HyperDbg Debugger

A debugger designed for analyzing, fuzzing and reversing

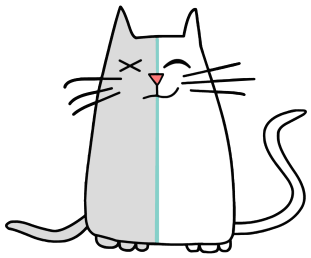
Version v0.10

Gaining Insights: Exploring Fresh Reverse Engineering Techniques

Research:
`https://research.hyperdbg.org`

Written by
Sina Karvandi
Soroush Meghdadizanjani
Saleh Khalaj Monfared

August 7, 2024



This article is about the new features to be introduced in the recent and future versions of HyperDbg, offering insights into their design/challenges and how they can benefit our reverse engineering endeavors by adding new reversing techniques.

---

1 Introduction

It has been four years since we released the HyperDbg debugger,[4] a free, open-source, and community-driven kernel-mode Windows debugger aiming to enhance and ease reverse engineering by bringing innovative hardware capabilities into the field.

Since then, we have continuously worked on refining the debugger to ensure its stability and broaden its compatibility with various processors and Windows versions. Additionally, we have been actively introducing novel techniques to provide reverse engineers with enhanced capabilities. Here's an introduction to these features.

2 Background

In this section, the preliminary background relevant to the study is discussed. Fundamental concepts such as hypervisors and second-layer paging (EPT or NPT) are discussed to establish a foundational understanding of the technologies central to our research. Additionally, the context for their significance in modern computing environments is providing, laying the groundwork for the subsequent discussion.

2.1 Hypervisors

Hypervisors, also known as virtual machine monitors (VMMs), are crucial for virtualization technology, enabling the creation and management of virtual machines (VMs) on physical hardware. They abstract hardware resources, allowing multiple VMs to run concurrently on a single machine. There are two types: Type 1 (bare-metal) and Type 2 (hosted). Examples include VMware vSphere/ESXi (Type 1) and Oracle VirtualBox (Type 2).

2.2 Intel Virtualization Technology (VT-x)

Intel's VT-x provides hardware-level support for virtualization, enhancing the efficiency and performance of virtualized environments. It introduces two modes of operation within x86-64 processors:

- **VMX-root Mode**: This mode is reserved for the Virtual Machine Monitor (VMM), allowing it to execute privileged instructions and modify the Virtual Machine Control Structure (VMCS). When a VM-exit occurs in the guest, control is transferred to the VMM, granting it exclusive control over virtualization resources.
- **VMX non-root Mode**: In this mode, the guest operating systems (running in ring 0) and user-level applications (running in ring 3) operate. VMX non-root Mode restricts the execution of privileged VMX instructions within the guest. If an attempt is made to execute such instructions, a VM-exit occurs, transferring control back to the VMM, which manages the virtualization process.

VT-x enhances virtualization performance and security by providing a streamlined approach to handling virtualized environments, with clear separation between VMM and guest operating systems.

2.3 Second Layer Paging (EPT or NPT)

Second Layer Paging, known as Extended Page Tables (EPT) in Intel processors or Nested Page Tables (NPT) in AMD processors, enhances virtualization performance and security. It offloads virtual-to-physical address translation to hardware, reducing overhead. EPT/NPT support creates a separate set of page tables for each VM, maintained by the processor, facilitating direct translation of guest virtual addresses to host physical addresses. This feature, supported by Intel VT-x and AMD-V, and boosts performance and scalability and also improves security by controlling memory access permissions.

2.4 Page Modification Logging (PML)

Page Modification Logging (PML) is a hypervisor feature designed by Intel to monitor accessed memory pages within virtual machines (VMs).[3] When configured, PML maintains a buffer with 512 entries to track dirty pages.[7] Notably, PML adds a minimal power consumption overhead, contributing to efficient power usage in the hypervisors.[2] Moreover, PML facilitates faster live memory migration.

PML is used extensively in logging memory writes. However, it does not capture read accesses, potentially leading to incomplete results, especially when vital information is contained in memory reads.

3 New Debugging Terms

Now that we see a couple of basic hypervisor-based concepts, it is time to familiarize ourselves with new debugging terms introduced in HyperDbg. These terms will provide us with the necessary background of how we can perform innovative techniques using these features.

3.1 Events

HyperDbg is an event-driven debugger. 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. Almost all of the HyperDbg features are developed as an event.

3.2 Calling Stages

Event calling stages in HyperDbg allow us to specify **WHEN** an event should be triggered during the emulation process.

There are three calling stages: *pre*, *post*, and *all*. The *pre* stage triggers the event before the emulation, while the *post* stage triggers it after the emulation. The *all* stage triggers the event both before and after the emulation. For example, by using the *post* calling stage in the '!monitor' command, we can observe and potentially modify the memory contents after a MOV instruction has been executed. Another example is using the *pre* calling stage in the '!exception' command to view the value of the CR2 register after a page-fault. By utilizing these calling stages, we can gain valuable insights into the behavior of specific events and customize our debugging experience.

3.3 Short-circuiting

Event short-circuiting is a powerful mechanism introduced in the newest versions of HyperDbg. It allows users to ignore the execution of certain events based on user-defined conditions. For example, if a *SYSCALL* event is triggered in the debuggee, event short-circuiting can be used to ignore the execution of this event as if no *SYSCALL* instruction was executed. This mechanism can also be applied to other events like *IN*/*OUT* instructions, *RDMSR* or *WRMSR* instructions, and memory writes. By short-circuiting an event, the *post* calling stage is no longer called. Event short-circuiting can be manually changed using the 'events' command or by using scripts with the '*event_sc*' function.

Event short-circuiting can be highly advantageous in various scenarios. For example, it can be used to prevent memory modifications by short-circuiting write events to specific memory addresses. It can also be used to modify the behavior of system calls by short-circuiting the execution of certain system calls based on specific conditions. Event short-circuiting provides flexibility and control over the program execution, allowing users to selectively bypass events based on their requirements. It is a powerful mechanism that enhances the debugging capabilities of HyperDbg.

4 Tracking Function Calls

At the recent release of HyperDbg, a novel capability known as 'tracking function calls' was introduced. This feature was primarily designed by using the Monitor Trap Flag (MTF) bit, plus changing the interruptibility of the system bit within hypervisors. Whenever the user tries to step through instructions, HyperDbg guarantees that no other instructions, regardless of the core, are executed simultaneously.

This functionality enables seamless transitions between the user-mode and the kernel-mode, enhancing the ability to automatically switch between the two. When combined with a symbol server, it introduces a distinct advantage by enabling the tracking of function calls across various functions while leveraging the state of registers and other related information. Figure 1 demonstrates how it can be used to create a trace of functions. Note that, you can trace a system-call starting from user-mode and ending in the kernel-mode.

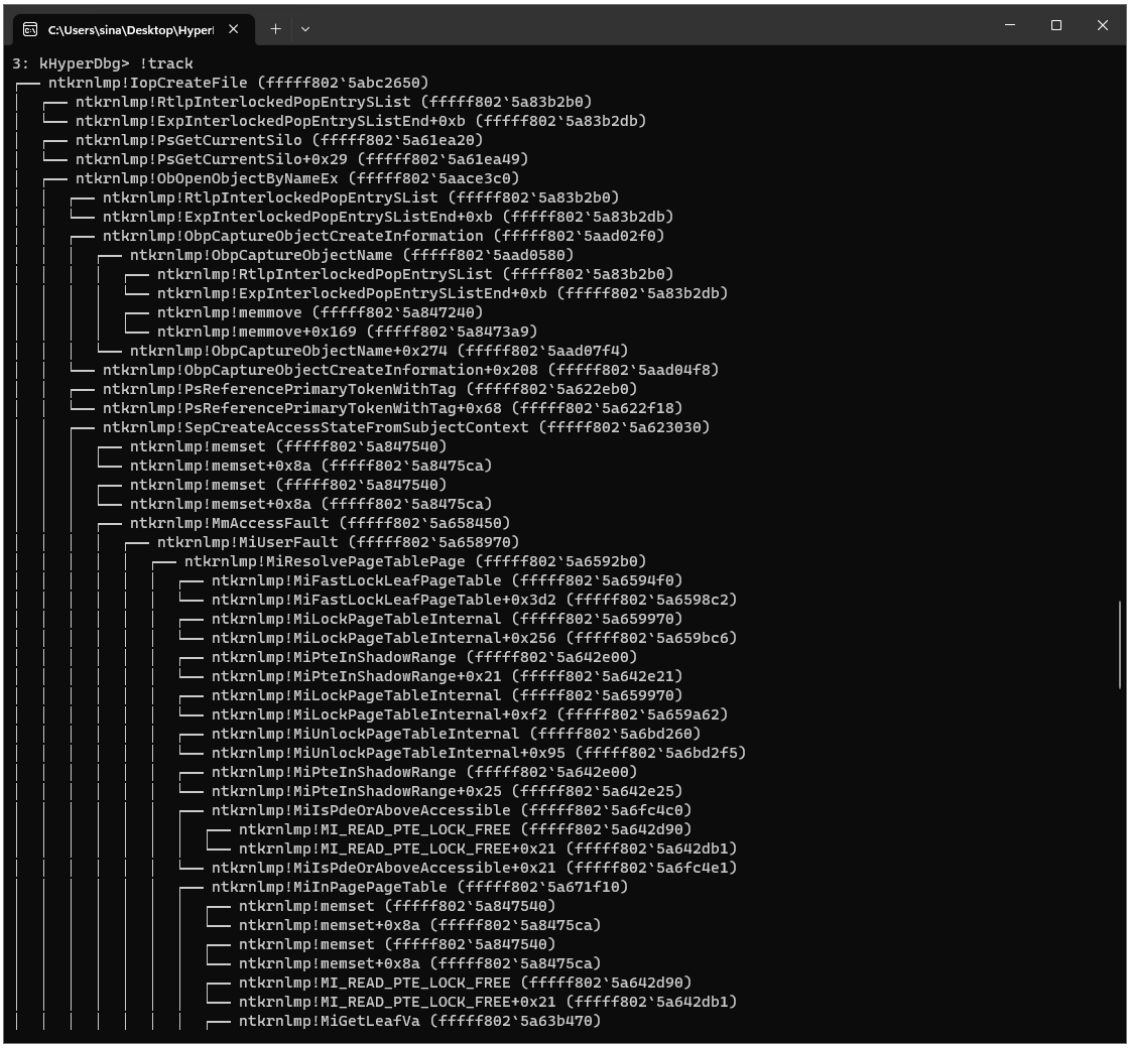


Figure 1: Tracking Functions

5 Running a Process without a Debug Flag

In HyperDbg, running a process is done without using the debug flag (DEBUG_PROCESS) in the *CreateProcess* function. This brings us with tons of benefits, while the main benefit is transparency. Whenever a process is running with the debug flag, Windows adds plenty of traces and indicators (e.g., the debug heap structures, the *BeingDebugged* PEB flags,[1] etc.) for the target process, so anti-debugging methods might search for these patterns and detect the presence of the debugger, thus, hiding their possible malicious behaviors. Although HyperDbg won't guarantee 100% transparency, avoiding running the process with the debug flag prevents most of these circumstances and offers more robust transparency.

5.1 How does it work?

To ensure maximum transparency, HyperDbg adopts a hypervisor-based approach for capturing the entrypoint of a Portable Executable (PE). This approach circumvents the use of debugging flags, which can leave distinguishable artifacts, such as debug-optimized heap structures, potentially revealing the presence of a debugging environment.

In this approach, the target PE is configured to load with a suspended flag (e.g., a *CREATE_SUSPENDED* flag), forcing Windows to set initial attributes, allocate physical memory, assign a *CR3* register, and establish basic process structures such as the Process Environment Block (PEB), before resuming with other processes. Once Windows switches context to the target process, HyperDbg monitors the PEB to trace any modules loaded by the PE loader, identifying the memory address at which the entrypoint of the program will be loaded. Upon loading the target module and determining its entrypoint, HyperDbg checks for the module's availability in memory.

If the process is already running in the system, the physical address of the entrypoint is readily available. HyperDbg then revokes the execution permission of the page corresponding to the entrypoint of the main module, triggering an Extended Page Table (EPT) violation upon any attempt to execute it. Since Windows conventionally attempts to execute the code page before its contents are fully loaded into memory, a page fault is triggered. However, as HyperDbg intercepts the execution flow using EPT before the page is fetched, it injects a page fault to fetch the page contents. Following this, HyperDbg gains access to the instructions of the main module for logging. Subsequently, HyperDbg restores the executable permission it had previously revoked from the code page, allowing the execution to proceed as normal.

6 Detecting Mode Changes

Detecting changes between user-mode to kernel and kernel-mode to user-mode is done by using the Mode-Based Execution Control (MBEC) available at Kaby Lake and later generations of Intel Processors. This feature introduces several intriguing use cases, plus tons of new reverse engineering opportunities. We'll see what are the use cases, but for now, let's see how this mode change detection is implemented.

HyperDbg is able to be configured with custom rules based on the execution mode of processes, enabling the generation of logs for subsequent analysis. It provides two methods to differentiate between user and kernel execution modes. The first method is based on Mode-Based Execution Control (MBEC) and the second method is based on the regular OS page-table modification.

The first approach is based on Mode-Based Execution Controls (MBEC). This method involves allocating an additional EPT page table (*MBEC-Denied EPT*) where user-mode execution is disallowed in the corresponding EPT entries. When Windows switches context to the target process, the EPT table is changed to *MBEC-Denied EPT*, prohibiting user-mode execution. Upon an attempt to execute user-mode instructions, an EPT Violation (VM-exit) occurs, notifying the VMM. HyperDbg can thus detect user-mode execution.

The second method is based on regular OS page-table modifications. This approach, primarily for older processors (Skylake and earlier), involves setting the U/S bit of the first regular OS page table (PML4) when HyperDbg detects a process context-switch to the target process, preventing user-mode execution. Page faults are intercepted through the Exception Bitmap, allowing the virtualized core to access user-mode code. When user-mode code is requested, a page-fault (#PF) occurs, which is intercepted by the VMM. This way, HyperDbg detects the user-to-kernel switching and adjusts the OS page tables accordingly.

During initialization, HyperDbg automatically selects the appropriate approach based on processor support for Mode-Based Execution Controls. If supported, it uses the first approach; otherwise, it switches to the latter. While the second approach ensures backward compatibility with older processors, it has additional performance overhead due to the management of multiple indirect page faults, which may impact the transparency of the debugger.

The second approach is less efficient than the first, as HyperDbg must manage numerous page-faults unrelated to execution mode transitions.

6.1 The Time Freeze Debugging

Time freeze debugging is a relatively new technique introduced in HyperDbg that makes it possible to block the execution of user-mode instructions for the target process. As previously mentioned about short-circuiting events, this mechanism can be combined with the user/kernel transition detection and make the user able to freeze the instruction fetch of certain execution modes based on user configuration.

6.2 Kernel-mode to User-mode

The transition from kernel-mode to user-mode is a new event HyperDbg, particularly in the context of effective memory introspection. This process is managed by the use of specific EPT pointers (EPTPs), which are used for monitoring and controlling the execution flow within the system. The EPT Pointer designed to disable user-mode execution plays the main role here. This EPTP is configured to contain normal entries while explicitly disabling all user-mode execution bits. Consequently, any attempt to execute user-mode code (CPL=3) results in a VM-exit due to an EPT Violation.

This mechanism ensures that any transition from kernel-mode to user-mode is immediately caught and handled, and therefore HyperDbg will be notified. This capability mainly uses the Mode-Based Execution Control (MBEC).

6.3 User-mode to Kernel-mode

Exactly like the previous scenario, the transition from user-mode to kernel-mode is implemented by using MBEC. To manage this transition within the system, a dedicated EPT Pointer (EPTP) is utilized, specifically designed to disable kernel-mode execution. This EPTP contains normal entries, similar to the user-mode interception, but with a distinction: all kernel-mode (supervisor) execution bits are disabled. Therefore, any execution attempt of kernel-mode code (CPL=0) triggers a VM-exit due to an EPT Violation.

This control mechanism is used to disable the execution of the kernel-mode codes and thus trigger an event for each, kernel code execution. By enforcing these boundaries and controlling the transitions between different execution modes, HyperDbg provides two new debugging events.

7 Intercepting Execution of a Memory Range

In addition to other capabilities related to advanced memory introspection, intercepting execution attempts within specific address ranges is another attempt to bring the capabilities of EPT to security researchers. This technique is particularly used for reverse engineers aiming to understand, the top-level functionality of a module both in user-mode and kernel-mode.

The implementation of this mechanism is through the manipulation of EPT page tables, showcasing the flexibility and power of hardware-backed virtualization technologies in modern computing environments.

The core of this interception mechanism uses EPT and disables the execution bit for EPT entries corresponding to the specified address range. By modifying the EPT page tables in this manner, any attempt to execute code within the targeted address range, regardless of whether it originates from user-mode (CPL=3) or kernel-mode (CPL=0) immediately causes an EPT Violation and notifies the debugger.

8 Real World Usage of New Reversing Techniques

By employing the newly implemented infrastructures that leverage hypervisor capabilities, tens of novel reverse engineering techniques can be added to our reverse engineering toolset. Many of these techniques are innovative and have the potential to significantly reshape our reverse engineering practices. Some of these new techniques are discussed below.

8.1 An Easier Approach to Analyze System-calls

Analyzing system calls has long been a source of interest for security researchers and reverse engineers. First, because system-calls are gates from low-privilege user-mode applications to the high-privilege operating system kernel. It is always challenging to trace system-calls, especially in the Windows operating system as they constantly try to prevent system-call hooks using different techniques by employing PatchGuard. On the other hand, the rate of the execution of system-calls are high and it is not easily possible to intercept a specific system-call.

To address this issue, HyperDbg comes with a new solution, to track system-calls from user-mode to kernel-mode. This will give the reverse engineers the opportunity to see what functions are called in the kernel during the execution of a system-call and when this system-call possibly terminates. This way, it is also possible to measure the actual function-level coverage of a system-call.

For this purpose, we need to trigger a breakpoint before the execution of a specific system-call, for example, the following code is a demonstration of such a breakpoint that triggers before the execution of *NtCreateFile*.

```
// Target breakpoint
DebugBreak();

Status = NtCreateFile(&hFile,
    GENERIC_ALL | SYNCHRONIZE,
    &oa,
    &ioStatusBlock,
    NULL,
    FILE_ATTRIBUTE_NORMAL,
    0,
    FILE_OVERWRITE_IF,
    FILE_SYNCHRONOUS_IO_NONALERT,
    NULL,
    0);
```

Once the above code is executed, the debugger will be notified and after that, the '!track' command can be used to track the routines related to the system-call from user-mode to the kernel-mode.

8.2 Ignoring Memory Writes to Variables in the Memory

Another new approach to debugging which is designed as a result of the event short-circuiting is the ability to ignore memory reads and writes.

Imagine, we have a variable that is changed constantly by the program. For example, a device that constantly updates a status variable and we need to change the actual status of the device or ignore some the changes.

Using the following script armed with the *event_sc* function (discussed earlier), we could easily ignore or change the memory writes.

```
!monitor w 7ff71f118210 l 4 pid 3fa8 script {
  printf("Writing into memory address: %llx is ignored\n", $context);
  event_sc(1);
}
```

This approach to debugging is possible when the event short-circuiting is combined with the event calling stages.

8.3 Bypassing the Execution of Certain Instructions

Almost all of the privileged instructions have some configurations in the VMCS which leads to a VM-exit once they're accessed or modified. Table 1 shows different reasons when a VM-exit could occur.

Using the short-circuiting mechanism, each of these instructions can be manipulated or ignored. For example, *IN* or *OUT* instructions are responsible for sending data inside and outside using Port Mapped I/O (PMIO). These instructions can be intercepted, modified, or even ignored. The same principles also apply to Memory Mapped I/O (MMIO) devices.

Other than that, instructions that are responsible for reading or modifying CPU configurations like *RDMSR* and *WRMSR* can also be modified.

Another example is ignoring or changing system-calls. The security researcher could easily modify the execution parameters of certain system-calls and possibly ignore their execution and modify the environment in a way as if the system-call was actually executed and returned a special status while in reality, it is suppressed by the debugger.

Table 1: Instructions Causing VM Exits

VM-Exits	Instructions
Always Cause Exit	
These instructions always cause an exit	INVEPT, INVVPID, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, VMXON
Could Cause Exit	
These instructions could cause an exit	CLTS, ENCLS, ENCLV, HLT, IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD, INVLPG, INVPCID, LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR, LMSW, MONITOR, MOV from CR3, MOV from CR8, MOV to CR0, MOV to CR3, MOV to CR4, MOV to CR8, MOV DR, MWAIT, PAUSE, RDMSR, RDPMC, RDRAND, RDSEED, RDTSC, RDTSCP, RSM, TPAUSE, UMWAIT, VMREAD, VMWRITE, WBINVD, WRMSR, XRSTORS, XSAVES
Other Sources of Exits	
Exceptions, Triple fault, External interrupts, Non-maskable interrupts (NMIs), INIT signals, Start-up IPIs (SIPIs), Task switches, VMX-preemption timer, interrupt-window exiting, NMI-window exiting	

8.4 Functions that are Called During Execution of a Module

One of the challenges facing security researchers when they want to start analyzing a new module is determining where to begin. Typically, modules contain numerous functions and assembly codes, many of which are uninteresting to reverse engineers. The objective may be to interact with the application, locate the functions handling specific functionalities, and start reverse engineering from there. Now the question is, how to find these crucial functions?

Usually, the reverse engineers try to put different breakpoints on entire functions in a module. While this approach might work in some scenarios, it might not be the case for all of the modules as the code execution might not begin on the very first instruction of a function, or even an obfuscated code or a packed/protected malware might easily understand this behavior and doesn't show it malicious intends in that case.

To address this problem, HyperDbg recently introduced the capability to intercept the execution of a user-specific range of the memory. Using this solution, the reverse engineers could easily monitor an entire module address space for possible executions while interacting with the program to force the module to execute instructions. Once, the instructions are executed, the debugger will be notified and pause the entire system.

This approach is also more transparent than putting breakpoints to the target application as it won't change any bits in the target memory range.

8.5 Freezing the Execution of Kernel Modules as well as Applications

Using the time-freezing infrastructures (discussed earlier), the reverse engineers are able to halt the execution of a special thread/process. This is also possible by using certain system-calls like *SuspendThread*[6] and *ResumeThread*,[5] but using the memory freezing approach, the security researcher is able to freeze the application once a special condition happens in the program (e.g., a special region of memory got executed).

A real-world example of this scenario occurs with threads monitoring each other's execution to detect whether one

is paused or being debugged. If debugging is detected, the threads terminate the entire program, as an implementation of an anti-debugging method. However, the solution proposed here involves freezing the thread by preventing instruction fetches in that particular thread. Essentially, the thread operates normally within the system, but when it attempts to execute instructions, HyperDbg prevents it. As the operating system is not involved in this process, if the other threads try to query the operating system about the state of their peer thread, the OS would return a normal status to them as the OS itself does not have any idea about the presence of the debugger.

9 Conclusion

In conclusion, the use of hypervisor technology has not only revolutionized the hardware virtualization industry but also has had impacts on the landscape of reverse engineering, offering new capabilities to reverse engineers. Using features such as execution interception and mode change detection, reverse engineers can now go deeper into the inner workings of programs with greater insights and efficiency. As hypervisor-based tools continue to evolve, they are becoming an indispensable asset in reverse engineering toolkits.

References

[1] Anti-debug: Debug flags, 2023. `https://anti-debug.checkpoint.com/techniques/debug-flags.html`.

[2] Stella Bitchebe, Djob Mvondo, Alain Tchana, Laurent Réveillère, and Noël De Palma. Intel page modification logging, a hardware virtualization feature: study and improvement for virtual machine working set estimation. *arXiv preprint arXiv:2001.09991*, 2020.

[3] Intel Corporation, 29.3.6. Page-Modification Logging. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3C*, 2023.

[4] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadizanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. Hyperdbg: Reinventing hardware-assisted debugging. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1709–1723, 2022.

[5] Microsoft. Resumethread function (processthreadsapi.h). Microsoft Docs, 2024.

[6] Microsoft. Suspendthread function (processthreadsapi.h). Microsoft Docs, 2024.

[7] Shiru Ren, Yunqi Zhang, Lichen Pan, and Zhen Xiao. Phantasy: Low-latency virtualization-based fault tolerance via asynchronous prefetching. *IEEE Transactions on Computers*, 68(2):225–238, 2018.