applied sciences

MDPI

Article

H-KPP: Hypervisor-Assisted Kernel Patch Protection

Michael Kiperberg [1,†] and Nezer Jacob Zaidenberg [2,3,*,†]

[1] Department of Software Engineering, Shamoon College of Engineering, Beer-Sheva 8410802, Israel; michaki1@sce.ac.il
[2] Holon Institute of Technology, Faculty of Sciences, Holon 5810201, Isreal
[3] Faculty of Information Technology, University of Jyväskylä, FI-40014 Jyväskylä, Finland
* Correspondence: scipio@scipio.org
† These authors contributed equally to this work.

Abstract: We present H-KPP, hypervisor-based protection for kernel code and data structures. H-KPP prevents the execution of unauthorized code in kernel mode. In addition, H-KPP protects certain object fields from malicious modifications. H-KPP can protect modern kernels equipped with BPF facilities and loadable kernel modules. H-KPP does not require modifying or recompiling the kernel. Unlike many other systems, H-KPP is based on a thin hypervisor and includes a novel SLAT switching mechanism, which allows H-KPP to achieve very low (≈ 6%) performance overhead compared to baseline Linux.

Keywords: virtualization; Kernel Integrity; DKOM

Citation: Kiperberg, M.; Zaidenberg, N.J. H-KPP: Hypervisor-Assisted Kernel Patch Protection. *Appl. Sci.* **2022**, *12*, 5076. https://doi.org/10.3390/app12105076

Academic Editor: Arcangelo Castiglione

Received: 19 January 2022
Accepted: 8 April 2022
Published: 18 May 2022

1. Introduction

The execution of an unauthorized code is one of the attackers' main vehicles in their malicious actions. Multiple solutions have been proposed to prevent these attacks at various stages, for example, Data Execution Prevention (DEP) [1] and driver signing [2] attempt to prevent the execution and introduction of unauthorized code. Unfortunately, both solutions were circumvented [3,4].

In addition to executing new code in kernel mode, attackers may attempt to perform direct kernel-object modification (DKOM) to achieve their goal by overwriting various fields of kernel objects directly. DKOM attacks can be used for privilege escalation and hiding the attack traces. Microsoft's Kernel Patch Protection (KPP) is an obfuscated integrity checking mechanism for the Windows kernel. As with the previously described security measures, KPP was reverse-engineered, and disabled [5]. In addition, KPP is not available for the Linux kernel, which also suffers from vulnerabilities [6]. In particular, 158 new vulnerabilities were discovered in 2021 alone [7].

We introduce H-KPP, hypervisor-based protection for Linux kernel code and data structures. H-KPP prevents the execution of unauthorized code in kernel mode. In addition, H-KPP protects certain object fields from malicious modifications. H-KPP can protect modern kernels equipped with BPF facilities and loadable kernel modules. H-KPP does not require modifying or recompiling the kernel. Unlike many other systems, H-KPP is based on a thin hypervisor and includes a novel SLAT switching mechanism, which allows H-KPP to achieve very low (≈6%) performance overhead compared to baseline Linux.

H-KPP accomplishes its main goal using two secondary-level address translation (SLAT) hierarchies: the kernel and user mode hierarchies. The hypervisor switches between the two hierarchies on every mode transition. The kernel-mode hierarchy prevents the execution of all the pages but the pages belonging to the verified kernel and modules. The transition between the hierarchies does not require the hypervisor's attention, thus keeping the overall system performance at optimum. H-KPP, unlike similar systems, supports modern kernel features, like just-in-time code generation, which is used by the BPF.

H-KPP's detection of DKOM is based on periodic verifications of various object fields. This behavior is similar to Microsoft's KPP. However, the functionality of H-KPP in this regard is much broader and is comparable with the functionality of Volatility, an offline forensic investigation tool [8].

We have verified the effectiveness of H-KPP by using a specially crafted kernel module that mimics some aspects of the malicious behavior of various attacks. It may seem more natural to run an actual attack; unfortunately, the attacks we have found target older versions of the Linux kernel. H-KPP successfully detected all the attacks that were performed by the kernel module.

We have measured the performance of H-KPP code protection using various benchmarks, which showed an average overhead of only ≈6%. We note that the overhead of full hypervisors, which form the basis for other solutions, is much more significant. In addition, we have evaluated the performance of the periodic kernel object verification. Our main contributions are:

- We present the design of H-KPP, a KPP-like system implemented as a thin hypervisor.
- We present a novel SLAT-switching technique used by H-KPP, which can be helpful in other scenarios.
- We present an online approach for kernel object verification that does not require recompilation of the kernel.
- We evaluate the effectiveness of H-KPP using a kernel module that mimics common attacks.
- We evaluate the performance of H-KPP.

2. Background

2.1. Virtualization

Many modern processors are equipped with extensions to their basic instruction set architecture that enables them to execute multiple operating systems simultaneously. H-KPP utilizes Intel's implementation of these extensions, which they call Virtual Machine Extensions (VMX). The software that governs the execution of the operating systems is called a *hypervisor* and each operating system (with the processes it executes) is called a *guest*. Transitions from the hypervisor to the guest are called VM-entries and transitions from the guest to the hypervisor are called VM-exits. While VM entries occur voluntarily by the hypervisor, VM exits are caused by events during the guest's execution, e.g., page faults, execution of a privileged instruction, or a timer's timeout. The event that causes a VM exit is recorded for future use by the hypervisor. A special data structure called Virtual Machine Control Structure (VMCS) allows the hypervisor to specify the events that should trigger a VM exit and many other settings of the guest.

Intel's Extended Page Table (EPT), a technology generally called Secondary Level Address Translation (SLAT), allows the hypervisor to configure a mapping between the physical address space, as it is perceived by a guest to the real physical address space. Similarly to the virtual page table, EPT allows the hypervisor to specify the access rights for each guest physical page. When a guest attempts to access a page that is either not mapped or has inappropriate access rights, an event called an EPT violation occurs, triggering a VM exit.

While, in general, hypervisors enable the execution of multiple operating systems on a single machine, we propose to use a hypervisor environment for securing a single operating system. The idea to use such a limited hypervisor for providing additional security is not new [9,10]. Unlike full hypervisors, which tend to degrade the overall system performance significantly, thin hypervisors may intercept only a small portion of the system's events, thus keeping the system performance at an optimum (see Section 5.2).

2.2. Execution Prevention

The execution of an unauthorized code is one of the attackers' main vehicles in their malicious actions. Multiple solutions have been proposed to prevent these attacks at various stages. For example, Microsoft's Data Execution Prevention (DEP) technology prevents

the execution of instructions from unauthorized pages by utilizing the page table's `NX` bit introduced by Intel. Linux also utilizes the `NX` bit on processors that support it starting from version 2.6.8 [11]. Unfortunately, this protection can be bypassed [3].

Another example is the driver signing facilities available on Windows [2], and Linux [12]. These facilities either prevent or issue a warning upon an attempt to load an unsigned driver (kernel module). These facilities aim to prevent malicious code from even getting into the kernel. Unfortunately, Kleissner demonstrated [4] that driver signing can be bypassed.

Microsoft's Kernel Patch Protection (KPP) is an obfuscated integrity checking mechanism for the Windows kernel. For example, this mechanism prevents patching of the System Service Descriptor Table (SSDT), which holds the addresses of the system-call handlers. KPP consists of multiple verification code pieces that form a complete algorithm for verifying the kernel's critical data structures. In addition, the code pieces verify the integrity of themselves in order to prevent their circumvention. As with the previously described security measures, KPP was reverse-engineered and bypassed [5]. H-KPP is most closely related to KPP in its goals. However, unlike KPP, H-KPP protects its verification procedure by moving it from the kernel mode to the more secure root mode.

2.3. Kernel Data Structures

Attacks based on direct kernel-object modification (DKOM) [13] allows the attacker to achieve his goal by overwriting various fields of kernel objects directly. This scheme allows the attacker to place the system in a custom state that may be unreachable using regular kernel functions. For example, typically, it is impossible to reach a state in which an active process is not linked to the process list. DKOM attacks can be used to modify dynamically allocated objects and static data structures, like the interrupt vector and the system call tables. Because the kernel itself can modify kernel objects during an execution of a vulnerable system-call handler [14], kernel code integrity alone does not guarantee the integrity of the kernel objects.

DKOM can also be used to grant higher privileges to the process controlled by the attacker. In addition, the attacker typically attempts to hide his malicious activity. For example, the hiding procedure may involve removing the malicious process from the process list. These modifications cannot be detected by the code integrity mechanism and therefore require another approach.

Lastly, the kernel's polymorphic nature, which is implemented using structures of function pointers, is the attackers' main target. By altering these pointers, the attacker can alter the behavior of various system aspects. For example, by altering the `show` field of the `tcp4_seq_ops` structure, the attacker can hide his network activity. Similarly, the attacker can hide a certain process by altering the `iterate` field of the `i_fop` field in the `inode` object representing the `/proc` directory. While the attacker's inability to introduce new code to the kernel imposes severe limitations, it is still possible [15] to reuse existing code to achieve some of the attacker's goals. Therefore, it is important to protect the function pointers from malicious modifications.

3. System Design

H-KPP is a thin hypervisor that supports the execution of a single operating system. The hypervisor is embedded in an EFI [16] application that boots before the operating system and initializes the hypervisor. After successful initialization, the EFI application boots the operating system bootloader, which initializes the operating system itself. The operating system executes under the inspection of H-KPP. The main goal of the hypervisor's inspection is to prevent the execution of malicious code in kernel mode.

H-KPP accomplishes its main goal using two secondary-level address translation (SLAT) hierarchies, see Table 1. The first hierarchy, kSLAT, is active when the guest resides in kernel mode, while the second hierarchy, uSLAT, is active when the guest resides in user mode. The hypervisor switches between the two hierarchies on every mode transition. Both hierarchies define identity mapping. This mapping is used solely to

prevent undesirable access to some pages. The kSLAT and the uSLAT hierarchies prevent access to the hypervisor's memory and prevent the modification of the kernel code pages. In addition, the kSLAT hierarchy prevents the execution of all the other pages.

Table 1. Configuration of SLAT hierarchies.

Page Type	kSLAT	uSLAT
H-KPP memory	No access	No access
Firmware	Read & Execute	No access
Other	Read & Write	Full access

For its correct execution, H-KPP requires information about the underlying operating system. A special tool extracts this information during the installation of H-KPP, before its first execution. The information is then delivered to H-KPP using a configuration file.

3.1. Information Collection

H-KPP requires two types of information about the underlying operating system: (a) kernel symbols, (b) module hashes. The offsets of the kernel's global variables, function, and structure fields allow H-KPP to inspect the operating system's internal state during the execution. Before granting execution permissions to the kernel's code page, H-KPP verifies its integrity by comparing its hash against a pre-calculated set of hashes. A special tool hashes the executable pages of all the kernel modules by traversing either local directories or the official repository of the operating system distributor. The hashes and the offsets are stored in a configuration file, which H-KPP loads during its initialization.

3.2. First-Phase Initialization

H-KPP's initialization consists of two phases. The first phase takes place before the operating system's bootloader. The second phase occurs during the initialization of the operating system, after the establishment of its memory layout, which is affected by ASLR [17].

During its initialization, the EFI application that embeds H-KPP allocates memory for the hypervisor by calling EFI's AllocatePages function with EfiRuntimeServicesData as its memory type argument. Memory regions allocated with this memory type appear as reserved in the memory map report that the EFI firmware delivers to the operating system. Therefore, benign operating systems and drivers do not access such memory regions.

After allocating memory for H-KPP, the EFI application initializes the hypervisor itself. The initialization includes the configuration of internal data structures and the installation of the SLAT hierarchies. H-KPP's initialization proceeds by executing the remaining EFI application as the single guest of the hypervisor. The application exits to firmware, which loads the operating system bootloader. The bootloader and the operating system itself execute as a hypervisor guest, thus allowing the hypervisor to inspect their execution. The hypervisor's memory is protected from the operating system by the SLAT.

As the last step of its first-phase initialization, H-KPP configures the interception of assignments to the LSTAR MSR, which holds the address of the system call handling function. After laying out its code and data structures in the memory, the operating system sets this MSR. Therefore, this MSR can be used as a trigger for the second phase of H-KPP's initialization.

3.3. Second-Phase Initialization

H-KPP reacts to the operating system's attempt to set the LSTAR MSR, by completing H-KPP's initialization. The initialization process assumes that at this point, the kernel determines the memory layout of its code and data structures. At first, the hypervisor determines the kernel base address using the following equation:

$$Base = Sym_{stext} + IDT[0] - Sym_{divide\_error}$$

In this equation, *IDT* is the Interrupt Descriptor Table, a table that holds the address of all the interrupt handlers. The processor delivers interrupt zero upon an attempt to divide by zero. In Linux, this interrupt is handled by the divide_error function. Sym_{name} denotes the address of the symbol named "name" as reported by the debugging information of the kernel binary.

After determining the kernel's base address, the hypervisor configures the kSLAT and the uSLAT as presented in Table 1 and sets the kSLAT to be the active SLAT hierarchy. Pages containing code are marked as readable, executable, but not writable (RX), while other pages are marked non-executable but readable and writable (RW). After determining the kernel's base address, H-KPP sets RX permissions to its code pages. In addition, H-KPP grants RX permissions to all the firmware code, which is invoked by the kernel through the EFI API.

H-KPP constantly monitors the loading of additional kernel modules and grants RX permissions to their code pages. This is achieved by intercepting the "load_module" function.

3.4. Functional Interpretation

During its execution, H-KPP collects information about the executing environment. This is achieved by intercepting the invocation of specific functions and analyzing their arguments and return values. The interception mechanism (IM) is implemented using software breakpoints. When IM is requested to intercept a function invocation, it replaces the first byte of this function with a breakpoint instruction. It saves the replaced instruction in the hypervisor's internal storage. In addition, IM configures the CPU to inform it about software breakpoint interrupts.

IM reacts to a software breakpoint by emulating the function's original first instruction. Then, IM pushes onto the stack a special data structure in order to induce an additional software breakpoint interrupt upon the termination of the intercepted function. As Figure 1 shows, the data structure consists of three fields. The first field is set to the function's first instruction, the breakpoint instruction. After completing its execution, the function will arrive at the breakpoint instruction, inducing a software breakpoint interrupt and transferring the control to IM. The second field, which is set to the magic number $0 \times 1{,}122{,}334{,}455{,}667{,}788$, allows IM to determine whether the software interrupt was generated due to an entry to the intercepted function or an exit from it. In the first case, IM pushes the special data structure as described. IM pops it and transfers the control to the original return address in the second case. The third field stores the values of the first six arguments passed to the intercepted function, which can be examined on an exit from the function.

```
Struct StackFramePrologue {
    u64 retAddr;
    u64 magic;
    u64 originalArgs[6];
}
```

Figure 1. Special stack data structure.

We note that IM is not required to include a full emulator for emulation of the first instruction of an intercepted function. In practice, our observations show that the first instruction belongs to the following list: (a) PUSH RBP, (b) CALL IMM32, (c) NOP. Therefore, in our current implementation, IM emulates only these three instructions.

IM notifies the interception requestor upon a function interception by invoking its callback function F. The value returned by F instructs IM to either act normally or deviate from the normal operation, see Table 2. During the interception of an entry to a function, the normal operation transfers control to the function by emulating its first instruction. A deviation is staying at the breakpoint instruction. During the interception of an exit to a function, the normal operation is returning to the calling function. A deviation is calling the same function again. The usefulness of this non-intuitive behavior is explained below.

Table 2. IM actions on interceptions.

Event	Normal	Deviation
Entry	Continue to function	Stay at breakpoint
Exit	Continue to caller	Re-enter function

3.5. SLAT Switching

H-KPP implements efficient switching between kSLAT and uSLAT upon transitions between kernel-mode and user-mode by utilizing Intel's VMFUNC instruction. Intel's virtualization technology (VMX) allows the hypervisor to configure a set of SLAT hierarchies that can be switched from the guest using the VMFUNC instruction without requiring transitions to the hypervisor, thus keeping the overhead at a minimum.

H-KPP injects the VMFUNC instruction using trampolines from four locations in the kernel (see Table 3): (a) transition to kernel mode due to system call, (b) transition to user mode due to a return from a system call, (c) transition to kernel-mode due to an interrupt, (d) transition to user-mode due to a return from an interrupt. H-KPP inserts a jump instruction to a pre-allocated memory region at each location. In addition, H-KPP generates code that performs VMFUNC, emulates the instruction that was replaced, and jumps back to the original location.

Table 3. Mode Transition Points.

Event	System Call	Interrupt
Entry	entry_SYSCALL_64	error_entry
Exit	syscall_return_via_sysret	retint_user

H-KPP allocates the required memory region in the kernel context by intercepting the "module_alloc" function, which is used by the kernel for general memory allocation. Upon the first entry to the "module_alloc" function, H-KPP does nothing and requests IM to proceed to the function. Upon the first exit from the "module_alloc" function, H-KPP stores the function's return value (the address of the allocated memory region) in an interval variable X and requests the IM to execute the "module_alloc" function again. Upon another entry to the "module_alloc" function, H-KPP changes the value of the "size" argument to match the total size of the SLAT switching code. Upon an exit, H-KPP performs several actions: (a) it modifies the kernel's page table to grant the newly allocated region with execution rights, (b) it grants this newly allocated page with execution rights in kSLAT, (c) it restores the return value to the value of the internal variable X, (d) it requests IM to disable the interception of the "module_alloc" function.

3.6. Module Loading

During its initialization, the kernel loads additional executable pages in the form of kernel modules. The kernel's "load_module" function is responsible for loading new modules. Its first argument contains information about the module to be loaded. In particular, it contains information about the module's name and a temporary buffer containing the module's code and data (see Figure 2). H-KPP intercepts entries to this function and exits from it. On an entry, H-KPP hashes the temporary buffer, page-by-page. If a particular page is paged out, the hypervisor injects a page-fault exception to the guest and requests the IM to stay at the breakpoint instruction. As a result, the operating system handles the page-fault exception by loading the missing page, and the hypervisor gets another opportunity to hash the temporary buffer. After completing the hash computation, the hypervisor verifies that the hash value exists in the pre-calculated database.

On an exit from the "load_module" function, H-KPP looks up the recently loaded module in the linked list of kernel modules pointed by the "modules" global variable. After finding the module in this linked list, H-KPP configures kSLAT to grant execution rights

to the module's code pages. We note that the lookup is performed by the module's name, which is available through the first argument of the "load_module" function, which IM preserved in the special stack data structure.

```
struct load_info {
const char *name;
struct module *mod;
Elf_Ehdr *hdr;
unsigned long len;
...
};
```

Figure 2. Layout of the kernel's load_info structure.

3.7. Berkeley Packet Filter

The Berkeley Packet Filter (BPF) [18] technology allows user-space processes to supply programs for filtering network packets. These programs, compiled to bytecode, are interpreted by the kernel. Alternatively, the bytecodes can be compiled to machine code by a just-in-time compiler embedded in the kernel, thus introducing new executable code.

The simplest solution to the problem of a new unverifiable code is disabling the BPF JIT compiler through the "bpf_jit_enable" variable. However, this change can harm the overall system performance in some cases. Therefore, in its current implementation, H-KPP intercepts the "do_jit" function and configures kSLAT to grant execution rights to the newly generated code pages.

4. Kernel Objects Protection

H-KPP detects the modification of three types of kernel object fields. The first type includes static objects, like the interrupt vector and the system-call table. Because the content of these static objects does not change during the execution of the operating system, H-KPP computes the hash of these objects after their initialization and then verifies their integrity by periodically comparing their hash results with the precomputed value. H-KPP uses SHA-1 as its hashing algorithm.

The second type includes function pointers of various objects, like the `i_fop` field of the `inode` object (see Figure 3). These pointers may be assigned different values depending on the loaded modules. For example, each file system can define a new set of functions pointed by the `i_fop` field. Therefore, H-KPP limits its verification procedure to checking that the `i_fop` field points to a module's variable and that each field of this variable points to the beginning of some function. The verification itself is performed periodically by traversing the object graph as depicted in Figure 3. More precisely, H-KPP traverses the inodes and constructs a set of all the objects pointed by the `i_fop` field. Then, H-KPP verifies that the fields of these objects point to functions. If so, H-KPP stores a pair consisting of an object address and its hash in H-KPP's internal data structure. In the future, instead of verifying each field of an object separately, H-KPP hashes it and compares its hash to the previously stored value.

The third type includes fields whose values are subject to certain invariants. For example, no two `task_struct` objects should reference the same credentials object via their `cred` fields. H-KPP performs this verification by traversing the list of processes and storing the addresses stored in the `cred` fields in an array. Then, H-KPP sorts the array and verifies that consecutive items are different.

Another example is the equality invariant over the sets of `task_struct` objects. As depicted in Figure 4, the set of active processes can be constructed using information from two sources: (1) the linked list formed by the `tasks` field of the `task_struct`, (2) the tree formed by the `children` and the `sibling` fields of the `task_struct`. H-KPP periodically tests the sets of task_struct objects obtained from these sources. For each source, H-KPP stores the addresses of the structures in an array. Then, the arrays are sorted and compared.

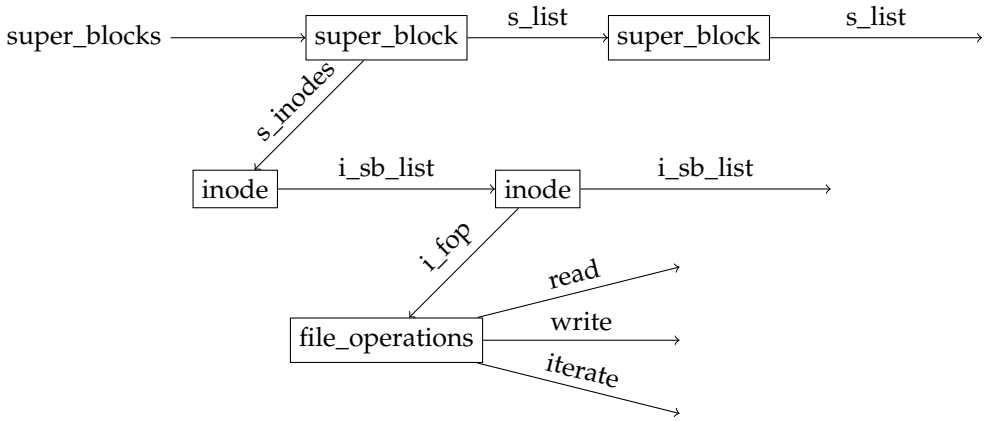

Figure 3. A graph of the `inode` obejcts. The `super_blocks` variable points to a linked list of all the `super_block` objects. Each `super_block` object contains a linked list of `inode` objects defined by the `s_inodes` and the `i_sb_list` fields of the `super_block` and the `inode` objects. Each `inode` object points via its `i_fop` field to the `file_operations` object which contains pointers to various functions, like `read`, `write`, `iterate`, etc.

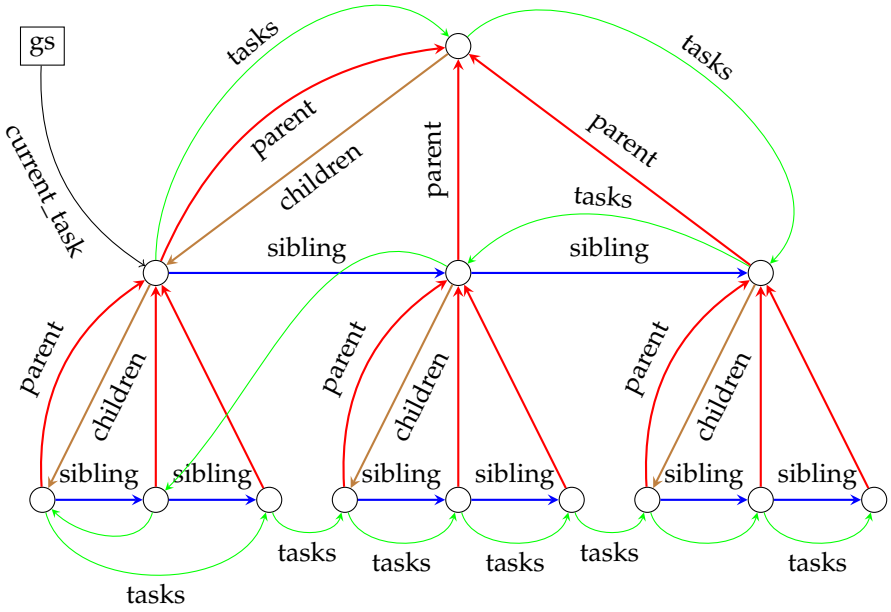

Figure 4. A graph of the `task_struct` obejcts. The `gs` register points to a structure, whose `current_task` field (black arrow) references the currently running `task_struct` object. From this object we can climb up the process tree using the `parent` field (red arrows). After reaching the root, we can descend using the `children` (brown arrows) and `sibling` fields (blue arrows). In addition, all the processes can be traversed using the `tasks` field (green arrows).

The verification procedure relies on the kernel symbols and the register values for locating and traversing the object graph. H-KPP intercepts the assignment to the `LSTAR` register, which signifies the completion of the kernel's initialization. At this point, H-KPP configures the VMX_PREEMPTION_TIMER, which induces periodic transitions to the hypervisor, thus allowing H-KPP to perform the periodic verification of the kernel objects. Table 4 summarizes the performed verifications.

Table 4. Kernel object verification summary.

Verification Type	Verification Subject	Referenced By
Hashing	Interrupt Descriptor Table System Call Table	`IDTR` register `sys_call_table` variable
Function Pointer	`inode.f_op` field `XYZ_seq_ops` structures	`super_blocks` variable `XYZ_seq_ops` variables
Set Equality Single Reference	`task_struct` objects obtained from 3 sources `task_struct.cred` field	`current_task` `current_task`

5. Results

5.1. Effectiveness

This section evaluates the effectiveness and the performance of H-KPP. In order to assess the effectiveness of H-KPP, we have developed a kernel module that mimics some aspects of the malicious behavior of various attacks. It may seem more natural to run an actual attack; unfortunately, the attacks we have found target older versions of the Linux kernel.

Our kernel module performs a sequence of operations that H-KPP should recognize as malicious. To demonstrate a successful recognition, we performed the following steps for each operation:

- rebooted the system;
- requested the kernel module to perform a specific operation;
- verified that a special message appeared in the hypervisor's log.

Regarding the last step, we note that the hypervisor outputs its log messages to the serial port in our current implementation, where these messages can be observed. We chose this communication channel only due to its simplicity. If needed, other communication channels, like Ethernet or USB, can be implemented. The following list summarizes the operations implemented by our kernel module.

1. The 1st operation demonstrates code injection, as performed by multiple attacks. While injecting a new code to various memory regions is possible, our kernel module demonstrates an injection to the stack. Then, the kernel module jumps to the injected code. The hypervisor reacts to the execution of unfamiliar code by writing a message to its log and resetting the system.
2. The 2nd and the 3rd operations demonstrate patching of the interrupt descriptor table (IDT) and the system call table, as performed by rootkit [19]. Specifically, the kernel module patches the keyboard interrupt entry of the IDT, thus allowing it to act as a keylogger. In the system call table, the kernel module patches the `getdents` system call, which allows it to hide files. The hypervisor reacts to both of these patches by writing a message to the log.
3. The 4th and the 5th operations demonstrate patching of the `/proc` inode's `inode.f_op.read` field and the patching of the `tcp4_seq_ops.show` field, which can enable an attacker to hide processes and network connections. In both cases, the hypervisor writes a log message.
4. The 6th operation demonstartes unlinking a given process from the process list, thus enabling an attacker to hide a process, even from the kernel itself. The hypervisor detects this attack using information from other sources and writes a log message.
5. The last operation demonstrates a privilege escalation attack by copying the credentials of a privileged process. Specifically, this operation alters the `task_struct.cred` field. The hypervisor recognizes that a particular credentials structure is referenced more than once and writes a log message.

5.2. Code Integrity Component Performance

In order to assess the performance of H-KPP, we have installed it in a virtualized environment running Ubuntu 20. Table 5 presents the exact specification of the testbed

environment. We used the Phoronix Test Suite (PTS) as a benchmarking tool. The PTS includes multiples programs that test various aspects of the system performance. We have selected a subset of these tests for inclusion in our performance analysis. Table 6 describes the selected tests.

Each test was executed in three configurations: without a hypervisor ("No HV"), with a hypervisor that does nothing ("Thin HV"), with H-KPP. Each test was executed several times, and the PTS displayed the average on the screen. Since the units of measurement are different for each test, in order to achieve uniformity, we display the result in percentages compared to the "No HV" configuration.

In Figure 5, we can see that the average overhead of the "Thin HV" configuration is 2.5%, and the average overhead of the "H-KPP" configuration is 6.2%. In contrast, the performance of a full hypervisor, like VirtualBox is above 38% [20].

Table 5. Testbed Specification.

Host CPU	Intel(R) Core(TM) i7-10610U
Host memory	16 GB
Host OS	Ubuntu 20.04.1 LTS
VMM	KVM/QEMU
Guest CPU	Intel(R) Core(TM) i7-10610U
Guest memory	8 GB
Guest OS	Ubuntu 20.04.1 LTS
Benchmarking Tool	Phoronix Test Suite (PTS) v5.2.1

Table 6. PTS Configuration.

Test Name	Description
unpack-linux	measures how long it takes to extract the .tar.xz Linux kernel package
encode-mp3	measures the time required to encode a WAV file to MP3 format
git	measures the time needed to carry out some sample Git operations
openssl	makes use of the built-in "openssl speed" benchmarking capabilities

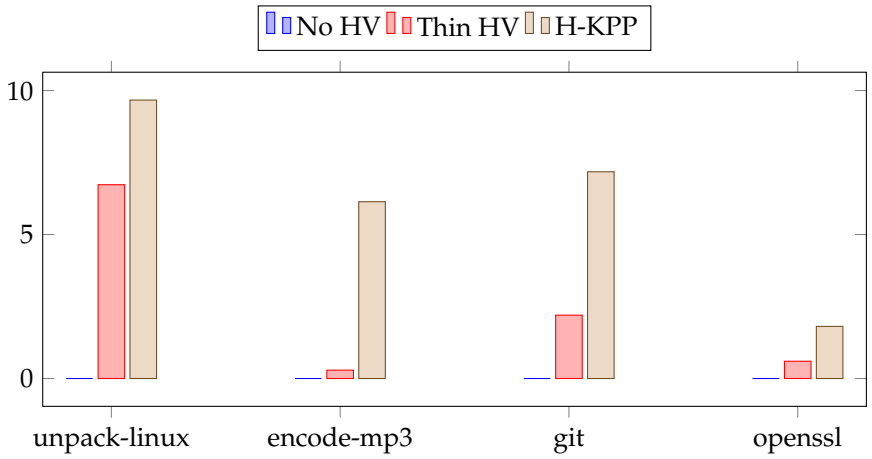

Figure 5. Test results in percents compared with "No HV" as a baseline. Lower is better.

5.3. DKOM Component Performance

In addition to its code integrity, H-KPP recognizes kernel object modification by performing periodic verification procedures. In this section, we analyze the performance of

each verification procedure in terms of the object graph size. We measured the difference between the timestamps at the beginning and the end of each procedure's execution. Then, depending on the procedure, we divided the result by the number of traversed objects. A potential operator of our system can achieve acceptable performance degradation by configuring the verification period according to the expected number of objects and the time required to verify each object. Table 7 summarizes our results.

Table 7. Verification time.

Verification Type	Time (ns)	Objects	Time Per Object (ns)
Hashing Interrupt Descriptor Table	6046	256	23
Hashing System Call Table	5408	447	12
`inode.f_op` field	8,432,219	118,425	71
`file_operation` objects	171,802	154	1115
`XYZ_seq_ops` structures	3852	4	962
Set equality `task_struct`	138,312	297	465
Single Reference `task_struct.cred`	72,250	297	243

6. Related Work

The main goal of hardware-assisted virtualization is the execution of multiple operating systems on a single hardware, as demonstrated by VirtualBox [21], VMware Workstation [22], Xen [23]. However, since its first introduction more than a decade ago, it has also been used for the construction of secure applications.

The applications cover a wide range of security areas. For example, Nitro [24], and its extension DRAKVUF [25], are Xen-based hypervisors for system call interception. They work by intercepting the system call mechanism of the x86 architecture, thus providing an OS-independent framework for virtual machine introspection (VMI). In particular, DRAKVUF introduces the notion of a stealth breakpoint for the interception of kernel functions. H-KPP uses a similar approach for intercepting the `do_jit` function, for example.

Another example is SBCFI [26]. SBCFI is a Xen-based hypervisor that provides control flow integrity verification for the underlying operating system. Currently, H-KPP does not include control flow hijacking countermeasures. However, it can be extended along the lines of SBCFI. Moreover, in theory, H-KPP's performance should be better due to its minimalistic design.

BitVisor [9] is a thin hypervisor that intercepts access to ATA hard disks to enforce storage encryption. A thin hypervisor is a preferred solution for security applications due to its low overhead compared to a full hypervisor. Like BitVisor, H-KPP is based on a thin hypervisor but provides different security services, namely protecting the kernel's code and data.

SecVisor [27] is another thin hypervisor that uses a secondary-level address translation (SLAT) and IOMMU to prevent unauthorized code execution in kernel mode. H-KPP is similar to SecVisor in its secondary-level address translation tables configuration. H-KPP differs from SecVisor in three aspects. Firstly, H-KPP introduces a new approach for rapid switching between two SLAT configurations, using code injection and the special `VMFUNC` instruction. This approach significantly improves the system performance. While it is difficult to make a direct comparison, we can see that SecVisor's performance in all the test cases was worse than Xen's, a full hypervisor. In contrast, H-KPP's performance is much better than VirtualBox's, which is also a full hypervisor. Secondly, unlike SecVisor, H-KPP can handle just-in-time generated code of BPFs. Thirdly, H-KPP protects the kernel's data structures and kernel's code protection.

Leon et al. [28] proposed a system that prevents the execution of unauthorized code in user mode. This system is also based on a thin hypervisor. Unlike H-KPP, the system proposed by Leon et al. targets user mode applications and therefore makes unrealistic assumptions about the kernel mode execution. In particular, it assumes that the kernel modules are compiled statically into the kernel and that the BPF mechanism is disabled.

However, the ideas presented by Leon et al. can be integrated into H-KPP, thus allowing it to protect the execution of user mode applications and the kernel. Resh et al. [29] proposed a similar system for the Windows OS.

PrivGuard [30] is a lightweight system for protecting kernel objects. PrivGuard is based on stack canaries and object duplication verifications performed during system calls. H-KPP is similar to PrivGuard in its protection of the `task_struct.cred` field. Unlike PrivGuard, H-KPP's protection cannot be circumvented by an attacker with kernel-mode privileges. In addition, H-KPP does not introduce any modifications to the kernel.

DLP-Visor [20] is a hypervisor-based system that attempts to prevent leakage of personal data by blocking the execution of certain system calls. DLP-Visor analyzes the system call execution at the level of the kernel itself. DLP-Visor's approach to limiting kernel object modification is based on access rights of user-mode application. Unlike DLP-Visor, H-KPP limits only direct modification of kernel object; all indirect modifications are allowed.

Volatility [8] is an offline tool for forensic investigation of memory snapshots. Obtaining a memory snapshot is not an easy task. ForenVisor [31], HyperSleuth [32] are two hypervisors that are capable of performing precise memory acquisition. Kiperberg et al. [33] described an adaptation of these ideas to modern operating systems.

Volatility includes multiple verification mechanisms intended to recognize malicious actions that were captured in the memory snapshot. H-KPP's kernel object verification procedure follows the lines of Volatility. Unlike Volatility, H-KPP can recognize an attack while it is being conducted.

Wang, Wu, and Liu proposed iCruiser [34] to protect link-based data structures of the kernel. iCruiser uses a special canary embedded in the linked objects to verify the integrity of the links. iCruiser requires recompilation of the kernel. In contrast, H-KPP's verification is not limited to link-based data structures and does not require kernel recompilation.

Srivastava, Erete, and Giffin proposed [35] to use a hypervisor to protect variables and structure fields from malicious modification. While the approach of the described system is more general than H-KPP's, to keep an acceptable performance degradation, the authors propose to partition the memory of kernel objects into protected and non-protected halves. This partition, obviously, requires kernel recompilation.

Graziano et al. [36] described a so-called "evolutionary" attack method on kernel objects that could not be recognized by traditional means. As an example of the attack's applicability, the authors could prevent the execution of a particular process without removing it from the process list. Graziano et al. described a hypervisor-based system that could be used to detect such attacks. While the current implementation of H-KPP is vulnerable to evolutionary attacks, it can be augmented with the simulator described by Graziano et al.

HUKO [37] introduces the notion of "subject-aware protection". HUKO defines three subjects: the kernel, a trusted kernel module, an untrusted kernel module. The kernel is permitted to perform any operation on a kernel object. Trusted kernel modules are permitted to perform only some operations. Untrusted kernel modules are permitted to perform only a tiny subset of operations. HUKO's object labeling mechanism relies on information delivered to it from the modified Linux kernel. H-KPP is similar to HUKO in that they both utilize a separate SLAT for each environment. Unlike H-KPP, HUKO is based on Xen, a full hypervisor. As a result, the HUKO's performance is not better than the performance of Xen.

7. Conclusions

We have described a hypervisor-based system for protecting kernel code and data. While the idea of protecting kernel code using a hypervisor is not new, H-KPP adapts this idea to new operating system facilities, like BPF. In addition, we have presented a novel approach for SLAT switching, which is based on code injected performed by the hypervisor. This SLAT switching approach allows the system to remain efficient and secure

simultaneously. Unlike other solutions, H-KPP is based on a thin hypervisor. As a result, H-KPP has a negligible performance overhead making it applicable in real-world scenarios.

In addition, unlike some previously described systems, H-KPP does not require any modifications of the underlying kernel. However, if such modifications are possible, more general and precise protection can be achieved using object partitioning [35].

Author Contributions: Conceptualization, M.K.; methodology, N.J.Z. software, M.K. and N.J.Z.; Validation, N.J.Z.; formal analysis, M.K.; investigation, M.K.; resources, N.J.Z.; data curation, M.K.; writing—original draft preparation—M.K.; writing—review and editing N.J.Z.; visualization M.K.; supervision N.J.Z.; project administration N.J.Z.; funding acquisition N.J.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
DKOM	Direct Kernel Object Manipulation
eBPF	extended Berkley Packet Filter
MSR	Module Specific Register

References

1. Microsoft. Available online: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention (accessed on 27 March 2022).
2. Microsoft. Available online: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/driver-signing (accessed on 27 March 2022).
3. Stojanovski, N.; Gusev, M.; Gligoroski, D.; Knapskog, S.J. Bypassing Data Execution Prevention on Microsoft Windows XP SP2. In Proceedings of the Second International Conference on Availability, Reliability and Security (ARES'07), Vienna, Austria, 10–13 April 2007; pp. 1222-1226. [CrossRef]
4. Kleissner, P. Stoned Bootkit. In Proceedings of the Black Hat, Las Vegas, NV, USA, 25–30 July 2009; pp. 5–7.
5. Ermolov, M.; Shishkin, A. Microsoft Windows 8.1 Kernel Patch Protection Analysis. Available online: https://www.ptsecurity.com/upload/corporate/ru-ru/analytics/Windows_81_Kernel_Patch_Protection_Analysis.pdf (accessed on 14 May 2022)
6. Chen, H.; Mao, Y.; Wang, X.; Zhou, D.; Zeldovich, N.; Kaashoek, M.F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In Proceedings of the Second Asia-Pacific Workshop on Systems, Shanghai China, 11–12 July 2011; pp. 1–5.
7. CVE Details. Available online: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33 (accessed on 27 March 2022).
8. The Volatility Foundation. Available online: https://www.volatilityfoundation.org/ (accessed on 27 March 2022).
9. Shinagawa, T.; Eiraku, H.; Tanimoto, K.; Omote, K.; Hasegawa, S.; Horie, T.; Hirano, M.; Kourai, K.; Oyama, Y.; Kawai, E.; et al. Bitvisor: A thin hypervisor for enforcing i/o device security. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Washington, DC, USA, 11–13 March 2009.
10. Rhee, J.; Riley, R.; Xu, D.; Jiang, X. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In Proceedings of the 2009 International Conference on Availability, Reliability and Security, Fukuoka, Japan, 16–19 March 2009; pp. 74–81.
11. KernelNewbies. Available online: https://kernelnewbies.org/Linux_2_6_8 (accessed on 27 March 2022).
12. The Kernel Development Community. Available online: https://www.kernel.org/doc/html/v4.13/admin-guide/module-signing.html (accessed on 14 May 2022).
13. Butler, J. DKOM (direct kernel object manipulation). In Proceedings of the Black Hat Windows Security, Seattle, WA, USA, 27–28 January 2004.

14. Kasspersky Lab. Overview of the Latest Windows OS Kernel Exploits Found in the Wild. CanSecWest/BlueHat. 2019. Available online: https://github.com/oct0xor/presentations/blob/master/2019-02-Overview%20of%20the%20latest%20Windows%20OS%20kernel%20exploits%20found%20in%20the%20wild.pdf (accessed on 27 March 2022).
15. Kiperberg, M. Preventing malicious communication using virtualization. *J. Inf. Secur. Appl.* **2021**, *61*, 102871. [CrossRef]
16. EFI Forum, Inc. *Unified Extensible Firmware Interface (UEFI) Specification*; UEFI Forum, Inc.: Beaverton, OR, USA, 2021.
17. Marco-Gisbert, H.; Ripoll, I. On the Effectiveness of Full-ASLR on 64-bit Linux. In Proceedings of the In-Depth Security Conference, Vienna, Austria, 18–21 November 2014.
18. Goldshtein, S. *The Next Linux Superpower: eBPF Primer*; USENIX Association: Dublin, Ireland, 2016.
19. Baliga, A.; Iftode, L.; Chen, X. Automated containment of rootkits attacks. *Comput. Secur.* **2008**, *27*, 323–334. [CrossRef]
20. Kiperberg, M.; Amit, G.; Yeshooroon, A.; Zaidenberg, N.J. Efficient DLP-visor: An efficient hypervisor-based DLP. In Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Melbourne, Australia, 10–13 May 2021.
21. Oracle, VirtualBox. Available online: http://https://www.virtualbox.org/ (accessed on 27 March 2022).
22. VMware, Workstation Pro. Available online: https://www.vmware.com/products/workstation-pro.html (accessed on 27 March 2022).
23. Barham, P.; Dragovic, B.; Fraser, K.; H.S.; Harris, T.; Ho, A. Xen and the art of virtualization. *Acm Sigops Oper. Syst. Rev.* **2003**, *37*, 164–177. [CrossRef]
24. Pfoh, J.; Schneider, C.; Eckert, C. Nitro: Hardware-based system call tracing for virtual machines. In *International Workshop on Security*; Springer: Berlin/Heidelberg, Germany, 2011.
25. Lengyel, T.K.; Maresca, S.; Payne, B.D.; Webster, G.D.; Vogl, S.; Kiayias, A. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In Proceedings of the 30th Annual Computer Security Applications Conference, New Orleans, LA, USA, 8–12 December 2014.
26. Petroni, N.L., Jr.; Hicks, M. Automated detection of persistent kernel control-flow attacks. In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VR, USA, 31 October–2 November 2007.
27. Seshadri, A.; Luk, M.; Qu, N.; Perrig, A. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, Washington, DC, USA, 14–17 October 2007; pp. 335–350.
28. Leon, R.S.; Kiperberg, M.; Zabag, A.A.L.; Resh, A.; Algawi, A.; Zaidenberg, N.J. Hypervisor-Based White Listing of Executables. IEEE Secur. Priv. **2019**, *17*, 58–67. [CrossRef]
29. Resh, A.; Kiperberg, M.; Leon, R.; Zaidenberg, N.J. Preventing execution of unauthorized native-code software. *Int. J. Digit. Content Technol. Its Appl.* **2017**, *11*, 72–90.
30. Qiang, W.; Yang, J.; Jin, H.; Shi, X. PrivGuard: Protecting sensitive kernel data from privilege escalation attacks. *IEEE Access* **2018**, *6*, 46584–46594. [CrossRef]
31. Qi, Z.; Xiang, C.; Ma, R.; Li, J.; Guan, H.; Wei, D.S. ForenVisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Trans. Cloud Comput.* **2016**, *5*, 443–456. [CrossRef]
32. Martignoni, L.; Fattori, A.; Paleari, R.; Cavallaro, L. Live and trustworthy forensic analysis of commodity production systems. In Proceedings of the International Workshop on Recent Advances in Intrusion Detection, Toulouse, France, 2–4 October 2000; Springer: Berlin/Heidelberg, Germany, 2010.
33. Kiperberg, M.; Leon, R.; Resh, A.; Algawi, A.; Zaidenberg, N. Hypervisor-assisted atomic memory acquisition in modern systems. In Proceedings of the International Conference on Information Systems Security and Privacy, Prague, Czech Republic, 23–25 February 2019; SCITEPRESS Science And Technology Publications: Setúbal, Portugal, 2019.
34. Li, W.; Wu, D.; Liu, P. iCruiser: Protecting Kernel Link-Based Data Structures with Secure Canary. In Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Vienna, Austria, 1–3 August 2016.
35. Srivastava, A.; Erete, I.; Giffin, J. *Kernel Data Integrity Protection via Memory Access Control*; Georgia Institute of Technology: Atlanta, GA, USA, 2009.
36. Graziano, M.; Flore, L.; Lanzi, A.; Balzarotti, D. Subverting operating system properties through evolutionary DKOM attacks. In Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Sebastián, Spain, 7–8 July 2016; Springer: Cham, Switzerland, 2016.
37. Xiong, X.; Tian, D.; Liu, P. *Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions*; In Proceedings of the NDSS San Diego, CA, USA, 6–9 February 2011.