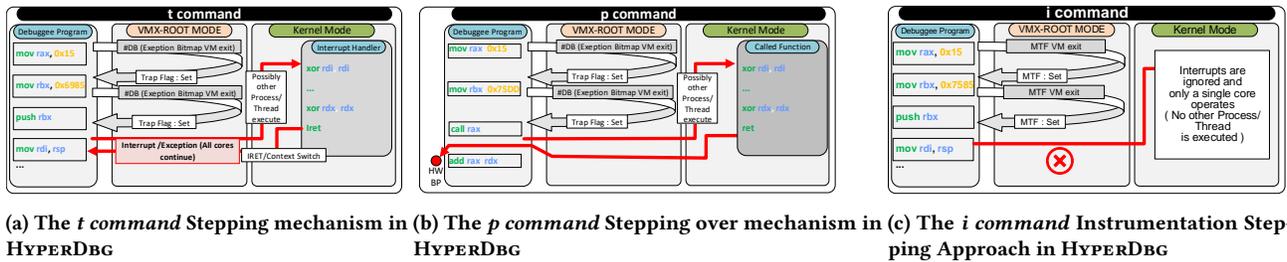


Figure 2: HYPERDBG Stepping Commands

### 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 [37] pages in the VMX-root [43], 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 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 TSX. TSX suppresses exceptions/faults without any switch between user/kernel modes. 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. [79] 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.

**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).

```

1 ; Use Intel TSX to suppress any
2 ; page-fault in VMX-root mode
3 XBEGIN $+xxx ; End of TSX
4 MOV RAX, Dword PTR:[RCX]
5 ; Access the target memory address ,
6 XEND ; End of TSX
7 MOV RAX, 1
8 JMP Return
9 MOV RAX, 0
10 Return :
11 RETN ; Return the result

```

Listing 1: Using Intel TSX to detect address validity.

**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 [76].

**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 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 [43] 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 debuggee is in VMX non-root mode.

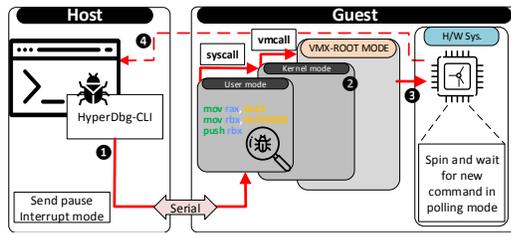


Figure 3: The overall view of the communication in HYPERDBG

## 5 FRONT-END ARCHITECTURE

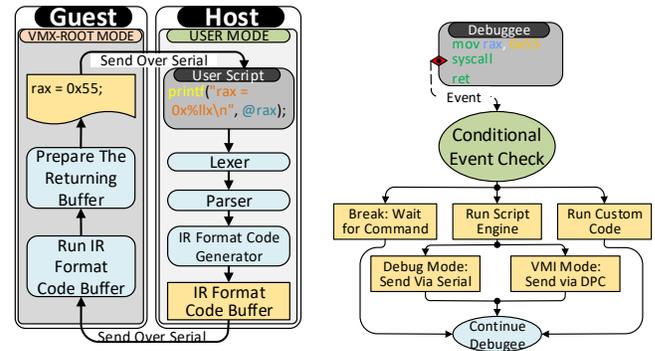
In the following section, we explore the intermediary components of HYPERDBG’s connecting back-end VMX-root mode sub-systems with the host machine as well as the user-interface functionality. Specifically, We describe guest-host communication and the kernel-level script engine. Although the core functionality 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.

### 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 IRQ 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. In addition to the serial communication, which is the conventional mode of communication in HYPERDBG, KDNet functionality [63], which is publicly available from Windows SDK [4], has also been incorporated in HYPERDBG as an alternative and more modern means of communication.

**5.1.1 Sending Data over Serial.** Following the connection initialization between a serial device and its corresponding 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) [23], 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.



(a) HYPERDBGs script engine’s (b) Script Engine invocation as an execution flow event

Figure 4: Description of Script Engine in HYPERDBG

### 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 scratch. To the best of our 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.

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.

## 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 hardening 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 [86] (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. 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 modify timing footprints used by analyzer software to detect virtualized environments. By using a statistical model of the execution time, extensive timing profiling is executed prior to the launch of the VMM module in order to emulate timestamps as close to the normal operating condition of the guest OS as possible.

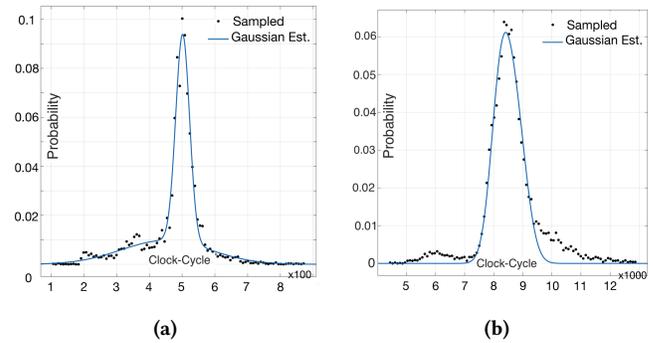
**Table 1: Anti-Debugging and Anti-VM exercises and mitigation in HYPERDBG**

Cat.	Methodology	Example of the Meth.	Example	Mitigation in HyperDbg	
Anti-Debugging and Fingerprinting Methods	API-Call (System-Call)	GetCurrentProcessId() CreateToolhelp32Snapshot() Process32Next() NtQueryInforProc() FindWindow()	[9, 56]	Modify results via EPT-Hook (hiding process)	
	PEB Field	IsDebuggerPresent() NtGlobalFlags()	[53]	HyperDbg is not detectable by default	
	Heap Structure	HEAP.Flags HEAP.ForceFlags	[44]	HyperDbg is not detectable by default	
	#BP Detection	Find BP (0xCC) inst. Read DR (Debug Register)	[57]	!dr to modify and disable unwanted BPs	
	Timing Measurement	GetTickCount(), QueryPerfCounter, GetLocalTime()	[58]	EPT-Hook Modification of results	
	Trap-Interrupt	Instruction Prefix, INT 3, 0x2D, Interrupt 0x41	[52]	Set Exception bitmap in VMCS	
	Control Flow Manipulation	NtSuspendThread(), NtSetInf.Thread(), CreateThread()	[54]	HyperDbg not detectable by default	
	Anti-VM/Hypervisor/Emulation	CPU Instructions	CPUID forces a VM-exit certain info in VM	[61]	VM-exit (CPUID result modification)
		Protection Model Instructions	SIDT, SLDT, SGDT STR, SMSW	[9]	VM-exit (emulation and modification)
		Architectural Delta-Timing	RDTSC+CPUID+RDTSC RDTSC(P)+RDTSC(P)	[61]	HyperDbg Trans. Mode (!hide command)
In/Out Instructions		Magic I/O port in VMware	[24]	VM-exit handled (I/O bitmap)	
Invalid MSR Access		Invalid MSR issues General Protection (#GP) Try-Catch	[95]	Emulate !msrread/!msrwrite command	
Exception Handling		General Protection Excep. (#GP)	[55]	Handled by default Inject routine into user-mode	

```

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

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



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

We use 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.

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 also rely 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).

## 6.3 Alternative Timing Attack Methods

In addition to the methods explained in 6.2, HYPERDBG can impede various forms of timing attacks used by malware to expose the presence of a debugger. Since HYPERDBG operates at the hypervisor

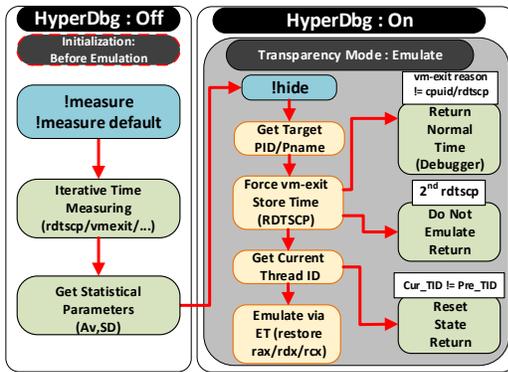


Figure 6: State Diagram Process of *rdtsc/rdtscp* emulation by HYPERDBG

level, it is possible to use HYPERDBG to safeguard against other timing attacks that utilize alternative timing resources, such as shared ticks and Hardware Performance Monitor Counters (RDPMC), both of which have built-in support to defend against in HYPERDBG’s Transparent Mode. Additionally, while timing-thread attacks are one of the more challenging attacks to defend against, it is possible to detect such attempts using any known timing resources using HYPERDBG. HYPERDBG is actively adding methods that safeguard against new techniques used by malware for debugger detection.

## 7 EVALUATION

In this section, we thoroughly evaluate HYPERDBG’s transparency and performance in different scenarios.

### 7.1 Transparency Evaluation

We evaluate the transparency mode of HYPERDBG using two of the best-known tools that offers stress-testing for anti-debugging and protection methods, *pafish* [3] and *al-khaser* [65].

In accordance with our expectations, the first method, which involves updating the `IA32_TIME_STAMP_COUNTER`, interferes with the primary functions of the system and causes screen flickering during our experiments. Regardless, the second method (emulation) was able to successfully pass these tools when enabling emulation for the anti-debugging testing software. As an extension of our transparency analysis, we separately evaluate HYPERDBG against common anti-debugging methods and commercial off-the-shelf packers/protectors.

**7.1.1 Evaluation Configuration.** In our experiments, we analyzed 10,853 malware samples in different categories derived from a malware database [85]. Each of these malware samples is executed in HYPERDBG’s normal and transparent mode in the VMI Mode as well as *x64dbg* (user-mode debugger) and *WinDbg* (kernel-mode debugger) in Microsoft Windows 10 20H1 for comparison. We employ a client/server paradigm to distribute the samples among the client systems that execute the binaries and record logs from the executions. Each client asks the server for its according sample over a simple HTTP application. We use two approaches to restore the system: a rebootless, Barebox-based method [46] and an automated system restoration method based on Windows System Restore.

**Barebox-based Approach** We first attempt to run the malware under a setup that aims to rebootlessly restore the system using a set of Barebox-based methods to decrease the restoration time and improve performance. After fetching the malware from the server, the client globally disables interrupts and continues the main thread on a single core. While this can impose some performance penalties and communication issues with external devices, by removing the context switch to other processes, it simplifies the system restoration process and allows for observation of the effects of malware on the system exclusively, as it prevents unnecessary modifications to the structures unrelated to the subject binary.

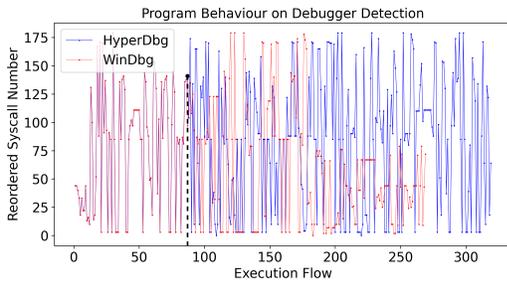
Next, the automation program loads the binary and locates its entry point by capturing the page fault that follows in the initialization phase of the execution of a binary in Windows. A hardware breakpoint is then applied in this address (entry-point), which allows HYPERDBG to be notified at the beginning of the program’s execution, once Windows is done with the initialization process of executing the binary. HYPERDBG uses this to trigger the mechanism used for rebootless restoration of the contents of the memory.

To increase the performance of the testing process by limiting the restoration only to the modified pages, we make a snapshot of the clean installation of Windows by disabling writing on pages, which is accomplished by clearing every *write* bit on EPT pages. This results in an EPT violation for a write request. We use the handler of this violation to make a clone of the pages that the malware attempts to write and then release the write lock to allow the binary to continue with its normal execution. While at the beginning of the execution, this can decrease the performance and sometimes make the system unresponsive, the performance improves as the execution progresses. To mitigate any resulting artifacts, HYPERDBG is set to run in Transparent Mode.

Once the execution is finished, and the logs are saved using the VMX-root Mode Compatible Message Tracing Mechanism (Section 4.3.3), the interrupts are re-enabled, and our master program updates the order of runs and restores the system. To restore the information on the volatile memory, we replace the pages that were cloned during the execution process back to their original state, while for disk writes, we use Shadow copy [64] combined with a mini-filter driver that monitors the modification/creation/deletion of the memory/registry. This process takes around 7 minutes per malware sample.

If the system cannot be restored with this approach, we resort to the second approach, which requires a reboot and is based on Windows Restore Point. These cases include restoration of the system for bare metal executions, rootkit/bootkit samples, and cases where the execution encounters an error. A malware is classified as rootkit/bootkit using the tags provided by the malware providers and by detecting any attempt to load drivers. The success of the execution is measured using a Win32 API call [62].

**Windows System Restore Approach** As an alternative approach, we rely on the Windows System Restore functionality. As there is no command-line tool for restoring snapshots, we implement a small tool to use the GUI tool automatically. This method takes about 25-35 minutes to restore the system, based on the state of the system and the modifications made by the malware.



**Figure 7: Syscalls executed in a malware using HYPERDBG and WINDBG**

Overall, the testing process takes about 468 hours (168 hours testing malware with HYPERDBG and 100 hours each for testing bare-metal, *x64dbg*, and *WinDbg*). We use 10 systems for this test.

**7.1.2 Evaluation by Anti-Debugging, -VM, and -Hypervisor.** Table 1 describes the common anti-debugging and anti-virtualization methods [1, 94], and HYPERDBG’s countermeasure to impede detection. Each method 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 employ a combination of rebootless and reboot-based approaches for system restoration after the execution of each malware sample.

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 Figure 8a, HYPERDBG’s Transparent mode increases the transparency surface by 22% compared to *WinDbg*, executing malware samples in all four categories while remaining undetected. This is due to the fact that HYPERDBG operates at the hypervisor-level, which minimizes footprints that anti-debugging/VM methods in malware use to detect a debugging environment. We manually investigated malware samples that detect the presence of HYPERDBG and reverse engineered the binaries to inspect their inner-workings. We could attribute the detection of HYPERDBG to two main factors. Employment of hypervisor-specific techniques that lead to the non-successful execution of malware in already virtualized environments, and utilization of methods that detect the absence of PatchGuard, or Driver Signature Enforcement (DSE) in the system. These methods can be countered by adding support for nested virtualization (cf. 10) and obtaining a valid driver signature. However, this does not imply a claim of full invisibility upon the addition of said improvements. For example, HYPERDBG is a new and open-source tool and it is possible for malware producers to focus on the stealthiness methods employed in this tool and find new methods to counter those efforts to reveal its presence.

**7.1.3 Syscall Malware Analysis.** The transparent mode of this debugger offers the advantage of stealthily monitoring malware’s

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 attach a kernel-mode debugger (*WinDbg*) to the malware, execute the malware, and record the syscall execution flow. We use a simple script code using HYPERDBG’s script engine to trace the `SYSCALLs` in the execution flow. 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 conclusion that this specific malware uses anti-debugging methods. To conceal its malicious intention, an entirely different (and most likely 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, as shown in Figure 8a.

**7.1.4 Evaluation by Packers and Protectors Testing.** We test HYPERDBG with binaries processed with packers and protectors. These binaries are tested on different debuggers as well as HYPERDBG in both regular debugging and transparent mode debugging. Table 2 shows the results of debugging these protected binaries.

## 7.2 Performance Evaluation

In terms of performance, we analyze HYPERDBG in three debugging scenarios which are discussed in the following. For our performance evaluations, we used a machine with an Intel core i7-6820HQ with 16GB of main memory, running Windows 10 20H1.

**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 ( $\mu = 411$  seconds,  $\sigma = 28.3$  seconds). It took *WinDbg* on average 1,221 seconds ( $\sigma = 118.4$  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*. For a fair comparison with *WinDbg*, we also use Windows SDK’s *KDNET*

**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.V2.13	PE64	✗	✗	N/A	✗	✓
MEW11.V1.2	PE32	✓	✓	✓	✓	✓
Pecomact.V3.11	PE32	✓	✓	✓	✓	✓
PELock.V2.0	PE32	✗	✗	✗	✓	✓
Petite.V2.4	PE32	Error	✓	✓	✓	✓
TelLock.V0.98	PE32	✓	✓	✓	✓	✓
YodaCrypter.V1.02	PE32	✗	✗	✗	✓	✓

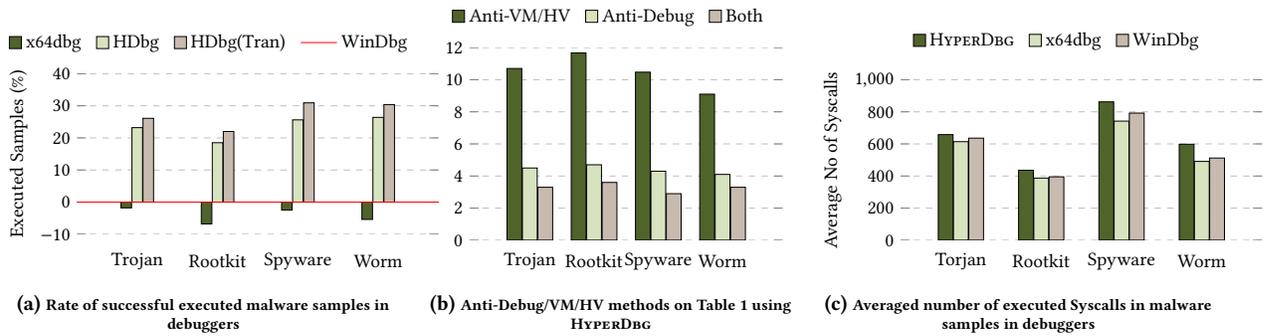


Figure 8: Evaluation results of malware execution experiments

as a transport for HYPERDBG’s Ethernet-based communication. In this scenario, HYPERDBG is 2.85 times faster.

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

To evaluate the performance of HYPERDBG on conditional breakpoints, we set one on the frequently used `nt!ExAllocatePoolWithTag` 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 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  $\sim 2018x$  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 from 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). As HYPERDBG resides on a more privileged ring than the kernel, it can intercept code-level exceptions that lead to the crash of the

system (application/OS) (e.g., page faults or division by zero) and discard the crash before calling the OS error handling routines.

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 7.1.3. 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* [14], 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 can still infect 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 over 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 other 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 [17, 42].

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 a user-mode analysis tool [7, 8, 38].

HYPERDBG can detect page faults in both the operating system and applications. In previous works, Shadow Paging, Page Tracking, and Pseudo-paging methods were used to detect page-faults [38]. 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 [12] 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. [33] where over 92 classes of evasive techniques executed by modern malware.

Furthermore, hardware-based artifacts such as processor's cache actualization [74], scheduling leakage in simultaneous multithreading (SMT) [59], 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 [80] to hide debugging procedures from malware. Likewise, other work proposes resilient malware detectors against evasive malware using hardware features [41, 83]. Leon et al. [50] 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 [87] and Cuckoo sandbox [32] to Bare-Metal sandboxes like BareBox [46], and BareCloud [47] to minimize the leakage of the virtualization environment. Pioneered by Ether [26] as the first hypervisor-based analyzer with more transparency level, Malt [94] 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 [77, 78], kernel failure reverse debugging [36] as well as machine learning approaches [6], to discover vulnerabilities and bugs. Similar ideas are deployed for embedded systems arming application tracing [27], debugging [67, 70], unpacking [91] on Arm processors.

Though designed as debugger, HYPERDBG delivers high-level 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.

## 10 DISCUSSIONS AND LIMITATIONS

**Transparency** Several timing attacks for detecting HYPERDBG are impeded. Still, by relying on external timing resources such as

NTP, a binary can leverage measurement methods beyond the domain of the local system to detect HYPERDBG's timing adjustments. Nonetheless, these methods for debugger detection can potentially be mitigated in a case-specific analysis of a binary; however, the development of a general solution for this set of approaches is considered out of scope in this paper.

**Stability** Direct emulation of system-wide general-purpose mechanisms (such as modifying MSRs, cf. Section 6.2) might interfere with the normal functionality of other applications that use timing measurements. Thus, we recommend using the second method (e.g., emulation of RDTSC and RDTSCP instructions) for already up and running test-bed environments. Furthermore, the Barebox-based restoration process requires manual restorations in some scenarios where Windows uses Asynchronous Procedure Calls (APC) as the inter-processor communication is disabled in such test-cases [16]. We plan to develop a general testing framework and release it publicly as part of the HYPERDBG project.

**Performance** Although faster than similar debuggers, our experiments show cases where HYPERDBG faces slowdowns due to an excessive number of VM-exits. To minimize the overhead, unnecessary VM-exits can be reduced, and emulation of the system at the early stage of VM-exits can be avoided to improve the overall performance of the system. For instance, the user might specify a specific core to apply the events.

**Future Works** In future releases, we intend to make a UEFI-based module for HYPERDBG, which permits HYPERDBG to run compatibly with Windows Virtualization-based Security (VBS). We also plan to add support for nested virtualization in HYPERDBG to support the execution of nested guest VMs inside the virtualized environment of HYPERDBG.

## 11 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 modular and scalable for convenient usage in academia and industry.

## ACKNOWLEDGMENT

The authors would like to thank Daniel Gruss and Kamyar Givaki for their guidance and support on the initial version of the paper.

## 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/aOrtega/pafish>. Accessed: 2022-02-15.
- [4] Ortega Alberto. 2022. Windows SDK. <https://developer.microsoft.com/en-us/windows/downloads/windows-sdk>. Accessed: 2022-08-11.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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)*.
- [9] 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.
- [10] Doug Brubacher. 1999. Detours: Binary interception of Win32 functions. In *Windows NT 3rd Symposium (Windows NT 3rd Symposium)*.
- [11] 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.
- [12] 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.
- [13] Tzi-cker Chiueh. 2008. Fast bounds checking using debug register. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 99–113.
- [14] Citeseer. 2019. PITOU : Kernel Payload and DGA. (2019).
- [15] IBM Co. 2019. Hypervisors. <https://www.ibm.com/cloud/learn/hypervisors>. Accessed: 2022-02-15.
- [16] IBM Co. 2021. Asynchronous Procedure Calls. <https://docs.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls>. Accessed: 2022-08-06.
- [17] Mitre Co. 2022. Access Token Manipulation. <https://attack.mitre.org/techniques/T1134>. Accessed: 2022-02-15.
- [18] 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.
- [19] Microsoft Co. 2022. Windows Debugger (WinDbg). <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools>. Accessed: 2022-02-15.
- [20] NuMega Co. 2022. SoftIce. [https://www.angelfire.com/bug/ass\\_1/Readme.htm](https://www.angelfire.com/bug/ass_1/Readme.htm). Accessed: 2022-02-15.
- [21] VMWare Co. 2022. What is a hypervisor? <https://www.vmware.com/topics/glossary/content/hypervisor>. Accessed: 2022-02-15.
- [22] GNU Community. 2022. GDB. <https://www.gnu.org/software/gdb/>. Accessed: 2022-02-15.
- [23] Intel Corporation. 2018. Intel 64 and ia-32 architectures software developer manuals.
- [24] Cyberbit. 2022. Anti-VM and Anti-Sandbox Explained. <https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/>.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] Mark Ermolov and Maxim Goryachy. 2019. Intel VISA: Through the Rabbit Hole. *Black Hat Asia* (2019).
- [30] 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.
- [31] 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.
- [32] Cuckoo Foundation. 2022. Cuckoosandbox. <https://cuckoosandbox.org/>. Accessed: 2022-02-15.
- [33] 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.
- [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] Steven M Hand. 1999. Self-paging in the Nemesis operating system. In *OSDI*, Vol. 99. 73–86.
- [38] 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.
- [39] 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>
- [40] Immunity Inc. 2022. immunitydbg. <https://www.immunityinc.com/products/debugger/index.html>. Accessed: 2022-02-15.
- [41] 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).
- [42] 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.
- [43] Sina Karvandi. 2019. Hypervisor From Scratch – Part 7: Using EPT & Page-Level Monitoring Features. (2019). <https://rayanfam.com/topics/hypervisor-from-scratch-part-7/>
- [44] Jong-Wouk Kim, Jiwon Bang, and Mi-Jung Choi. 2020. Defeating Anti-Debugging Techniques for Malware Analysis Using a Debugger.
- [45] 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.
- [46] Dhillung 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.
- [47] Dhillung 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.
- [48] Andrew Ko and Brad Myers. 2008. Debugging reinvented. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 301–310.
- [49] 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.
- [50] Roece S Leon, Michael Kiperberg, Anat Anatey Leon Zabag, and Nezer Jacob Zaidenberg. 2021. Hypervisor-assisted dynamic malware analysis. *Cybersecurity* 4, 1 (2021), 1–14.
- [51] Cătălin Valeriu Liță, Doina Cosovan, and Dragoș Gavriluț. 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.
- [52] Check Point Software Technologies LTD. 2022. Anti-Debug: Assembly instructions. <https://anti-debug.checkpoint.com/techniques/assembly.html>.
- [53] Check Point Software Technologies LTD. 2022. Anti-Debug: Debug Flags. <https://anti-debug.checkpoint.com/techniques/debug-flags.html>.
- [54] Check Point Software Technologies LTD. 2022. Anti-Debug: Direct debugger interaction. <https://anti-debug.checkpoint.com/techniques/interactive.html>.
- [55] Check Point Software Technologies LTD. 2022. Anti-Debug: Exceptions. <https://anti-debug.checkpoint.com/techniques/exceptions.html>.
- [56] Check Point Software Technologies LTD. 2022. Anti-Debug: Misc. <https://anti-debug.checkpoint.com/techniques/misc.html>.

- [57] Check Point Software Technologies LTD. 2022. Anti-Debug: Process Memory. <https://anti-debug.checkpoint.com/techniques/process-memory.html#software-breakpoints>.
- [58] Check Point Software Technologies LTD. 2022. Anti-Debug: Timing. <https://anti-debug.checkpoint.com/techniques/timing.html>.
- [59] 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.
- [60] Gary McGraw and Greg Morrisett. 2000. Attacking malicious code: A report to the infosec research council. *IEEE software* 17, 5 (2000), 33–41.
- [61] Christian Rossow Michael Brengel, Michael Backes. 2010. Detecting Hardware-Assisted Virtualization. (2010).
- [62] Microsoft. 2022. GetExitCodeProcess function. <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodeprocess>. Accessed: 2022-02-15.
- [63] Microsoft. 2022. KDNNet. <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection-automatically>. Accessed: 2022-02-15.
- [64] Microsoft. 2022. Volume Shadow Copy Service. <https://docs.microsoft.com/en-us/windows-server/storage/file-server/volume-shadow-copy-service>. Accessed: 2022-02-15.
- [65] mrexodia. 2022. al-khaser. <https://github.com/LordNoteworthy/al-khaser>. Accessed: 2022-02-15.
- [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] 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.
- [74] 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.
- [75] Nguyen Anh Quynh and Kuniyasu Suzuki. 2010. Virt-ICE: Next-generation debugger for malware analysis. *Black Hat USA* (2010).
- [76] 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.
- [77] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *NDSS*.
- [78] 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.
- [79] 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.
- [80] Hao Shi and Jelena Mirkovic. 2017. Hiding debuggers from malware with apate. In *Proceedings of the Symposium on Applied Computing*. 1703–1710.
- [81] Julian Stecklina and Thomas Prescher. 2018. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).
- [82] LLDB Team. 2022. LLDB. <https://lldb.lvm.org/>. Accessed: 2022-02-15.
- [83] 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.
- [84] VMware Inc. 2019. Performance Evaluation of Intel EPT Hardware Assist. [https://www.vmware.com/pdf/Perf\\_ESX\\_Intel-EPT-eval.pdf](https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf). Accessed: 2022-02-15.
- [85] vx underground. 2021. vx-underground malware collection. <https://www.vx-underground.org/> 1 (2021).
- [86] 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.
- [87] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy* 5, 2 (2007), 32–39.
- [88] 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.
- [89] Carsten Willems, Ralf Hund, and Thorsten Holz. 2013. Hypervisor-based, hardware-assisted system monitoring. In *Virus Bulletin Conference*.
- [90] x64 Debugger. 2022. X64. <https://x64dbg.com/>
- [91] 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.
- [92] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 227–238.
- [93] 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.
- [94] 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.
- [95] 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.