Noname manuscript No.
(will be inserted by the editor)

Leveraging Branch Traces to Understand Kernel Internals from Within

Marcus Botacin · Paulo Lício de Geus · André Grégio

Received: date / Accepted: date

Abstract Kernel monitoring is often a hard task, requiring external debuggers and/or modules to be successfully performed. These requirements make analysis procedures more complicated because multiple machines, although virtualized ones, are required. This requirements also make analysis procedures more expensive. In this paper, we present the Lightweight Kernel Tracer (LKT), an alternative solution for tracing kernel from within by leveraging branch monitors for data collection and an address-based introspection procedure for context reconstruction. We evaluated LKT by tracing distinct machines powered by x64 Windows kernels and show that LKT may be used for understanding kernel's internals (e.g., graphics and USB subsystems) and for system profiling. We also show how to use LKT to trace other tracing and monitoring mechanisms running in kernel, such as Antiviruses and Sandboxes.

1 Introduction

Kernel tracing is an essential task in many scenarios, such as in security, but it may become costly because it requires multiple machines and/or an external debugger to be performed with all integrity and consistency guarantees (e.g., atomic memory dumps, pausable tasks, and exception chain inspection). However, tasks like profiling do not require all guarantees provided by external debuggers to achieve their goals. Therefore, the implementation of these analysis procedures would benefit from a lightweight kernel tracing approach. A lightweight approach could allow uses ranging from analyses in resource-constrained scenarios as well as in online, in-place kernel evaluations, thus being a significant improvement regarding current approaches.

Marcus Botacin & André Grégio - Federal University of Paraná E-mail: mfbotacin,gregio@inf.ufpr.br · Paulo Lício de Geus - University of Campinas E-mail: paulo@lasca.ic.unicamp.br

We believe that branch monitors can be used as an alternative for kernel tracing since they are processor features (thus not imposing huge overheads) and can be accessed from within the inspection environments (through Model Specific Registers—MSRs), thus not requiring external components. Therefore, we here propose the Lightweight Kernel Tracer (LKT), a branch-based framework for kernel tracing.

Current solutions for kernel tracing often rely on full, complex analysis environments, such as developing instrumented virtual machines able to trace kernel instructions, as in the case of Drakvuf [17] and HyperDbg [9]. Branch monitors were used as additional source of data on such VM-based systems, as in CXPInspector [48], but no solution relies entirely on branch-collected data. In this sense, branch data was already used for kernel crash analysis, using the processor tracer feature [51], but no tracing solution was developed. This way, as far as we know, LKT is the first solution for kernel tracing **entirely based** on branch monitor data.

As LKT does not rely on additional sources of kernel data, it allows tracing kernel from within the monitored system, a significant improvement over VM-based solutions. By introspecting into the collected addresses, context reconstruction is made possible, thus allowing us to perform many trace-related tasks, such as profiling and monitoring.

We leveraged LKT's capabilities to perform reverse engineering procedures in the x64 Windows kernel and its subsystems, thus showing how a lightweight kernel tracer can be used to enrich the knowledge about system's working in a general manner. More specifically, LKT allowed us to dig into and understand the call chains performed when multiple graphic cards were used, when a USB device was hot-plugged, and when the native firewall was filtering packets. In addition, we show that LKT is able to inspect all system modules, including the ones responsible for handling its own interrupt routines (self-tracing).

Moreover, our evaluation shows that LKT allows profiling systems in many ways, such as checking kernel code core migration—by instantiating LKT on distinct cores—and profiling the number of executed drivers and modules—by checking instruction-to-module-image mappings. It is also possible to detect and monitor other kernel monitoring solutions, such as callback-based drivers, since the callback addresses are known. We illustrate this LKT capability by reversing some antivirus (AVs) drivers to identify to which subsystems they attach their monitoring modules (e.g., USB, filesystem, or network monitoring drivers).

In summary, our main contribution are the following:

1. We propose a branch monitor-based solution for kernel tracing from within, with no need for external components, and present its development, including implementation decisions and the developed introspection procedure.
2. We evaluate LKT to show how it can be used for distinct tracing tasks, such as reverse engineer and profiling.
3. We dig into many Windows subsystems to provide a deeper understanding of how they work and how analysts can use LKT for kernel reverse engineering.

This paper is organized as follows: in Section 2, we present background information regarding kernel organization and monitoring, thus establishing the basis for this work's development; in Section 3, we present related work, which allows us to better position our developed solution; in Section 4, we introduce LKT and its present its development; in Section 5, we present LKT's evaluation and demonstrate how it can be used for many tracing-related tasks; finally, we draw our conclusions in Section 7.

2 Background

In this section, we present background concepts regarding the Windows kernel organization and its debugging procedures.

2.1 Windows Organization: Userland and Kernel

Most modern computer systems are organized into two distinct rings: userland and kernel. These two modes present distinct memory addressing—through segments—and memory protection, e.g. SMEP [20], to prevent privilege escalation attacks.

Despite these differences, the two modes work in a very similar way, having binary executables loaded and mapped into addresses within their protected address ranges.

To protect against fixed-offset payloads, modern systems randomize binary placement through the Address Space Layout Randomization (ASLR) mechanism. As the kernel and userland modes present distinct addressing spaces, the ASLR mechanisms are also distinct in each of them. However, module base addresses may be enumerated in both cases, which allows for the development of introspection procedures.

Modules loaded in memory can be device drivers or even the kernel itself, because the Windows kernel is compiled on the `ntoskernel` binary. As a regular PE file, kernel exports can be dumped, thus also helping on introspection procedures.

2.2 Kernel Debugging

Kernel analysis procedures are mostly performed by using debuggers, because they are able to step and stop execution at distinct inspection points. In this sense, Windbg [49] is probably the most used tool for such purpose.

Stopping and changing values of a running kernel is a delicate task because consistency must be kept, thus requiring inspection solutions to have a whole-system view to not generate deadlocks or setting the system in an unrecoverable state. Therefore, this task is mainly performed from outside by externally debugging a kernel running within a VM/emulator [41].

The usual way of performing kernel inspection is by using a serial port [11], but other connection facilities may be leveraged, such as USB [34] or via NIC adapters [35]. These approaches present a significant drawback regarding infrastructure since two machines are required: the external monitor and the monitored machine, though modern systems may be virtualized to perform both tasks in the same physical machine. Although the use of VMs makes analysis easier, the task still cannot be performed from within. Also, enabling debug mode may require a reboot and the use of specific boot parameters [25], which also makes the task more time-consuming.

As an alternative to external debuggers, internal, local debuggers have been deployed [30]. Despite not providing all the same resources that an external one does due to consistency issues—breakpoints are disabled, for instance, because the kernel cannot be stopped—, these solutions are very useful, since many tasks, such as tracing, do not require all the features provided by an external debugger. As an advantage, in-place analysis can be performed in a shorter time and with no additional hardware resources.

In-place kernel analysis gave its first steps when allowing crash dumps to be analyzed using the same machine [23]. Currently, local debugging solutions can perform online routines traces and memory/register values can be read on-demand, which indicates a trend towards the development of more powerful local kernel debuggers and tracers. In this context, we propose an alternative for local kernel tracing based on the branch monitor existing in modern Intel processors.

2.3 Branch Monitoring

Modern processors present a series of ways of collecting execution metadata. Most of these ways are based on hardware sensors and counters, which can be programmed to monitor given events and then read for data collection. Intel's processors, for instance, present two kinds of monitors [13]; the Precise Event Based Sampling (PEBS), an event-driven counter; and the branch monitors, branch address-based data collectors.

Intel's branch monitors are presented in two forms: Last Branch Record (LBR), which store data on a circular, register-based buffer; and the Branch Trace Store (BTS), which store data on O.S. memory pages. While LBR requires polling for data collection, BTS presents an interrupt-based mechanism to notify the buffer is full. Filters are available for both LBR and BTS, so one may choose to monitor only userland or kernel branches and/or even specific branch types (e.g., `CALLs` or `RETs`). Moreover, when BTS is enabled, one can filter out branches associated to the interruption handler and collect only the monitored branches that originated the interruption. According to Intel's manual [13]: "freezing LBRs on PMI (bit 11) allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI".

The initial application for branch monitors was profiling, since they allow the identification of hot code regions [38] and performance bottlenecks. Over time, these were turned into security features, being used for the detection of ROP attacks [39,7].

However, most branch monitor-based applications are userland-focused, thus not covering the kernel. In this paper, we propose filling such gap by leveraging BTS capabilities for kernel monitoring, which allows us to trace kernel from within.

3 Related Work

Kernel tracing may be performed for many purposes, from malware analysis and detection [19] to profiling [42]. In all cases, many event features can be employed [47]. As a general drawback, however, data collection is often performed using solutions that impose significant performance penalties. As an example, entire virtual machines were proposed to trace kernel, both on the x86 [15,14,17], as well as on the ARM platform [12]. Despite effective, many analysis tasks require only a subset of those solutions' monitoring capabilities, thus they could be replaced by lightweight ones with significant performance gains.

The HyperDBG framework [9], for instance, leverages hardware VM extensions to create a complete debugger which is able to trace the running kernel. The solution is able to perform all kinds of analysis, including online value changes. However, if we are interested only in function tracing, the solution is too complex, because an entire hypervisor needs to be deployed. Similarly, CXPInspector [48] is a hardware-assisted VM able to perform system profile. Such task, however, could be performed using a lightweight solution, since runtime value changes are not required.

In this scenario, we looked for alternative solutions able to help us to trace kernels. A noticeable class of hardware feature that could be leveraged for such purpose are processor event monitors, such as branch monitors. Previous work have demonstrated the processor trace feature could be used for kernel crash analysis [51], but, as far as we know, no online kernel tracing solution was developed based on such feature. The work of Botacin et al. [4] presented a framework for binary analysis using the branch monitor, which can be considered the closest work to ours. Their framework. however, does not cover kernel data capture or multi-core support, extensions presented by us in this work.

4 LKT: A Lightweight Kernel Tracer

In this work, we propose the Lightweight Kernel Tracer (LKT). It uses the BTS branch monitor as a way of tracing the kernel from within. LKT relies on BTS interrupts to enrich the collected data, thus allowing flow reconstruction. LKT targets x64 Windows Operating

Systems since the analysis of a closed-source, modern kernel is a suitable task for LKT's evaluation.

Previously, a BTS-based framework—named Branch Monitor (BM)—was proposed for the userland scenario [4]. Our work can be seen as an extension of the BM framework to handle kernel data. In most part of this work, we describe the differences from BM to LKT. Implementation details on how to read the BTS monitor are omitted in this paper but can be found online [3].

Similarly to the BM, LKT is also structured in a client-server architecture: the kernel driver is the server-analogous and the userland application is the client-analogous. Raw data is collected in the kernel and enriched in userland. As the major distinction between the LKT and the BM framework, BTS flags were set to capture only kernel data and to skip userland branches. As a consequence, the general overhead is reduced, because kernel branches are much less frequent than userland ones.

Figure 1 presents an overview of the LKT architecture. Userland branches (in red) are ignored. The green arrows indicates the captured flows. When a module running inside the kernel takes a branch, the BTS processor feature stores it in an OS page. After reaching a given storage threshold, an interrupt is raised and is handled by the LKT driver, also inside the kernel. The captured branches are stored in a queue and further dispatched to userland. The userland collector will enrich such data based on an introspection procedure (further described).

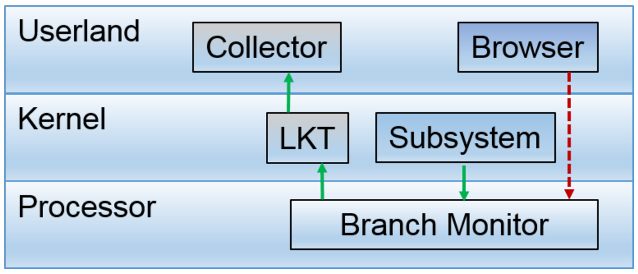

Fig. 1: **LKT Framework**. Only kernel branches are captured by the branch monitor. Branch Monitor interrupts are handled by the `LKT` module in the kernel. I/O calls provide userland with kernel taken branches.

As we are only aiming to trace the kernel, without modifying it, LKT is allowed to perform delayed data collection—storing data in OS pages when the interrupt happens and only copying the data to userland when possible—, thus not overloading the I/O mechanism. To do so, LKT implements a multi page-based data collector, which works as a straightforward extension of existing double buffer approaches [40,44].

Figure 2 shows a temporal view of data handling procedures by LKT. When enabled, the BTS feature starts to store taken branches in an OS page. The LKT driver does not have data to handle until an interrupt has occurred; consequently, no data is passed to the userland collector. When the storage threshold is reached and an interrupt is raised for the first time, the LKT module changes the page BTS that is storing data to the next one. At the same time, LKT starts copying data from the previously used page to its internal queue. As the first page was not completely handled by LKT at this time, the collector still does not have data to handle yet. When a new interrupt is raised, LKT proceeds regularly, setting BTS to store on a new page and collecting data from the previously written one. As now at least one page was entirely copied, the userland collector may start collecting data from the LKT driver. In the next interrupt, BTS is supplied with the first page, as it was handled both by LKT and the userland collector. This procedure is repeated until monitoring is finished.

As the branch monitor is a *per-core* feature and the kernel—and its subsystems—can be scheduled in distinct cores at each context switch, LKT has to be enabled in all cores. When an interrupt is raised, the core is identified through the `GetCurrentProcessorNumber` routine [27], thus allowing associating branches to the core in which they were executed.

LKT sets its interrupt routine via the `_HalpSetSystemInformation()` API, whereas the original BM framework works by redirecting interrupts to a Non-Maskable Interrupt (NMI) handler. Although both approaches are native and do not require patching kernel structures, we considered the first one as more advantageous when considering a multi-core implementation: LKT is aimed to operate on multiple cores and BM's NMI interrupts must not overlap, which limits the original BM's threat model to a single core operation whereas LKT can natively operate in a multi-core fashion because it has independent interrupt routines for each processor core.

To keep the delayed data collection working in the multi-core scenario, LKT extends the multi-page approach to work in a circular way via a centralized memory pages repository for all processor cores. This repository is managed by mutexes and in the distinct times in which each core raises an interrupt routine, the LKT interrupt handler re-enables BTS to store data in the next available page. The stored page is labeled with the core number so as to allow one to associate branches with the core in which they were executed. The LKT

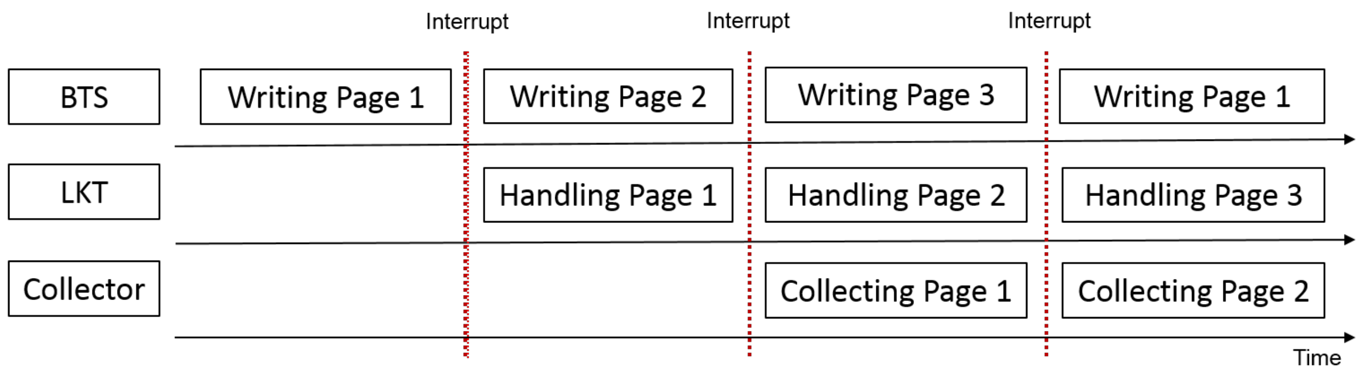

Fig. 2: **LKT Multiple Buffers**. When BTS is writing in a given memory page, LKT is handling a previous written page. Similarly, the Collector is retrieving a page previously handled by LKT.

collector works by continuously polling the LKT driver for more data until a termination signal is sent. Upon a termination signal, LKT flushes all buffered data so as to avoid losing branch information related to termination events.

4.1 Introspection

LKT may implement distinct data handling policies during its operation. Each one of them is more suitable for a distinct kernel monitoring task (as presented in Section 5). To implement such policies, the branch data must be enriched so as to provide human-readable information, which is performed through an introspection procedure, as following described.

The branch data collected by BTS, as shown in Code 1, is composed only by raw data, i.e. instruction addresses with no meaning for human readers. To extract high-level semantic information from branch data, it has to be enriched by an introspection procedure. An introspection procedure for userland branches was proposed in the original BM framework. We present here an extension of such proposal to cover kernel space branches. Whereas introspecting into kernel space branches, the introspection procedure is implemented in userspace, as for the original BM framework.

Code 1: **BTS-collected data**. BTS provides the source and target addresses of taken branches, with no high-level information.

```
SOURCE|TARGET
0xe717814|0xe6b128
0xe717a20|0xeaf3fb
```

The first data enrichment step implemented by LKT consists of identifying which module a given branch refers. To do so, LKT's client enumerates all loaded modules and retrieve their base address using the `DriverView` [36] tool. We opted for relying on this third-party tool so as to keep the LKT operation compatible with the original BM solution, which relies on a third-party tool at userland for its introspection routine. However, LKT could operate with any other information retrieval tool.

Despite being called driver, the tool is able to enumerate all kinds of loaded modules, which include the kernel image (`ntoskrnl`) and all its subsystems (e.g. `acpi`), as shown in Code 2. This enumeration step should be repeated at each system run due to ASLR effects over the loaded modules.

Code 2: **Module Introspection**. Kernel Image enumeration is used to associate branch addresses to modules.

```
<driver_name>ACPI.sys</driver_name>
<address>0116A000</address>
<end_address>011D7000</end_address>
----
<driver_name>ntoskrnl.exe</
    driver_name>
<address>97A83000</address>
<end_address>981CB000</end_address>
```

Once the image module was identified, LKT needs to identify the called routine within the given module, which is done by looking to the offset between the target of a branch and the module base addresses ($branch\_target = module\_base + offset$). LKT can match the identified branch offsets to a list of routine exports to identify routine names.

To retrieve the exported routine offsets of a given module, LKT uses the `DLLView` [37] tool. Despite being called DLL, the tool is able to retrieve exports from any Windows executable file, including kernel image (`ntoskrnl`) and its subsystems, as shown in Code 3. Unlike the previous case, there is no ASLR effect on routine offsets, so these can be enumerated once and stored in a database.

Code 3: **routine Introspection**. Binary Exports are matched against branch addresses to identify called routines within modules.

```
C:\Windows\System32\drivers\ACPI.
    sys
 > DeRegisterOpRegionHandler 0
    x6b6f0
 > RegisterOpRegionHandler   0
    x6b684
----
C:\Windows\system32\ntoskrnl.exe
 > CmRegisterCallback        0
    x014050b150
 > CmSetCallbackObjectContext 0
    x014052cd2c
 > CmUnRegisterCallback      0
    x014050a9a4
```

A summary of the whole introspection procedure is presented in Figure 3. For a given branch (`0xBA3F`), LKT first identifies the value `0xBA00` as the base module address of `ntoskrnl`. The table holds a pointer to a list of routines exported by this model. The offset `0x3F` is used to index this list, allowing LKT to identify the called routine as being the `IoFreeIrp`.

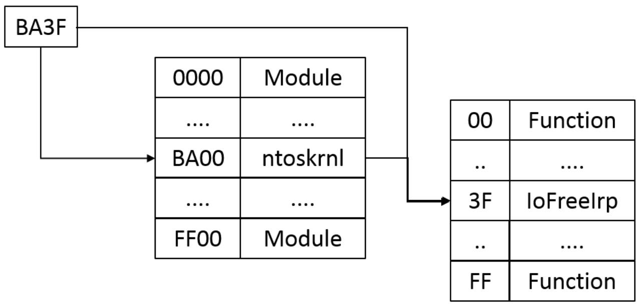

Fig. 3: **LKT Introspection Procedure**. Branch Addresses are mapped to loaded modules to bridge the semantic gap. Branch base addresses are used to identify modules base addresses. The respective offsets are used to map the exported routines.

By repeating this procedure for each taken branch, the whole kernel execution path can be reconstructed. We highlight that this introspection procedure is able to retrieve names and entry points only for the routines exported by the modules, and not for module's internal routine calls. In the traces, the latter are reported as belonging to an `Unknown` location.

5 Experiments & Results

In this section, we present many applications enabled by LKT to showcase the broadness of its impact.

5.1 Understanding Windows Kernel Internals

A straightforward application for the developed monitoring solution is to monitor the kernel itself. Tracing the kernel allows for a better understanding of its working mechanisms and internals. We follow by demonstrating such possibilities.

5.1.1 The Video/Graphics Subsystem

Modern notebooks often present multiple graphics cards: an internal one, used for lighter tasks, and an external card, used for the heavier ones. It allows for battery saving while performing ordinary user activities and good performance under high-load situations. Code 4 presents an excerpt of a kernel trace obtained from a notebook having an integrated Intel graphics card and an external ATI one. It allows us to understand how graphics card switching is performed.

Code 4: **Video Subsystem**. During card switches, requests are routed from the Intel driver to ATI one.

```
0x20078 <intelppm.sys+Unknown>
0x60000 <atikmdag.sys+Unknown>
0x20043 <atikmdag.sys+Unknown>
0x77365 <vdrvroot.sys+Unknown>
```

We observe that the kernel first asks the Intel driver (`intelppm`) to provide a given resource or to take a given action. The execution proceeds to the ATI driver (`atikmdag`), which indicates that the Intel driver skipped from answering the request, thus forcing the kernel to reroute it to another driver. We can infer that the kernel operates on a call chain model, asking the next driver in the call chain when the first does not handle a given request. Finally, after the request is handled, the execution returns to the video subsystem (`vdrvroot`), which keeps performing requests.

5.1.2 Interrupt Handling

Given the system-view capability of BTS, LKT is able to inspect all control flow changes, which includes interruption handling. Code 5 shows an excerpt of an Interrupt Service Routine (ISR) of a device driver. In this case, branch target addresses are collected even during the interrupt handling and further interpreted as being part of the ISR routine.

Code 5: **Interrupt Handling**. Hardware events are handled by the HAL subsystem.

```
43fb <Monitor.sys+Unknown>
b4da <hal.dll+Unknown>
6703 <ntoskrnl.exe+Unknown>
adf0 <hal.dll+
     HalPerformEndOfInterrupt
     >
```

We notice that the cleanup operation of the device driver (`Monitor.sys`) returns to the Hardware Abstraction Layer (HAL) subsystem, implemented by the `hal.sys` module. The HAL subsystem abstracts hardware differences and provides a uniform interface for driver programming, making development easier. This trace excerpt shows that the HAL subsystem is responsible for notifying the kernel (`ntoskrnl.exe`) about the ISR finish event and performing all ISR cleanup actions (`HalPerformEndOfInterrupt`).

5.1.3 Encrypted Volumes

Modern filesystems provide built-in encryption capabilities. Our monitoring solution allows us to infer some details about such implementations. Code 6 presents a trace excerpt related to the filesystem subsystem.

Code 6: **Encrypted Volumes**. The New Crypto API is used to perform crypto-related tasks in the filesystem.

```
0063 <volmgrx.sys+Unknown>
0043 <cng.sys+BCryptUnregisterProvider>
```

We notice that the volume manager directly calls the crypto subsystem when a volume is unmounted since the running context must be cleaned. We also observe the encryption is performed using the Cryptography Next Generation (CNG) [24] subsystem, implemented by the `cng.sys` module.

5.1.4 Network Subsystem

A constant system monitoring may also allow us to better understand how the network subsystem works. Code 7 presents a trace excerpt related to network activities.

Code 7: **Network Subsystem**. Network requests are filtered by the firewall, which calls specific modules to parse the payload (HTTP, in this case).

```
0x004e <ndis.sys+NdisRegisterProtocol>
0x0dc8 <wfplwfs.sys+Unknown>
0x04f0 <HTTP.sys+Unknown>
```

We notice that the execution flow in the network subsystem starts in the Network Driver Interface Specification (NDIS) driver, which is a driver wrapper for network protocols [32]. In particular, `NdisRegisterProtocol` [31] is responsible for registering requests to be handled at the transport level. The execution flow allows us to identify the Windows Firewall (`wfplwfs.sys`) as the requester. We can also identify the carried protocol as being HTTP, given the call to the `HTTP.sys` driver. From this point on, the NDIS subsystem will intercept network traffic and redirect it to be filtered by the native system firewall.

Understanding the workings of this mechanism is important from both the attackers' and defenders' perspectives: as the kernel has an embedded protocol parser, a bug in it may lead to remote exploitation through the incoming packets.

5.1.5 External Device Handling

A frequent task on modern computer systems is to plug and unplug external devices. Code 8 shows a trace excerpt from the execution flow during a USB device connection.

Code 8: **Handling External Devices**. The Plug aNd Play (PNP) subsystem in action during the connection of an external USB device.

```
00ba <CLASSPNP.SYS+Unknown>
48a0 <usbehci.sys+Unknown>
6260 <UsbHub3.sys+Unknown>
9e50 <bthport.sys+DllInitialize>
```

The connection of new devices is handled by the Plug aNd Play (PNP) subsystem (`classpnp.sys`). When the device is plugged in, the subsystem asks for specific information, retrieved by the USB drivers (`usbehci.sys` and `usbhub3`); After identifying the specific device—a Bluetooth adapter, in this case—the Bluetooth profile subsystem [21] (`bthport`) is called.

5.2 Understanding Drivers and Modules

Besides understanding the behavior of whole kernel subsystems, our solution can also be leveraged to understand individual modules. The trace excerpts shown in Code 9 and Code 10, respectively, allow us to observe that the module `win32k.sys` performs memory allocation (`ExAllocatePoolWithTag`) and operates on a per-thread basis (`PsGetCurrentThreadWin32Thread`), thus requiring thread identification to succeed.

Code 9: **Module Tracing**. We were able to identify the `win32k` module performing memory allocation during its execution.

```
0x3660 <win32k.sys+Unknown>
0x4000 <ntoskrnl.exe+
    ExAllocatePoolWithTag>
```

Code 10: **Module Tracing**. The `win32k` module requires knowledge of the calling thread to run properly.

```
0x7e9f <win32k.sys+Unknown>
0xb120 <ntoskrnl.exe+
    PsGetCurrentThreadWin32Thread>
```

Code 11: **Module Tracing**. The traced module (`Monitor`) makes use of Kernel lists during its execution.

```
0x5471 <Monitor.sys+Unknown>
0xc05c <ntoskrnl.exe+ExAllocatePool
    >
0x4000 <ntoskrnl.exe+
    ExAllocatePoolWithTag>
0x5483 <Monitor.sys+Unknown>
0x7d10 <ntoskrnl.exe+
    ExpInterlockedPopEntrySList>
0x3280 <ntoskrnl.exe+
    KeReleaseInStackQueuedSpinLock>
```

By observing a larger trace excerpt, such as the one presented in Code 11, we can infer even more complex behaviors. For instance, we discovered that the `Monitor.sys` module allocates memory (`ExAllocatePool`) and then pops a value from the list (`PopEntrySList`), which suggests the popped value is copied to the allocated memory space, which acts as a buffer. As the kernel (`ntoskrnl.exe`) handles lists in an atomic way (`Interlocked`), it has to release the lock (`ReleaseSpinLock`) after accessing the list.

5.2.1 Multi-Core Modules

As our solution is multi-core–based, we can trace the cores independently and identify the code portions running on each one. Our first finding is that many routines from the `ntoskrnl` can be called independently on each core, as shown in Code 12.

Code 12: **Multi-core Monitoring**. We notice routines being independently invoked on distinct cores.

```
Time|Core|Addr|Module
5.42|0|0x0a40|<peauth.sys+Unknown>
5.42|0|0x1e20|<ntoskrnl.exe+
    ExFreePoolWithTag>
5.44|1|0x9000|<win32k.sys+Unknown>
5.44|1|0x1e20|<ntoskrnl.exe+
    ExFreePoolWithTag>
```

We observe that at almost the same timestamp (5.4), two distinct modules (`peauth.sys` and `win32k.sys`) branch from their internal routines (identified by the `Unknown location`) to the entry point of the same routine (`ExFreePoolWithTag`) from the kernel image (`ntoskrnl`). As no thread or process information is retrieved and they were running concurrently on distinct cores (0 and 1), we infer that the routine is implemented in a way each execution context is independent from the other one.

In addition, LKT may also be used to identify multi-threaded modules. We identified two multi core-related cases: (i) core migration; and (ii) threaded code execution. A core migration happens when the OS schedules a thread on a distinct core than the one in which it has been scheduled before. Although it might be triggered by many reasons, it often happens due to performance reasons. When the OS identifies that a given core is overloaded with processing tasks, it might schedule the thread in a less used core to speed up processing. In the threaded code execution case, a given code is not migrated from one core to another. Instead, two or more instances of the same code are executed in distinct cores in parallel to speed up processing.

When a core-migration event is monitored by LKT, one can observe that the same branches of the same thread are taken on two distinct cores in subsequent moments. When a threaded code execution is monitored by LKT, one can observe that the same branches are taken on distinct cores at nearly the same time, as exemplified by Code 13.

Code 13: **Multi-core Monitoring**. The subsequent execution of the same code region on distinct cores may indicate the use of threads.

```
Time|Core|Addr|Module
3.20|0|0x5471|<Monitor.sys+Unknown>
3.20|0|0x43fb|<Monitor.sys+Unknown>
3.24|1|0x5471|<Monitor.sys+Unknown>
3.24|1|0x43fb|<Monitor.sys+Unknown>
```

We observe that at almost the same timestamp (3.2), the same addresses (`0x5471` and `0x43fb`) from the same module (`Monitor`) were in execution on distinct cores (0 and 1). We highlight that the executed instructions belong to its internal code and not to an exported routine entry point, given the `Unknown` identifier. It suggests the executed region is part of a threaded code, fact which was further manually confirmed.

LKT currently cannot differentiate between multiple threads running on the same core, because its introspection procedure does not provide context information, such as the thread identifier, for individual branches within a memory page of collected branches. Whereas thread identifier information can be retrieved by using additional introspection procedures, such as using the `GetThreadId` routine [28], one must ensure that distinct threads were not scheduled during the same branch collection window. It might be achieved by limiting the number of collected branches during a window, as performed by the original BM framework.

5.3 Monitoring the monitor

More than understanding the kernel and its modules, the system-wide view capability of BTS also enables tracing monitoring drivers themselves, as shown in Code 14. This is a significant advantage over other tracing mechanisms, such as callbacks, which allow tracing external events but not their own ones.

Code 14: **Self Monitoring**. BTS is able to trace branches from the BTS driver.

```
0x5471 <BranchMonitor.sys+Unknown>
```

Given such capability, we leveraged LKT to inspect the monitoring driver of our sandbox solution for malware analysis [6]. Code 15 shows the identified routine calls.

Code 15: **Monitoring a sandbox**. LKT allows us to identify the usage of callback routines by the sandbox driver.

```
0x4f09|<Monitor.sys+Unknown>
0xa354|<ntoskrnl.exe+
      PsSetCreateProcessNotifyRoutine>
0x4ec7|<Monitor.sys+Unknown>
0x0b40|<ntoskrnl.exe+
      CmRegisterCallbackEx>
```

We identified calls to `PsSetCreateProcessNotifyRoutine` [33], a process callback, and to `CmRegisterCallbackEx` [22], a registry callback. As described by the solution paper [6], both callbacks are used for activity monitoring.

In addition, when looking to the number of times that each code region was executed during the sandbox run, as shown in Figure 4, we identified multiple calls to the addresses `0x4a30|<Monitor.sys+Unknown` and `0x4b48|<Monitor.sys+Unknown`, which were supposedly the callback handler routines.

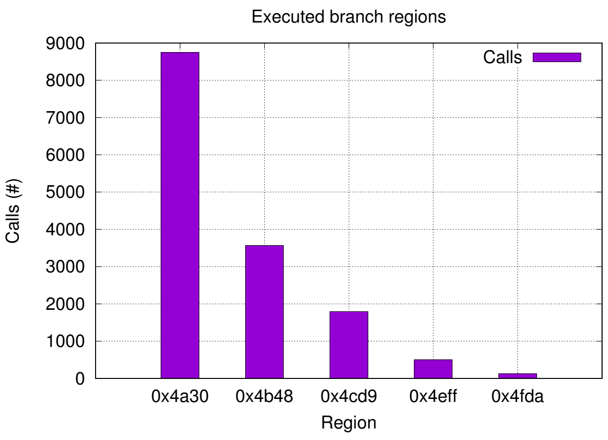

Fig. 4: **Executed Regions**. The most frequently executed regions of the sandbox drivers allow us to identify the callback routine handlers.

In practice, many modules and system components rely on callbacks for their monitoring operations, thus they can be identified by LKT. Code 16 shows that the volume management subsystem (`volmgrx.sys`) relies on a filesystem (`fltmgr.sys`) callback (setup via `FltIsCallbackDataDirty` [26]) for its operations. It may rely on the registered callback, for instance, to identify whether there is a pending operation on a removable device.

Code 16: **Callback Monitoring**. The filesystem uses callback to identify pending operations in a given device.

```
0x0047 <volmgrx.sys+Unknown>
0x0069 <fltmgr.sys+
      FltIsCallbackDataDirty>
```

Code 17 shows that the crypto subsystem (`cng.sys`) relies on a callback (registered via `EntropyRegisterCallback`) to get entropy. It might allow, for instance, a non-blocking operation while enough data is not available to generate random numbers.

Code 17: **Callback Monitoring**. The crypto subsystem uses a callback to retrieve entropy for random number generation.

```
0x341f <cng.sys+EntropyRegisterCallback>
```

Callbacks are often used in the security context to allow solutions to enforce security policies on modifying data. Therefore, we leverage LKT to identify callback usage in a particular class of security applications: Anti-Viruses (AVs).

We have installed and traced the Avast and Avira solutions, as they were the most downloaded AVs by the time we queried Softonic's security applications Top10 [46]. The identified number of callbacks and modules are presented in Table 1. More details are presented in Appendix A.

Table 1: **AV Callbacks**. Number of AV modules which invoked at least one system callback routine and the number of distinct invoked callback routines.

AV	Modules	Callbacks
Avast	12/13 (92%)	8
Avira	2/6 (33%)	1

We notice that both solutions perform their monitoring activities by using callbacks. Such discovery is important to understand how modern AVs work because the so-far used kernel hooking-based monitoring techniques [8] are now prevented by the Kernel Patch Protection (KPP) mechanism [6]. Therefore, AVs had to find new ways of implementing system monitoring mechanisms.

Table 2: **Kernel Profiling**. Number of executed modules during a run. We observe `PC I` executes more modules than `PC II`. We also observe modules are core-scheduled in an unbalanced way.

PC \ Core	I	II	III	IV
I	70	44	31	49
II	63	9	53	31

Table 3: **Kernel Profiling**. Number of times that the ACPI module was called within the execution window. We notice that the `PC I` performs more calls than the `PC II`, thus presenting a higher performance penalty.

PC \ Core	I	II	III	IV
I	215	1	1	2
II	58	2	0	6

Table 4: **Kernel Profiling**. Number of times an interrupt routine was executed. As `PC I` handles more interrupts than `PC II`, its performance is more affected.

PC \ Core	I	II	III	IV
I	100	0	98	96
II	65	0	15	98

5.4 Kernel Profiling

Application profiling is a frequent development task, as programmers are always required to extract the best results of their implementations and available hardware resources. Kernel profiling is a particular class of such approach, aimed to assure that the kernel is not wasting time which could be better employed on performing users' tasks. An example of an important kernel profiling task is the TCP/IP stack verification [50].

To give some insight on how LKT could be used for kernel profiling, we monitored the kernel while running the same version of the Chrome web browser and opening the same home page on two distinct computers—running Windows 8 x64 on a 4GB, i5 and i7 processors, respectively.

Tables 2, 3 and 4 show, respectively, statistics for the number of distinct modules which executed at least one branch during monitoring, the number of times the ACPI module was invoked, and the number of times an interrupt handling routine was executed. For these cases, we considered a sequence of consecutive branches from the same module as a single entry. The results are based on the average values for 5 executions.

Distinct kernel performances are expected in the two machines as they present distinct drivers setups. We observed that the first machine actually executed more code pieces (more drivers/modules were scheduled) than the second. Running more kernel modules implied on handling more interrupts and ACPI events. The higher number of interrupts and ACPI-handling routines observed in the machine I in comparison to the machine II suggests a higher performance impact on the first because the system spent more time running kernel code instead of actual userland tasks.

In both cases, however, we notice a poor workload balancing by the OS scheduler. Most kernel code is executed in the first system core (core I) and only few tasks are performed on the other cores. On the one hand, this result can be explained by the intrinsic complexity of scheduling on multi-core systems [45]. On the other hand, it highlights a significant opportunity improvement, as enforcing third party drivers to run on cores other than the first one may increase their performance. Whereas this enforcement policy depends on multiple aspects (e.g., cache management, data sharing, so on), a tool such as LKT might help to identify when this strategy was successful.

5.4.1 Identifying Hot Code Regions

A typical profiling task is to identify hot code regions—regions frequently executed, thus significantly affecting performance when optimized. Performing hot code region identification using LKT is straightforward since the same instruction addresses will appear multiple times in the trace, as shown in Code 18.

Code 18: **Hot Code Region Identification**. Spin Lock acquisition is responsible for the repeated execution of the same instructions.

```
0x7814 <ntoskrnl.exe+Unknown>
0x2580 <ntoskrnl.exe+
     KeAcquireInStackQueuedSpinLock>
0x7814 <ntoskrnl.exe+Unknown>
0x2580 <ntoskrnl.exe+
     KeAcquireInStackQueuedSpinLock>
0x7814 <ntoskrnl.exe+Unknown>
0x2580 <ntoskrnl.exe+
     KeAcquireInStackQueuedSpinLock>
0x7814 <ntoskrnl.exe+Unknown>
```

One can easily identify the `0x7814` offset of `ntoskrnl.exe` as a hot code region, since it is frequently executed. Such behavior, however, is expected, since this code region refers to a lock implementation, used for synchronization. More specifically, it refers to the SpinLock [29] acquisition, which is performed on a busy-waiting way, to avoid significant performance penalties inside the kernel, which would be the case of alertable locks.

5.5 LKT Performance penalty

As LKT is intended to be a lightweight alternative for kernel tracing, we evaluated its performance impact when tracing a real system. We measured the imposed overhead on userland applications while performing ordinary tasks. We considered the overhead as the relative difference between the number of CPU clock ticks spent to run the benchmark application on the same system with and without LKT enabled. The number of ticks were collected in the creation and termination of the benchmark processes. The experiments were repeated for a distinct number of iterations (routine calls). It makes the kernel to be interrupted at distinct frequencies, thus avoiding the impact of system calls on performance to be masked among other routine's invocation. All experiments were conducted in the same I7-powered system previously described. Table 5 and 6 show, respectively, the impact imposed on applications that print a random value to the screen and write a random value to a disk file. The values are presented as multiples of millions tick counts and were retrieved from an average of one hundred executions.

Table 5: **LKT Overhead**. Performance penalty while printing on the screen.

Iterations	Base	Monitored
100	3	9 (200%)
1000	32	49 (53%)
10000	300	426 (42%)

Table 6: **LKT Overhead**. Performance penalty while writing to a disk file.

Iterations	Base	Monitored
100	42	76 (80%)
1000	425	503 (18%)
10000	4268	4834 (13%)

We first observe that both the base execution time and the overhead measures are application-dependent. It was expected, as each routine is internally implemented in a distinct way. Handling files, for instance, is more expensive than printing to the screen as more routines are internally called—aiming to, respectively, open the handle, position the file pointer, write the data and close the handle. In both cases, however, the execution time grows linearly with the number of calls (iterations). In spite of that, the relative overhead is greater on smaller loads than on larger loads. This happens because the performance penalty imposed to the system calls in the conditions of high system activities is partially masked by the throughput of continuously handling the higher loads. It might happen, for instance: (i) when system calls are enqueued by the OS and processed together; and (ii) when cache-resident data is reused in subsequent calls. In turn, when a small load is traced, the whole performance penalty of kernel-userland transitions is observed in every single kernel invocation.

LKT's worst observed result was a 2x slowdown—on a unique instance—, which is reasonable for a micro-benchmark. In comparison, a recent survey on Linux tracers [10] have shown that the fastest kernel tracer solution imposes a 5x slowdown whereas the slowest one may impose up to an 8x slowdown. It shows that LKT is a suitable candidate for lightweight tracing tasks.

6 Discussion

In this section, we discuss our contributions, the current implementation drawbacks, theoretical limitations and pinpoint future research directions.

6.1 Solution advances and development opportunities

The most noticeable advance enabled by LKT is to trace kernel from within, eliminating the need for kernel extensions, external debugging modules and/or additional hardware resources. LKT does not require special configurations to be loaded nor impose significant overhead, as it is based on a processor feature.

LKT's analysis capabilities allow for a variety of applications, such as profiling and tracing. In particular, it allows for tracing other monitoring solutions, which is a desired feature when tracing shielded applications that prevent direct inspections, such as AV engines. In some cases, LKT may be used even for bug finding, as one observing the flow of driver's execution and/or userland-kernel calls might be able to understand which routine path was traversed by the flawed execution and therefore identify the root cause of a bug.

An application that could benefit from LKT approach is kernel self-tracing. LKT could allow, for instance, a kernel to profile itself and detect performance bottlenecks. On the one hand, it seems straightforward for kernels to instrument themselves via either source-code routines or runtime callbacks to enable self-monitoring capabilities. On the other hand, one should notice that third-party modules, such as drivers, are often distributed as binary blobs and do not provide many metadata collection resources, such as crash report routines. Therefore, branch data could be leveraged to collect metadata from modules originally not providing these capabilities. This type of data could be used, for instance, to whether a given one is misbehaving and the root cause of it. For instance, one could leverage LKT to trace a buggy driver execution and identify which is the root-cause branch that leads to the flawed path. Similarly, one could identify which was the last executed kernel routine before a driver crash.

6.2 Limitations

The major limitation of tracing kernel from within without suspending kernel execution, as performed by the passive collection implemented via BTS, is that stepping execution and value changes are not allowed, under the risk of corrupting system state. We believe that this drawback is acceptable because not all trace-enabled applications require such capabilities.

An alternative to providing such capabilities would be to rely on multiple kernels, so one is able to control the other. Such approach is leveraged by Barebox [16], thus eliminating the need for reboots. Such approach is also employed in other contexts, like in double terminal support [18]. However, we consider this approach as expensive as launching a VM or using an external monitor.

LKT's introspection procedure may also present limitations in some scenarios. Due to Windows Kernel Patch Protection, some symbols are not exported anymore, such as SSDT addresses [6]. If these addresses are required for some use case, an additional introspection procedure must be developed. In fact, this problem is common to distinct systems: the Linux kernel is also reported to not have a well-defined kernel interface, so introspection approaches must be developed [2]. LKT's introspection procedure is also limited to the symbols exported by the deployment versions of the considered modules. The use of debugging symbols (e.g., PDB files) would allow LKT to present more high-level information to the analysts. The implementation of this feature is currently left as future work.

6.2.1 Rootkits and kernel security

In this work, we restricted our evaluation to benign cases, as we have to trust kernel integrity for data collection and symbol introspection. Therefore, rootkit analysis is out of the scope of this work. Whereas LKT could be leveraged to trace any kernel modules, including kernel rootkits, we could not assure kernel integrity without relying on a more privileged ring [43]. When running on the same privilege level, a rootkit could, for instance, tamper with LKT routines and even disable the BTS monitor. To provide full rootkit analysis capabilities, BTS interrupts could be redirected to be delivered as SMIs, and so being handled by the SMM mode at a more privileged ring (BIOS), as proposed and discussed in more details in related work [5].

Previous work leveraged LBR to identify kernel taken branches in the presence of a rootkit [1]. Compared to LKT, using LBR is more cost-intensive, since LBR requires polling for data collection and LKT is interrupt-based. In addition, whereas previous work's introspection procedure hook kernel data structures, ours is solely based on branch addresses.

6.3 Future Work

As for future work, on the one hand, we aim to extend LKT to provide more analysis facilities without losing the lightweight characteristic. On the other hand, branch facilities could also be used to enrich external monitors. A straightforward approach would be to redirect Performance Monitoring Interrupts (PMIs) to be delivered as SMIs, so that they are handled by the SMM mode, for instance. An efficient implementation for this monitoring feature should be investigated as the use of the SMM mode also presents performance drawbacks [5].

7 Conclusion

In this paper, we introduced the use of branch monitors to trace kernels from within, thus presenting a lightweight alternative to ordinary, external kernel debuggers. We developed an address-based introspection routine that allows us to enrich branch-collected data and perform context reconstruction. Our developed framework, the Lightweight Kernel Tracer (LKT), allowed us to reverse engineer the Windows kernel to gather more knowledge about its internal working—understanding the call chains performed when multiple graphics cards were used, when a USB device was hot-plugged, and when the native firewall was filtering packets—and perform kernel profiling—identifying kernel core migration, interrupt frequency, number of executed drivers along the stack—through their executed routines. We also demonstrated how LKT can be used to detect and inspect other tracing mechanisms, such as AntiViruses, by identifying when and/or in which context callback routines are called by the kernel.

Reproducibility. We believe that LKT can be a practical alternative on many scenarios, thus we released the LKT framework as an open-source solution. The code is available at `https://github.com/marcusbotacin/BranchMonitoringProject`.

Acknowledgements This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

References

1. Akao, Y., Yamauchi, T.: Krguard: Kernel rootkits detection method by monitoring branches using hardware features. In: 2016 International Conference on Information Science and Security (ICISS), pp. 1–5 (2016). DOI 10.1109/ICISSEC.2016.7885860
2. Bissyandé, T.F., Réveillère, L., Lawall, J.L., Muller, G.: Diagnosys: automatic generation of a debugging interface to the linux kernel. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 60–69 (2012). DOI 10.1145/2351676.2351686
3. Botacin, M.: Hardware-assisted malware analysis. `https://secret.inf.ufpr.br/papers/marcus-msc.pdf` (2017)
4. Botacin, M., de Geus, P., Grégio", A.: "enhancing branch monitoring for security purposes: from control flow integrity to malware analysis and debugging". "Transactions on Privacy and Security (TOPS)" (2018)
5. Botacin, M., Geus, P.L.D., grégio, A.: Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. ACM Comput. Surv. **51**(4), 69:1–69:34 (2018). DOI 10.1145/3199673. URL `http://doi.acm.org/10.1145/3199673`
6. Botacin, M.F., de Geus, P.L., Grégio, A.R.A.: The other guys: automated analysis of marginalized malware. Journal of Computer Virology and Hacking Techniques **14**(1), 87–98 (2018). DOI 10.1007/s11416-017-0292-8. URL `https://doi.org/10.1007/s11416-017-0292-8`
7. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng, R.H., Yu, M.: Ropecker: A generic and practical approach for defending against rop attack. In: Proceedings of the NDSS Symposium (2015)
8. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 6:1–6:42 (2008). DOI 10.1145/2089125.2089126. URL `http://doi.acm.org/10.1145/2089125.2089126`
9. Fattori, A., Paleari, R., Martignoni, L., Monga, M.: Dynamic and transparent analysis of commodity production systems. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pp. 417–426. ACM, New York, NY, USA (2010). DOI 10.1145/1858996.1859085. URL `http://doi.acm.org/10.1145/1858996.1859085`
10. Gebai, M., Dagenais, M.R.: Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. ACM Comput. Surv. **51**(2), 26:1–26:33 (2018). DOI 10.1145/3158644. URL `http://doi.acm.org/10.1145/3158644`
11. Haiku: Virtualbox serial debugging on windows. `https://www.haiku-os.org/guides/virtualizing/virtualbox-windows-debugging/`
12. Horsch, J., Wessel, S.: Transparent page-based kernel and user space execution tracing from a custom minimal arm hypervisor. In: 2015 IEEE Trustcom/BigDataSE/ISPA, vol. 1, pp. 408–417 (2015). DOI 10.1109/Trustcom.2015.401
13. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual. Intel (2013)
14. Khen, E., Zaidenberg, N.J., Averbuch, A.: Using virtualization for online kernel profiling, code coverage and instrumentation. In: 2011 International Symposium on Performance Evaluation of Computer Telecommunication Systems, pp. 104–110 (2011)
15. Khen, E., Zaidenberg, N.J., Averbuch, A., Fraimovitch, E.: Lgdb 2.0: Using lguest for kernel profiling, code coverage and simulation. In: 2013 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), pp. 78–85 (2013)

16. Kirat, D., Vigna, G., Kruegel, C.: Barebox: Efficient malware analysis on bare-metal. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11, pp. 403–412. ACM, New York, NY, USA (2011). DOI 10.1145/2076732.2076790. URL http://doi.acm.org/10.1145/2076732.2076790
17. 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 (2014)
18. Li, H.: A system architecture of double kernels for trusted windows terminal. In: Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC), pp. 2562–2566 (2013). DOI 10.1109/MEC.2013.6885467
19. Li, X., Zhang, Y., Tang, Y.: Kernel malware core implementation: A survey. In: 2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, pp. 9–15 (2015). DOI 10.1109/CyberC.2015.26
20. MalwareList: Intel smep – a new hardware-based security on windows 8. https://malwarelist.net/2012/10/20/intel-smep-on-windows-8/ (2012)
21. Microsoft: Bluetooth profile drivers overview. https://msdn.microsoft.com/en-us/library/windows/hardware/ff536598
22. Microsoft: Cmregistercallbackex function. https://msdn.microsoft.com/en-us/library/windows/hardware/ff541921
23. Microsoft: Crash dump analysis. https://msdn.microsoft.com/pt-br/library/windows/desktop/ee416349(v=vs.85).aspx
24. Microsoft: Cryptography api: Next generation. https://msdn.microsoft.com/pt-br/library/windows/desktop/aa376210
25. Microsoft: Debugging windows setup and the os loader. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-windows-setup-and-the-os-loader
26. Microsoft: Fltiscallbackdatadirty function. https://msdn.microsoft.com/en-us/library/windows/hardware/ff543311
27. Microsoft: Getcurrentprocessornumber function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms683181
28. Microsoft: Getthreadid function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms683233(v=vs.85).aspx
29. Microsoft: Introduction to spin locks. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-spin-locks
30. Microsoft: Local kernel-mode debugging. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/performing-local-kernel-debugging
31. Microsoft: Ndisregisterprotocol (windows ce 5.0). https://msdn.microsoft.com/en-us/library/ms904134.aspx
32. Microsoft: Network driver interface specification. https://technet.microsoft.com/en-us/library/cc958797.aspx
33. Microsoft: Pssetcreateprocessnotifyroutine function. https://msdn.microsoft.com/en-us/library/windows/hardware/ff559951
34. Microsoft: Setting up kernel-mode debugging over a usb 3.0 cable manually. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-usb-3-0-debug-cable-connection
35. Microsoft: Setting up kdnet network kernel debugging manually. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection (2018)
36. NirSoft: "driverview". http://www.nirsoft.net/utils/driverview.html (2015)
37. NirSoft: "dll export viewer. http://www.nirsoft.net/utils/dll_export_viewer.html (2016)
38. Paleari, R.: Fast coverage analysis for binary applications. http://roberto.greyhats.it/2015/02/fast-coverage-analysis-for-binary.html (2015)
39. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), pp. 447–462. USENIX, Washington, D.C. (2013). URL https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas
40. Petit, L., Nafaa, A., Jurdak, R.: Historical data storage for large scale sensor networks. In: Proceedings of the 5th French-Speaking Conference on Mobility and Ubiquity Computing, UbiMob '09, pp. 45–52. ACM, New York, NY, USA (2009). DOI 10.1145/1739268.1739278. URL http://doi.acm.org/10.1145/1739268.1739278
41. RedHat: Debugging a kernel in qemu/libvirt. https://access.redhat.com/blogs/766093/posts/2690881 (2017)
42. Rhee, J., Zhang, H., Arora, N., Jiang, G., Yoshihira, K.: Software system performance debugging with kernel events feature guidance. In: 2014 IEEE Network Operations and Management Symposium (NOMS), pp. 1–5 (2014). DOI 10.1109/NOMS.2014.6838353
43. Rossow, C., Dietrich, C.J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., Steen, M.v.: Prudent practices for designing malware experiments: Status quo and outlook. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, pp. 65–79. IEEE Computer Society, Washington, DC, USA (2012). DOI 10.1109/SP.2012.14. URL http://dx.doi.org/10.1109/SP.2012.14
44. Saidi, S., Tendulkar, P., Lepley, T., Maler, O.: Optimizing explicit data transfers for data parallel applications on the cell architecture. ACM Trans. Archit. Code Optim. **8**(4), 37:1–37:20 (2012). DOI 10.1145/2086696.2086716. URL http://doi.acm.org/10.1145/2086696.2086716
45. Siddha, S., Pallipadi, V., Mallick, A.: Process scheduling challenges in the era of multi-core processors (2007)
46. Softonic: Security & privacy for windows. https://en.softonic.com/windows/security-privacy
47. Tate, A., Bewoor, L.: Survey on frequent pattern mining algorithm for kernel trace. In: 2017 IEEE 7th International Advance Computing Conference (IACC), pp. 793–798 (2017). DOI 10.1109/IACC.2017.0163
48. Willems, C., Hund, R., Holz, T.: Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. VirusBulletin (2013)
49. WinDbg: Windbg. http://www.windbg.org/
50. Xie, P., Wu, B., Liu, M., Harris, J., Scheiman, C.: Profiling the performance of tcp/ip on windows nt. In: Proceedings IEEE International Computer Performance and Dependability Symposium. IPDS 2000, pp. 133–137 (2000). DOI 10.1109/IPDS.2000.839471
51. Xu, J., Mu, D., Xing, X., Liu, P., Chen, P., Mao, B.: Postmortem program analysis with hardware-enhanced post-crash artifacts. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 17–32. USENIX Association, Vancouver, BC (2017). URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-jun

A Appendix

In this appendix, we present more detailed information about the identified modules and callback routines for Avast (Table 7) and Avira (Table 8) solutions.

Table 7: **Avast**. Modules and Callbacks.

Name	Description	System Callbacks	Other Callbacks
aswArPot	Anti Rootkit	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine	ExRegisterCallback ExCreateCallback
aswbidsdriver	Activity Monitor	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine	CmRegisterCallback
aswbidsh	Monitor Helper	PsSetCreateProcessNotifyRoutine	
aswblog	Logging	None	None
aswbuniv	Universal	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine	
aswHdsKe	Network	NotifyIpInterfaceChange ZwNotifyChangeKey	ExRegisterCallback ExCreateCallback
aswMonFlt	Filesystem	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine	ExRegisterCallback ExCreateCallback
aswRdr	Network Redirect	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine	ExRegisterCallback ExCreateCallback
aswSnx	Virtualization	NtNotifyChangeMultipleKeys NtNotifyChangeKey PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine	ExRegisterCallback ExCreateCallback FltSetCallbackDataDirty
aswSP	Self Protection	PsSetLoadImageNotifyRoutine, PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine	ExRegisterCallback ExCreateCallback
aswStm	Stream Filter	PsSetLoadImageNotifyRoutine, PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine	ExRegisterCallback ExCreateCallback
aswVmm	VM Monitor	PsSetLoadImageNotifyRoutine, PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine KeRegisterBugCheckReasonCallback	ExRegisterCallback ExCreateCallback

Table 8: **Avira**. Modules and Callbacks

Name	Description	Other Callbacks
avdevprot	USB	PsSetCreateProcessNotifyRoutine
avgntflt	Filesystem	PsSetCreateProcessNotifyRoutine
avipbb	General	None
avkmgr	Manager	None
avnetflt	Network	None
avusbflt	USB	None