Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection

Sergej Proskurin[1], Julian Kirsch[1], and Apostolis Zarras[2]

[1] Technical University of Munich
{proskurin,kirschju}@sec.in.tum.de
[2] Maastricht University
apostolis.zarras@maastrichtuniversity.nl

Abstract The growing complexity of modern malware drives security applications to leverage Virtual Machine Introspection (VMI), which provides a complete and untainted view over the Virtual Machine state. To benefit from this ability, a VMI-aware Virtual Machine Monitor (VMM) must be set up in advance underneath the target system; a constraint for the massive application of VMI. In this paper, we present *WhiteRabbit*, a VMI framework comprising a microkernel-based VMM that transparently virtualizes a running Operating System, on-the-fly, for the purpose of forensic analysis. As a result, the systems to be analyzed do not have to be explicitly set up for VMI a priori. After its deployment, our framework exposes VMI services for remote applications: *WhiteRabbit* implements a LibVMI interface that enables it to be engaged by popular VMI applications remotely. Our prototype employs Intel as well as ARM virtualization extensions to take over control of a running Linux system. *WhiteRabbit*'s on-the-fly capability and limited virtualization overhead constitute an effective solution for malware detection and analysis.

1 Introduction

Malware can be executed with the same privileges as sensitive parts of the Operating System (OS). Once installed, it can hide itself from the OS and security applications. To tackle this, researchers moved security applications into a highly privileged environment realized through virtualization [1]. In essence, virtualization adds a software layer, the Virtual Machine Monitor (VMM), that implements a virtual hardware interface. This interface, the Virtual Machine (VM), manages an execution environment for guest OSes. A VMM has a complete view over the entire VM state and provides isolation from guest VMs. This forbids malware inside a VM to deceive applications executing as part of the VMM.

Security applications use virtualization for different purposes, including *malware detection* and *analysis* [1,2,3,4,5,6] as well as *system integrity validation* [7,8]. To examine the state of a guest OS Virtual Machine Introspection (VMI) must be applied [2]. The state of a guest comprises a vast amount of binary information that needs interpretation. Thus, every VMI application uses additional

semantic knowledge to map the binary information, e.g., to high-level OS kernel data structures; widely-known as the *semantic gap* [1]. Yet, conventional approaches require the systems to have a VMI-aware VMM *before* operation. This increases the administrative overhead and constraints the employment of VMI.

In this paper, we combine VMI along with on-the-fly virtualization concepts to address the previously stated limitations. We design and implement *WhiteRabbit*, a framework for forensic analysis that can be transparently deployed on general purpose systems by moving the live OS into a dynamically initialized virtual environment. First, we develop a VMM that is capable of seamlessly taking over control of a running OS on-the-fly. Next, we outline VMI mechanisms that enable forensic analysis from outside of the virtualized OS. To provide even more flexibility, we incorporate essential VMI functionality that can be used through a LibVMI interface. After its deployment, our prototype acts as a vehicle providing VMI services to remote applications. Contrary to existing VMI solutions, our system does not require the target OS to be set up for VMI in advance. Instead, we deploy *WhiteRabbit* spontaneously on general purpose systems. As a result, the target systems are transformed into monitored environments that can be remotely controlled by custom or existing VMI tools.

In summary, we make the following main contributions:

- We elaborate the design and architecture of the *WhiteRabbit* VMI framework, a microkernel-based VMM that transparently shifts a live OS into a VM on-the-fly without leaving any traces.
- We implement a prototype that is able to virtualize Linux OSes on-the-fly by leveraging virtualization extensions of Intel as well as ARM architectures.
- We develop a LibVMI interface to facilitate remote VMI through existing LibVMI applications.

2 Virtualization Technology

Intel Virtualization Technology. Intel VT-x contains a set of Virtual Machine Extensions (VMX) that simplifies the process of virtualization. These introduce two additional modes: *VMX root* and *VMX non-root*. Intel VT-x duplicates the four privilege levels (protection rings, numbered from 0 to 3), to provide full compatibility for systems running in both the VMX root and VMX non-root. Typically, a VMM operates in the high-privileged VMX root and the guest operates in the less-privileged VMX non-root. Before the guest can be initiated, the VMM must allocate and initialize a hardware defined data structure called Virtual Machine Control Structure (VMCS): it manages transitions between the VMM and a particular guest. This is done by holding the guest virtual CPU state that is loaded on VM entries and the host CPU state that is restored on VM exits. If the guest requires multiple CPUs, the VMM must maintain one VMCS for each virtual CPU. The VMCS also holds execution control fields that determine the guest's behavior. By configuring these fields, the VMM defines the set of events that will trap into the VMM. Further organization of the VMCS comprises control fields determining the behavior during VM entries and exits.

ARM Virtualization Technology. ARM distributes software execution across different *privilege levels* on ARMv7, which are called *exception levels* on ARMv8. In this paper, we use the term exception levels (ELs) over privilege levels. Different ELs restrict access to privileged resources. Similar to x86, the execution of OSes is distributed across two exception levels: EL0 and EL1. User applications execute in the less privileged EL0 and the OS kernel in the higher privileged EL1. Systems with hardware virtualization extensions introduce EL2 that is dedicated for VMMs with the highest privileges. Guest VMs in EL0 and EL1 trap into the higher privileged VMM in EL2 (e.g., on privileged instruction fetches).

3 Threat Model

We assume an adversary with root privileges, who can fully control the OS and all security-relevant parts of the kernel. Thus, she can inspect the OS for agents in form of processes or kernel modules. Yet, she cannot perform Direct Kernel Structure Manipulation attacks by exploiting the semantic gap to evade VMI. While the attacker is not concerned about virtualized systems, she will abort her attack on disclosure of an analysis framework. Thus, she can employ techniques that reveal the presence of virtualization-based analysis frameworks—while she can carve the guest's memory, she cannot use Direct Memory Access (DMA) or operate with higher privileges than *WhiteRabbit*. Also, she disregards side channel attacks against VMMs, especially those that are based upon a flawed CPU architecture [9,10]. Further, the attacker has access to the system's registers and can search the file system for indications of an analysis framework.

Even though *WhiteRabbit* provides a stealthy environment, VMI applications that are built upon it may employ services detectable by the adversary. For instance, *WhiteRabbit* does not provide any means to cloak in-guest instrumentation. For this, *WhiteRabbit* could be extended to support multiple guest-physical memory views to satisfy integrity checks as it has been shown by work based on the Xen altp2m subsystem [6,11] or similar techniques [12] employing Second Level Address Translation to coordinate access to guest-physical memory.

4 The WhiteRabbit VMM

WhiteRabbit is a microkernel-based VMM designed for on-the-fly virtualization of live OSes. For this, we leverage Intel and ARM virtualization extensions. The Xen Project and Linux KVM VMMs could be adapted for on-the-fly virtualization. Yet, both would entail a considerable amount of functionality that would remain unused if applied for our purpose. Figure 1 summarizes our architecture for x86-64 and ARM. *WhiteRabbit* comprises custom subsystems and supplies VMI capabilities to remote parties. To tackle potential exposures, *WhiteRabbit* hides from the virtualized OS by using Second Level Address Translation (SLAT). It implements a LibVMI interface for access by remote parties enabling introspection of the virtualized OS. As a result, *WhiteRabbit* facilitates the use of prevalent LibVMI tools on systems that have not been set up for VMI.

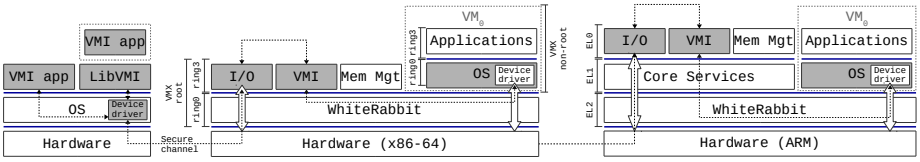

Figure 1: A remote host (left) analyses the on-the-fly virtualized system in the middle (x86-64) and right (ARM). Shaded components are involved in VMI.

The microkernel character of *WhiteRabbit* reduces the size and complexity of the VMM: we implement only essential VM-maintenance functionality inside the high-privileged protection *ring 0* on Intel and across *EL1* and *EL2* on ARM. We move additional components required for memory management, remote communication, and VMI into the user space in *ring 3* on Intel and *EL0* on ARM. This architectural choice isolates user space components from the guest and hardens the system; user space component crashes do not affect the entire system.

WhiteRabbit provides a memory management system that cuts off dependencies to the virtualized OS. Therefore, *WhiteRabbit* is not bound to the OS memory management which could be observed and controlled by adversaries. During initialization, *WhiteRabbit* must allocate memory required for its operation by means of the buddy allocator of the OS or by adjusting the system's page tables. Either way, since the VMM hiding technique utilizes SLAT, the allocated memory blocks become invisible for the guest OS. Besides, *WhiteRabbit* uses only custom provided functionality without the need for any guest OS services. Thus, (assuming OS-independent deployment) it could subvert different OSes.

To remotely access the VMI functionality, *WhiteRabbit* maintains I/O drivers that manage a secure communication channel, whose operation is either entirely cut off or must be multiplexed with the guest. To isolate the communication channel, it should be realized through unused I/O devices or by hardware multiplexing of I/O resources, e.g., through Intel VT-d or VT-c and ARM SMMU.

5 On-the-Fly Virtualization

WhiteRabbit must be deployed on the target system to dynamically virtualize a running OS. We distinguish between *OS-dependent* and *OS-independent* deployment strategies. Both can be performed locally or remotely. The OS-dependent strategy requires a kernel module to set up *WhiteRabbit* underneath the target OS. The kernel module can comprise either (*i*) the entire *WhiteRabbit* implementation or (*ii*) act as a means for transportation. The former approach implements the *WhiteRabbit* functionality as part of the kernel module. This strategy must be regarded critically as it allows *WhiteRabbit* to use target OS services: employed services might reveal the presence of *WhiteRabbit* or provide false information controlled by malware. The latter uses the kernel module as a loader to deploy *WhiteRabbit* in form of an OS-independent binary into memory. While the loader requires OS services, *WhiteRabbit* must not be OS-agnostic.

The OS-independent strategy uses a DMA channel. By taking the role of the bus master, hardware devices can initiate communication and hence arbitrary access (assuming deactivated IOMMU) to other node's memory. Thus, DMA capable interfaces (e.g., FireWire and Thunderbolt) can be abused to transparently load *WhiteRabbit* into the system's memory. Yet, after the code injection, the system requires additional means to execute the payload. One idea is to use Intel Active Management Technology (AMT) to remotely launch *WhiteRabbit*.

The tasks of *WhiteRabbit* are best described according to the taxonomy of Popek and Goldberg: a VMM is a modular *control program*, whose modules belong to three groups comprising an *allocator*, a *dispatcher*, and an *interpreter* [13]. We distribute the tasks of *WhiteRabbit* across these groups. For simplicity, we assume in the following *WhiteRabbit* is deployed as a kernel module.

Allocator. The allocator places a running OS into a virtual environment without letting the target OS understand the change. The allocator leverages hardware virtualization extensions to provide the guest with the illusion of having unrestricted access to all system's resources. This process is strongly hardware-dependent. As such, we discuss the necessary steps for Intel and ARM.
`Intel:` The allocator records the system's state (i.e., before OS virtualization) in the VMCS guest-state area that holds control registers determining the guest's behavior. Also, it sets up the host's state and registers the entry point of the VMM that will be executed at every VM exit[3] in VMX root. *WhiteRabbit* grants direct hardware access to the VM and does not emulate any hardware resources.
`ARM:` While ARM's virtualization support closely resembles its x86-64 counterpart, it entails peculiarities. For instance, ARM cannot initialize EL2 from a less-privileged exception level: in case the system has not set up EL2 exception vectors at system boot, there is no way to retrospectively place these vectors. That is, to enable virtualization the boot loader launches the OS kernel in EL2 before entering EL1. There, the OS installs a general purpose hypervisor stub (the lowvisor) initializing the aforementioned exception vectors [14]. This lowvisor allows Linux subsystems in EL1 (e.g., Linux KVM) to reinitialize the exception vectors through a hypercall and take control over EL2. Thus, after ensuring that the lowvisor has not been occupied, the allocator takes control of EL2.

In both cases, the allocator enforces events of interest to trap into the VMM for analysis. These comprise hardware events, execution of certain instructions, and access to critical system registers. Besides, the allocator sets up subsystems (e.g., memory management and device drivers) to manage the system's hardware. This way, *WhiteRabbit* becomes independent from the virtualized OS. To provide stealth, the allocator hides *WhiteRabbit* from the guest by using SLAT.

Dispatcher. The dispatcher is triggered on every VM exit and can be regarded as a scheduler. The dispatcher analyzes the VM exit reason, based on which it decides which operation to perform. It is the interpreter (described in the following) that is responsible to perform tasks on behalf of the dispatcher.

[3] The terms *VM entry* and *VM exit* refer to Intel's terminology describing transitions from the VMM into the guest and reverse. In this paper, we use these terms to describe transitions on the ARM architecture as well.

Interpreter. Hardware virtualization extensions define a class of *unconditionally* and *conditionally trapped* instructions. The former class comprises privileged instructions that always trigger VM exits. The latter class of instructions trigger VM exits only if the allocator has configured them to trap. The same applies to hardware events. The interpreter simulates guest instructions and hardware events that trap into the VMM and appropriately updates the guest's state. Apart from that, as the interpreter is capable of manipulating the guest's state, it is used as our framework's VMI subsystem. For this, the interpreter leverages memory and device management services that have been set up by the allocator.

6 Bridging the Semantic Gap

WhiteRabbit allows remote hosts to analyze the virtualized OS (Figure 1). For this, VMI tools have to interpret the vast amount of binary information (i.e., the guest's state). To bridge this semantic gap, *WhiteRabbit* offers (*i*) in-band and (*ii*) out-of-band delivery as well as (*iii*) derivative view generation patterns [15].

In-band delivery. This pattern involves the guest OS to collect semantic information. For this, *WhiteRabbit* allows remote VMI tools to inject kernel modules into the guest. Similar to X-TIER [5], once *WhiteRabbit* receives a kernel module, the VMI component (Figure 1) will process the module to provide a generic and OS-agnostic module representation before injecting it into the guest. To simplify the VMI component, these steps could be prepared by the remote host.

First, the VMI component allocates memory to rearrange the module's sections. *WhiteRabbit* cannot just add physical memory to the VM, as the guest's memory management system has tracked all of the available physical memory. Instead, it obtains the memory from the local memory pool that has been extracted from the guest beforehand (Section 4). Then, the VMI component incorporates code required to communicate with *WhiteRabbit*. This code comprises wrappers responsible for relaying calls to external functions and announcing the end of the module's execution through hypercalls. Finally, *WhiteRabbit* adjusts the page tables of the interrupted guest process, temporarily uncovers the module's memory via SLAT, and adjusts the guest's instruction and stack pointer. To prevent the module from being interrupted and thus potentially revealed, *WhiteRabbit* must deactivate the timer interrupt and intercept external interrupts.

Out-of-band delivery. *WhiteRabbit* implements an interface for LibVMI; a library to dynamically extract and control the VM's state. To bridge the semantic gap, LibVMI uses out-of-band kernel symbol information that is delivered ahead-of-time. Also, LibVMI offers an API for Volatility tools. This way, *WhiteRabbit* offers analysis via custom and prevalent LibVMI and Volatility forensics tools.

Derivation. Derivative view generation benefits from the fact that critical static in-guest data structures are rooted in hardware [16]. E.g., one can build a chain of references between the syscall dispatcher and an immutable, hardware anchor (the `IDTR` or fast system call `MSR` registers on Intel). It is difficult to identify and utilize such hardware anchors. Yet, through LibVMI, *WhiteRabbit* facilitates

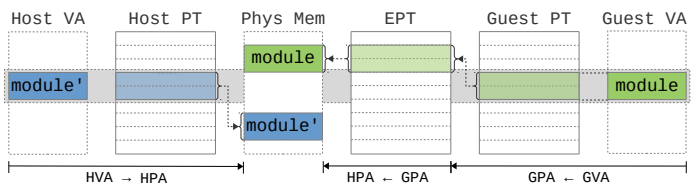

Figure 2: Hiding *WhiteRabbit* through relocation and the SLAT mechanism.

remote tools to derive a view of the guest. Besides, critical guest data structures are *evasion-evident* if they are rooted in hardware [16]. *WhiteRabbit* can observe changes to such data structures. By additionally protecting all elements along the chain from the hardware anchor to the data structure, the data structure becomes *evasion-resistant*: any modification along this chain can be detected by matching the integrity of the system's configuration with a known value.

7 Hiding Techniques

Split-personality malware behaves differently if it believes it is being monitored [17]. Given that perfect VM *transparency* (ability of being indistinguishable with real hardware) is not feasible [18], *WhiteRabbit* hardens disclosure of its presence. We associate the memory footprint of *WhiteRabbit* with the *execution environment* and kernel module traces with the *hardware* category of the anti-virtualization taxonomy [19]; *WhiteRabbit* uses SLAT to sidestep both.

Execution environment. A malicious guest application with sufficient privileges can carve the physical memory, e.g., for signatures that reveal the presence of a VMM. To prevent an exposure in memory, we utilize the system's SLAT tables such that the physical memory holding *WhiteRabbit* becomes invisible to the guest: access to this memory is intercepted and redirected to an empty page.

A problem arises if we deploy *WhiteRabbit* as a kernel module: the first instruction of the guest will be one of the last instructions of the kernel module that has virtualized the OS: the module (inside the VM) needs to return to the Linux kernel. At this point, the kernel initializes remaining entries of the `struct module` that resides in the module's memory. Also, the kernel frees the memory that has been used for the initialization of the kernel module and is not part of its core sections. Therefore, the kernel must access the memory previously made invisible to the guest OS. We discuss the solution of this issue in the following.

Hardware. *WhiteRabbit* must not leave any traces inside of the guest if it was deployed as a kernel module. As we have shown, the utilization of SLAT alone leads to an issue during module initialization. To solve this, we force the guest to free the memory and data structures linked to *WhiteRabbit* but simultaneously continue its execution under cover of SLAT (Figure 2). *WhiteRabbit* must relocate its memory to another location. The relocated module is shown as `module'` in the figure. To avoid address relocations, we map `module'` to the same virtual address space as the original `module`. The setup comprises the guest's page

tables and SLAT mapping the module's guest-virtual-addresses (GVAs) to the original host-physical-addresses (HPAs) and the host's page tables mapping the host-virtual-addresses (HVAs) to the relocated HPAs. This way, the host can use the original virtual addresses to address `module'` (HVAs correspond to GVAs).

Finally, the `module` returns a negative value at the end of its `init` routine, as soon as the OS has been virtualized. Alternatively, the module can initialize a work-queue that initiates a clean module destruction without making the kernel suspicious. In both cases, the guest OS deallocates all data structures associated with *WhiteRabbit*. To ensure that the contents of these data structures cannot be reconstructed from memory, we zero them out from outside of the VM.

8 Evaluation

We evaluated *WhiteRabbit* by first analyzing its effectiveness in regard to anti-debugging and anti-virtualization techniques. Second, to demonstrate *WhiteRabbit*'s practicality, we investigated the induced virtualization overhead and compared it with the popular Xen Project hypervisor and Linux KVM.

8.1 Effectiveness

We have virtualized a Linux and were able to single-step and extract analysis-sensitive processes. Therefore, we employed state-of-the-art anti-debugging and anti-virtualization techniques that impede dynamic analysis and stop execution as soon as they believe they reside in a sandbox. The following summarizes these techniques and shows that they are rendered ineffective against *WhiteRabbit*.

Anti-debugging. Linux bares an API via the `ptrace` system call to allow debugging of user space processes. This API utilizes hardware-based memory watchpoints and single-stepping capabilities, as well as the ability to access foreign address spaces. However, `ptrace` entails that any tracee can be traced by exactly one process. This property is abused for anti-debugging: hostile machine code can use `ptrace` to trace itself. Consequently, if `ptrace` fails, the caller is aware of a tracing application; if it succeeds, no other tracer will be able to attach itself to this process. While this situation can be side-stepped by intercepting calls to `ptrace` and adjusting the return values, the idea can be extended to multiple malicious processes tracing each other to completely hinder debugging.

Besides, debuggers (e.g., *gdb* and *lldb*) leave environment artifacts that can reveal debuggers. These artifacts include (*i*) address space layout randomization allocating the `text`, `data`, and `vDSO` pages at unusual addresses, (*ii*) environment variables, (*iii*) the parent process' name containing the debugger's name, and (*iv*) software breakpoints non-transparently placed into the tracee's address space. We have open sourced a debugger detection tool implementing the above. [4]

WhiteRabbit does not make use of any of the above techniques. In fact, *WhiteRabbit* does not leave in-guest user space artifacts and thus cannot be detected by these and similar anti-debugging mechanisms.

[4] `https://github.com/kirschju/debugmenot`

Anti-virtualization. We have armed the virtualized system with custom and publicly available sandbox-detection tools including *paranoid fish*, *al-khaser*, and *virt-what*. These tools use (*i*) *static heuristics*, (*ii*) *low-level system properties*, and (*iii*) *user behavior artifacts* to disclose sandboxed environments. Except for different timing behavior (Section 9) none of these tools detected *WhiteRabbit*.

Static heuristics target virtualization artifacts (e.g., drivers, execution environment and hardware configuration, vendor information as well as memory and file system artifacts) that are specific for virtual environments. *WhiteRabbit* aims at having as little discrepancies to the physical machine as possible: it does not adjust any system configuration and leaves no guest-visible artifacts (Section 7).

Hardware artifacts are timing properties and effects of imperfect instruction and device emulation. *WhiteRabbit* permits the guest to directly access the hardware without emulating any hardware devices. Thus, it does not expose itself through such indicators. On the other hand, timing can reveal the VMM; we exposed *WhiteRabbit* by comparing the time of unconditionally trapped instructions with reference values. However, with today's omnipresent virtualization technology, it is insufficient to reveal the virtual environment alone (Section 9).

User behavior artifacts target the system's credibility by observing its state and configuration, including mouse cursor activity or an unusually small size of the hard drive or memory. Sophisticated systems check wear-and-tear relics, e.g., log files, browser history, and network behavior [20]. Such artifacts lose relevance, as *WhiteRabbit* virtualizes production systems with realistic wear-and-tear relics.

Careless VMI tools that are built upon *WhiteRabbit* might implement less-stealthy techniques. To address this, *WhiteRabbit* provides the necessary means to intercept critical events. Thus, VMI tools must handle such events and return inconspicuous register values to cloak analysis.

8.2 Performance

It is crucial that both the VMM and VMI tools affect the system's performance as little as possible. Because we highlight *WhiteRabbit* as a vehicle for generic VMI tools, our performance evaluation focuses on the *virtualization overhead*; it does not consider VMI tools built upon *WhiteRabbit*. We deem the overhead of VMI tools out of scope, as the performance highly depends on their purpose.

We applied our prototype to the Linux kernel v4.13 on top of an Intel Skylake microarchitecture based host with an Intel Core i7-6700 CPU (3400 MHz) and one active core (a limitation of our prototype). We configured the *performance* CPU frequency scaling governor to avoid performance drops, e.g., due to power consumption oriented configurations. To estimate the virtualization overhead, we carried out three experiments including a set of CPU- and memory-intensive macro- and micro-benchmarks. The results are mean values over three runs.

First, we compared the virtualization overhead of *WhiteRabbit* with Xen v4.11 and Linux KVM. To simulate a comparable load, we have granted only one core to all VMMs (i.e., we pinned the VMM and guest to the same physical core). Interestingly, with active Intel Turbo Boost technology, Xen outperformed the bare metal host. As such, we deactivated Turbo Boost to avoid different

Table 1: Virtualization overhead (OHD) of *WhiteRabbit*, Xen and KVM measured by the Phoronix Test Suite v7.6.0 on x86-64.

Benchmark (*unit*)	**w/o**	**KVM**	**(OHD)**	**Xen**	**(OHD)**	**WhiteRabbit**	**(OHD)**
Blake2 (*Cycles/Byte*)	5.94	5.94	(0.00%)	5.94	(0.00%)	5.94	(0.00%)
C-Ray (*s*)	107.08	108.56	(1.38%)	107.67	(0.55%)	107.09	(0.00%)
Gzip Compression (*s*)	11.50	12.06	(4.86%)	11.98	(4.17%)	11.74	(2.08%)
John-the-Ripper DES (*Real C/s*)	5,419,000	5,340,000	(1.45%)	5,394,000	(0.46%)	5,417,667	(0.02%)
John-the-Ripper MD5 (*Real C/s*)	16,844	16,583	(1.54%)	16,748	(0.56%)	16,822	(0.13%)
N-queens (*s*)	216.70	220.94	(1.95%)	217.77	(0.49%)	216.80	(0.04%)
OpenSSL (*Signs/s*)	145	141.83	(2.18%)	142.53	(1.70%)	144.70	(0.20%)
7-Zip Compression (*MIPS*)	4,603	3,736	(18.83%)	3,988	(13.36%)	4,443	(3.47%)
RAMspeed Integer (*MB/s*)	17,370.73	16,630.25	(4.26%)	16,942.71	(2.46%)	17,016.09	(2.04%)
RAMspeed Floating Point (*MB/s*)	17,734.84	16,744.56	(5.58%)	16,875.53	(4.84%)	16,861.26	(4.92%)

Table 2: SPEC CPU2017, in *sec.*

Benchmark	**w/a**	**WhiteRabbit**	**OHD**
600.perlbench_s	282	286	(1.41%)
602.gcc_s	409	419	(2.44%)
605.mcf_s	624	641	(2.72%)
620.omnetpp_s	382	406	(6.28%)
623.xalancbmk_s	283	294	(3.88%)
625.x264_s	378	378	(0.00%)
631.deepsjeng_s	357	363	(1.68%)
641.leela_s	460	460	(0.00%)
648.exchange2_s	264	265	(0.37%)
657.xz_s	2220	2379	(7.16%)

Table 3: Lmbench 3.0, in $\mu sec.$

Benchmark	**w/a**	**WhiteRabbit**	**OHD**
fork+execve	50.04	58.23	(16.36%)
fork+/bin/sh	254.36	289.84	(13.94%)
pipe	1.51	1.65	(9.27%)
protection fault	0.30	0.31	(3.33%)
read	0.09	0.09	(0.00%)
select 500 fd	2.46	2.52	(2.38%)
select 500 TCP fd	8.16	8.35	(2.32%)
signal handle	0.66	0.65	(1.51%)
sock	1.99	2.07	(4.02%)
write	0.05	0.06	(19.99%)

microcode decisions in regard to performance states. We executed a set of CPU- and memory-intensive macro-benchmarks of the *Phoronix Test Suite v7.6.0*. The results are shown in Table 1, which we divided into CPU- (upper part) and memory-intensive (lower part) benchmarks. Overall, the results indicate only a minor overhead for all candidates. Yet, *WhiteRabbit* outperforms Xen and KVM. While KVM produces less than 4.02% CPU and 4.92% memory overhead on average, the virtualization overhead of *WhiteRabbit* is kept to a minimum at 0.74% for CPU and 3.48% for memory benchmarks on average. According to our measurements, Xen outperforms KVM and approaches *WhiteRabbit* with an averaged 2.66% CPU and 3.65% memory overhead. While we expected the arithmetically heavy benchmarks *Gzip* and *7-Zip* to induce a similar overhead as the other CPU-intensive benchmarks, they are outliers for all candidates.

Performance measurements among VMMs can be unreliable as each VMM might emulate and scale the guest's clock source differently. As our prototype does not emulate any clock sources, we can precisely determine the resulting virtualization overhead. Therefore, we ran the *SPECspeed Integer* benchmarks of the *SPEC CPU2017* suite and summarized the results in Table 2.

Finally, we used *lmbench 3.0* micro-benchmarks, to observe the performance overhead on system software level (Table 3). The overall picture suggests that the special-purpose design of *WhiteRabbit* is ideally suited as basis for VMI tools.

9 Limitations

Malware can evade analysis through anti-virtualization techniques [19]. These consider side effects of emulated instructions, as certain instructions are not sufficiently documented [21]. This can be addressed by trying to make the VM indistinguishable from real hardware; the lack of hardware behavioral knowledge can be met through massive testing [17]. Nevertheless, timing attacks present the main issue: adversaries with access to external time sources can detect discrepancies caused by virtualization. Consequently, a system that achieves perfect VM transparency is infeasible in practice [18]. Yet, the trend toward system consolidation through virtualization renders the goal of VM transparency obsolete. If a system is virtualized, it does not necessarily mean the malware is subject to analysis. Thus, it is more affordable for attackers to target both physical and virtual environments than exclusively focusing on physical machines.

Besides, the combination of in-band and out-of-band delivery with derivative patterns establishes a solid ground for forensics analysis. Nonetheless, this combination cannot detect every modification performed by VMI-aware malware. Derivative approaches cannot reconstruct the entire state [15]. This is because data structures that have been reconstructed through delivery patterns cannot be bound to hardware. Consequently, unannounced structural modifications of these data structures (e.g., through malicious relocation in memory) may remain unnoticed. This is the result of the *strong semantic gap* [22]. As such, VMI tools cannot rely on the guest's integrity as long as every semantically relevant data structure is not bound to hardware or its trustworthiness is not validated [22].

Another limitation is that DMA-capable devices have access to the system's physical memory. Through DMA, adversaries can locate *WhiteRabbit* in memory despite SLAT. To approach this, *WhiteRabbit* could restrain DMA access by engaging the system's IOMMU (Intel VT-d or ARM System MMU).

Also, since *WhiteRabbit* is deployed on-the-fly, a VMI application may miss the point of infection. That is, one-shot exploits can be injected to gather critical information and unloaded before deploying *WhiteRabbit*. The same applies to periodical system checks by regularly loading and unloading *WhiteRabbit*: conducted attacks may slip through periodic system checks and leverage the semantic gap to delude VMI applications [22]. These restrictions render *WhiteRabbit* more suitable for detection and analysis of long-living, persistent malware.

10 Countermeasures

WhiteRabbit is a powerful tool for forensic analysis that becomes a dangerous weapon in hands of adversaries. To defeat intruders, we propose countermeasures. A proactive approach suggests to employ a native VMM, such as Xen, that executes on bare metal and leverages the system's virtualization extensions. If an attacker initializes *WhiteRabbit* from a compromised VM, the underlying VMM will intercept and discard any subversion attempt: on Intel, instructions required to set up VMX root operation implicitly trap into the VMM; on ARM,

the VMM deflects attempts to reconfigure `VBAR_EL2`. Even if the maliciously utilized *WhiteRabbit* supported nested virtualization (enabling VMM hierarchies), it would not be able to take control of the system's virtualization extensions as they would be occupied by the benign VMM. The same applies to hosted VMMs, such as KVM: subversion attempts from a compromised VM would not be able to take over control of the VMM. On the other hand, an adversary might subvert the entire system before KVM controls the system's virtualization extensions.

Assuming the underlying VMM supported nested virtualization, would it be possible to subvert the compromised guest and execute as a nested VMM inside VMX non-root? Although the native VMM would intercept every virtualization attempt, without additional precautions and VMI analysis, it would not hinder *WhiteRabbit* from subverting the guest (much like it would not hinder valid second level virtualization), but rather forward all guest VM exits to the nested, vicious VMM. This issue is an open question for our future work.

If the attacker succeeded to inject and execute *WhiteRabbit* in VMX root, she could subvert the running VMM. Yet, a transparent execution of the VMs would only be possible if the intruders managed to reveal and set up shadowed copies of all VMCS data structures, as these represent the used virtual CPUs. Further management of the system setup would require the support of nested virtualization. Thus, a VMM does not prevent but rather hardens a subversion.

11 Related Work

PI [4] is an in-band delivery framework for injecting security applications into a guest VM. Vogl et al. [5] extend this idea with *X-TIER*, a framework for malware detection and removal. In contrast to PI, which hijacks user space processes, X-TIER injects kernel modules into the guest. DRAKVUF [6] is a VMI-based, dynamic malware analysis system using LibVMI and thus out-of-band delivery.

Nitro [16] introduces a VMI framework that uses its hardware architecture knowledge to derive semantic information about the guest OS. *Ether* [3], on the other hand, manipulates the hardware managed fast system call dispatcher location and redirects guest system calls to a fixed, unpaged memory location resulting in page-faults that are intercepted by the VMM. Another derivative view generation approach is taken by Litty et al. [7]. They present *Patagonix*, which is a hash-based memory validation framework on top of Xen. It employs binding semantic knowledge related to the MMU and the paging mechanism for malware detection. Similarly, Kittel et al. [8] present a Linux kernel validation approach considering run-time code patching performed by the kernel.

SubVirt [23] introduces one of the first Virtual Machine based rootkits (VMBRs) that can be permanently installed as a VMM underneath existing Linux and Windows OSes. In the meantime, VM-based rootkits are evolved to hardware-assisted VM (HVM) rootkits. Rutkowska introduces *Blue Pill* [24], an HVM rootkit being able to transparently move an executing OS instance into a virtual environment controlled by a thin VMM. In parallel to Blue Pill, *Vitriol* [25] present a mostly similar HVM rootkit to subvert Mac OS X on Intel. Later,

the *New Blue Pill* [26] was presented also supporting Intel VT-x technology. In addition, Cloaker [27] and CacheKit [28] present hypervisor-assisted rootkits for the ARM architecture. Further, Buhren et al. [29] demonstrate attack vectors on ARM that allow to subvert a running Linux on-the-fly.

Similar to *WhiteRabbit*, HyperSleuth [30] is a small VMM that virtualizes a running Windows XP on-the-fly on Intel. However, HyperSleuth does not utilize the hardware-assisted SLAT mechanism and thus entails higher software overhead. It also does not hide its in-guest artifacts. This exposes its presence to in-guest malware and thus is not suited for analysis of split-personality malware.

12 Conclusion

In this paper, we presented *WhiteRabbit*, a microkernel-based architecture that unifies VMI with on-the-fly virtualization. *WhiteRabbit* comprises a thin and self-sufficient native VMM that can be deployed on-the-fly on Intel and ARM architectures. By incorporating the system's Second Level Address Translation, *WhiteRabbit* is able to hide its presence in memory, expose a LibVMI-compatible interface to enable the use of remote forensics applications, and allow to inject custom security agents into the guest's address space. We validated our kernel module based prototype on Linux running on-top of Intel x86-64. Our results demonstrate that the dynamic virtualization of a running OS is fast and further system virtualization does not present a significant performance overhead.

References

1. P. M. Chen, B. D. Noble, When Virtual Is Better Than Real, in: USENIX Workshop on Hot Topics in Operating Systems (HotOS), 2001.
2. T. Garfinkel, M. Rosenblum, A Virtual Machine Introspection Based Architecture for Intrusion Detection, in: ISOC Network and Distributed System Security Symposium (NDSS), 2003.
3. A. Dinaburg, P. Royal, M. Sharif, W. Lee, Ether: Malware Analysis via Hardware Virtualization Extensions, in: ACM Conference on Computer and Communications Security (CCS), 2008.
4. Z. Gu, Z. Deng, D. Xu, X. Jiang, Process Implanting: A New Active Introspection Framework for Virtualization, in: Annual Information Security Symposium, 2012.
5. S. Vogl, F. Kilic, C. Schneider, C. Eckert, X-Tier: Kernel module injection, in: International Conference on Network and System Security (NSS), 2013.
6. T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, A. Kiayias, Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System, in: Annual Computer Security Applications Conference (ACSAC), 2014.
7. L. Litty, H. A. Lagar-Cavilla, D. Lie, Hypervisor Support for Identifying Covertly Executing Binaries, in: USENIX Security Symposium, 2008.
8. T. Kittel, S. Vogl, T. K. Lengyel, J. Pfoh, C. Eckert, Code Validation for Modern OS Kernels, in: Workshop on Malware Memory Forensics (MMF), 2014.
9. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre Attacks: Exploiting Speculative Execution, arXiv preprint arXiv:1801.01203.

10. M. Lipp, M. Schwarz, T. Gruss, Daniel Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown, arXiv preprint arXiv:1801.01207.
11. M. Shockley, C. Maixner, R. Johnson, M. DeRidder, W. M. Petullo, Using VisorFlow to Control Information Flow without Modifying the Operating System Kernel or its Userspace, in: International Workshop on Managing Insider Security Threats, 2017.
12. Z. Deng, X. Zhang, D. Xu, SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization, in: Annual Computer Security Applications Conference (ACSAC), 2013.
13. G. J. Popek, R. P. Goldberg, Formal Requirements for Virtualizable Third Generation Architectures, Communications of the ACM 17 (7) (1974) 412–421.
14. C. Dall, J. Nieh, KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor, in: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.
15. J. Pfoh, C. Schneider, C. Eckert, A Formal Model for Virtual Machine Introspection, in: Workshop on Virtual Machine Security (VMSec), 2009.
16. J. Pfoh, C. Schneider, C. Eckert, Nitro: Hardware-Based System Call Tracing for Virtual Machines, in: Advances in Information and Computer Security, 2011.
17. H. Shi, A. Alwabel, J. Mirkovic, Cardinal Pill Testing of System Virtual Machines, in: USENIX Security Symposium, 2014.
18. T. Garfinkel, K. Adams, A. Warfield, J. Franklin, Compatibility Is Not Transparency: VMM Detection Myths and Realities, in: USENIX Workshop on Hot Topics in Operating Systems (HotOS), 2007.
19. X. Chen, J. Andersen, Z. M. Mao, M. Bailey, J. Nazario, Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware, in: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2008.
20. N. Miramirkhani, M. P. Appini, N. Nikiforakis, M. Polychronakis, Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts, in: IEEE Symposium on Security and Privacy (S&P), 2017.
21. C. Domas, Breaking the x86 ISA, Black Hat, USA.
22. B. Jain, M. B. Baig, D. Zhang, D. E. Porter, R. Sion, SoK: Introspections on Trust and the Semantic Gap, in: IEEE Symposium on Security and Privacy (S&P), 2014.
23. S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, J. R. Lorch, SubVirt: Implementing Malware With Virtual Machines, in: IEEE Symposium on Security and Privacy (S&P), 2006.
24. J. Rutkowska, Subverting VistaTM Kernel for Fun and Profit, Black Hat, USA.
25. D. A. Dai Zovi, Hardware Virtualization Rootkits, Black Hat, USA.
26. J. Rutkowska, A. Tereshkin, IsGameOver () Anyone, Black Hat, USA.
27. F. M. David, E. M. Chan, J. C. Carlyle, R. H. Campbell, Cloaker: Hardware Supported Rootkit Concealment, in: IEEE Symposium on Security and Privacy (S&P), 2008.
28. N. Zhang, H. Sun, K. Sun, W. Lou, Y. T. Hou, CacheKit: Evading Memory Introspection using Cache Incoherence, in: IEEE Symposium on Security and Privacy (S&P), 2016.
29. R. Buhren, J. Vetter, J. Nordholz, The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture, in: Information and Communications Security, 2016.
30. L. Martignoni, A. Fattori, R. Paleari, L. Cavallaro, Live and Trustworthy Forensic Analysis of Commodity Production Systems, in: International Symposium on Research in Attacks, Intrusions and Defenses (RAID), 2010.