VMIGEN: Utilizing Virtual Machine Introspection for Fuzzing Complex Closed-Source Targets

Florian Schweins[1], Moritz Schloegel[2], Moritz Bley[1], Nico Schiller[1], Thorsten Holz[3]
[1]*CISPA Helmholtz Center for Information Security, firstname.lastname@cispa.de*
[2]*Arizona State University, moritz.schloegel@asu.edu*
[3]*Max Planck Institute for Security and Privacy, thorsten.holz@mpi-sp.org*

Abstract—**Modern fuzzing is a highly successful testing method, but it still struggles with stateful software that expects complex, context-rich inputs. Instead of further tuning the fuzzing process itself, we introduce VMIGEN, a new approach that captures interactions by implicitly recording both complex inputs and the corresponding system states needed for effective testing. By using *Virtual Machine Introspection* (VMI), a technique for observing the state and behavior of a VM from the outside, we can monitor actual runtime events for a given system. This way, we can extract concrete inputs and snapshot the whole system *at relevant interactions* to preserve the full system state, thereby enabling effective fuzzing. At the same time, our approach does not require access to source code, allowing us to test closed-source software on Windows. To demonstrate VMIGEN's effectiveness, we use it to test kernel drivers, including those of anti-virus engines, and Remote Procedure Call (RPC) interfaces. Our comprehensive evaluation shows that our VMI-based method enables an existing fuzzer to achieve up to 6.6× more code coverage. In total, VMIGEN allowed us to discover 33 previously unknown bugs, which we disclosed in a coordinated way to the affected vendors.**

Index Terms—**Fuzzing, Harnessing, Windows**

1. Introduction

In recent years, fuzzing has proven to be a highly effective method for identifying faults in a wide range of software systems. A groundbreaking step forward was *American Fuzzy Lop* (*AFL*) [65], which proposed lightweight code instrumentation to provide coverage feedback. This has led to extensive research focusing on different aspects of fuzzing, such as improving the scheduling of inputs [8], [59], [67], refining mutation strategies [37], [39], [40], [50], or overcoming common obstacles [2], [9], [64]. Over time, many different angles have been analyzed, and much progress has been made, as reflected in state-of-the-art fuzzers such as *AFL++* [18] and *libAFL* [19]. Curiously, however, we have not overcome several noteworthy limitations.

First, fuzzers typically struggle with highly stateful applications, as setting up the correct internal state is nearly impossible for a fuzzer to infer within a reasonable time frame [3], [13]. Second, and in a similar vein, fuzzers often struggle with generating complex, highly-structured inputs (even though domain-specific approaches such as grammar fuzzers can partially address this issue) [69]. Third, most existing tools heavily depend on access to the source code for efficient instrumentation, harness generation, or inferring how to craft specific inputs. While seemingly mundane, this reliance has led academia to largely avoid ecosystems without source code. A prime example underlying this observation is Windows: not only are most applications proprietary, but even the kernel is closed source. Despite its prevalence, little research has been conducted to remedy the situation.

That said, there are several relevant contributions in this area: *NtFuzz* [11] aims at fuzzing the closed-source Windows kernel without requiring any prior domain knowledge. However, it is restricted to 32-bit versions of Windows and does not incorporate code coverage. Similarly, while *Syzkaller* [15]—proven effective for testing the Linux kernel—offers experimental, basic support for Windows, it also lacks support for code coverage. On the other hand, there are tools such as *kAFL* [57] and *what the fuzz* [61] that offer coverage-guided solutions for fuzzing arbitrary closed-source Windows components. However, their flexibility comes at a cost: They are *not* designed as out-of-the-box solutions and require the user to develop individual harnesses to enable fuzzing of the system under test. Unfortunately, this is difficult and does not scale when dealing with complex operating system components. These often expect highly structured inputs and rely on intricate, stateful interactions that a human expert needs to model to enable effective fuzzing. This requires an in-depth knowledge of the target's internal workings, which is hard to obtain given the closed-source environment.

Approach. In this paper, we present the design and implementation of VMIGEN, a novel and practical method to overcome the challenges of highly-structured input and stateful fuzzing in closed-source environments. Our fundamental insight is that we already have the technical means to fuzz components of Windows, that is, generating inputs and steering execution via code coverage, but we lack the capability to meaningfully relieve the human domain expert from identifying the input structure and accounting for state. At the same time, we observe that the current approach of viewing our target in isolation robs us of valuable information that can be obtained from observing its interactions with an analyst in a controlled test setup, where other components *successfully communicate* with it and trigger a variety of different states. By closely observing the system, we can monitor such interactions based on the inputs to the software and capture them together with the system state in real time. The information contained in these interactions can then substantially benefit an existing fuzzer by providing it with a corpus of informed inputs and the state it needs to successfully explore the target.

On a technical level, we implement our idea with *Virtual Machine Introspection* (*VMI*), a technique for monitoring and analyzing the behavior of virtual machines (VMs) at runtime on the hypervisor level without affecting the operation of the VM [25], [51]. We monitor the entire system's runtime and use VMI to automatically observe relevant input channels used by the component under test. As representative examples demonstrating the effectiveness of combining VMI and fuzzing, we focus on two mechanisms of the Windows operating system where interactions frequently cross a privilege boundary: IOCTL requests from user space to kernel drivers and RPC calls to services, which often run as privileged processes. Crucially, our VMI-based approach allows us to monitor and provide well-structured inputs to a fuzzer *without* requiring source code or external analysis techniques. Beyond providing informed inputs, we create a snapshot of the entire system via the hypervisor when we discover a new interaction with our target component. This snapshot comprises the complete system state, including all preconditions required to execute the deeper code logic within the software component we want to test.

Using the snapshots and observed inputs, an existing snapshot-based, whole-system fuzzer can then effectively test stateful software expecting highly-structured input *without* needing its source code. In other words, we set up existing fuzzers for success by providing a starting point where the required context is already set up—similar to a manually generated fuzz harness—and a corpus of informed inputs. We stress that our contribution is not the use of snapshots, as proposed by previous works [56], [60], but instead using VMI to automatically guide the capture of snapshots to points in the execution where complex, stateful OS interactions can be observed. In other words, our contribution indicates *when* to take snapshots, and it also allows us to extract relevant inputs from the observed interactions.

Our evaluation shows that a fuzzer with access to the snapshots and inputs provided by VMIGEN reaches a significantly higher code coverage than without access to this information. In numbers, the usage of VMIGEN resulted in $4.4\times$ more coverage on average, with a maximum increase of up to $6.6\times$. At the same time, using VMIGEN resulted in a considerable increase in bugs found during our evaluation, uncovering 23 more bugs than the baseline, which lacked access to snapshots and inputs. Several of the found bugs allow an attacker to elevate local privileges or even escape from highly restricted sandboxes. Following coordinated disclosure, Microsoft and Kaspersky assigned five CVEs.

Contributions. In summary, our main contributions are:

- A VMI-based technique to capture complex inputs and state in the form of snapshots, enabling fuzzing complex targets without source code.
- A prototype, called VMIGEN, that allows for testing closed-source Windows kernel drivers and RPC services, both of which elude current fuzzers.
- A comprehensive evaluation showing that VMIGEN yields significantly more code coverage and bugs.

We release our code at https://github.com/MPI-SysSec/VMIGen to foster research in this area.

2. Technical Background

Before diving into details, we highlight two Windows components eluding fuzzing and briefly introduce VMI.

2.1. Challenging Windows Components

The focus of fuzzing research has primarily been on open-source Linux applications [6], [14], [42], [69], with significantly less work in the Windows domain [11], [32], [66]. In this work, we focus on two particularly interesting components that elude effective fuzzing.

Kernel-Mode Drivers. In contrast to user-mode applications, which have limited ability to interact with the system and hardware, kernel drivers run in the most privileged protection ring (*ring 0*) with complete control over the OS and hardware. Most device drivers expose an IOCTL interface, allowing user-made applications to communicate with them using separate syscalls. Each IOCTL operation includes a control code specifying the type of operation and optional input/output buffers. From an attacker's perspective, kernel-mode drivers are attractive targets because they run with the highest privileges, making kernel compromise possible. Additionally, each driver handles unique operations with specific inputs, requiring individual input validation to prevent vulnerabilities. To reduce the risk of malicious drivers, Windows enforces strict code-signing.

While kernel drivers are high-value targets, manually analyzing closed-source drivers—whether from Windows or third parties—is tedious and labor-intensive. Tools like *iofuzz* [41] and *IOCTL Fuzzer* [49] hook and mutate IOCTL calls but lack code coverage feedback, which limits their effectiveness. A more sophisticated tool, called *what the fuzz* (*WTF* [61]), uses snapshots to store the context of an IOCTL call and then emulates the snapshot while mutating the input data of the kernel driver. Using snapshots allows the fuzzer to test a target driver while preserving its current state. However, the snapshot creation is manual, relying on the analyst rather than automation.

MSRPC. The second component we discuss is the Microsoft Remote Procedure Call interface, Microsoft's variant of the Distributed Computing Environment / Remote Procedure Calls (DCE/RPC) specification [21]. It allows C/C++ programs to exchange data and call functions located in foreign address spaces of other processes, even across system boundaries, effectively enabling distributed client/server applications. According to the specification, an MSRPC server groups its functionality into interfaces, each identified by a Universally Unique Identifier (UUID) and version number, which are made available to potential clients. Before a client can use an RPC interface, it must first create a *binding handle*, which is an object that represents a logical connection to the interface of the server. If a server has to maintain client-specific state across several calls, the client can obtain a *context handle* for use in subsequent calls. Finally, the actual MSRPC call data is serialized by Windows' RPC runtime before it is sent to the server and deserialized by the corresponding remote component of the server. Hence, in contrast to kernel-mode drivers, the RPC runtime generically handles input processing, i.e., (de-)serialization, before processing the input. How the data is transferred to and parsed by the server depends on which RPC protocol sequence is used.

RPC servers often run with elevated privileges and offer their functionality to less privileged processes. As a result, they represent a clear privilege boundary that is attractive from an attacker's point of view. Exploitable vulnerabilities in these interfaces can result in anything from a local denial of service attack to the full compromise of a remote system. Due to the nature of these RPC interfaces, analyzing them requires significant manual effort. Common approaches like fuzzing are limited due to the previously outlined challenges, especially related to state. Tools such as *RpcView* [7] enable recovering certain information, such as function prototypes, for undocumented RPC server interfaces. However, many functions expect binding or context handles to refer to an existing state. These state requirements in combination with (de-)serialization prevent the approach of naïvely mutating the arguments of the RPC interface. Instead, it would be necessary to manually generate harnesses, which set up a state and eventually fuzz the interface. However, this also requires significant manual reverse engineering to understand the interfaces' in- and output behavior.

In summary, both Windows kernel drivers and RPC interfaces are security-sensitive targets that elude current fuzzing attempts due to their closed-source nature, paired with complex inputs and inherent statefulness.

2.2. Virtual Machine Introspection

Before discussing our approach towards solving these challenges, we provide a brief background on VMI. In virtualized environments, virtual machines (VMs) are typically controlled by a privileged component called the *hypervisor* or *virtual machine monitor*. While the hypervisor usually runs in an elevated root mode on a host OS, with full access to CPU instructions and all resources (e.g., system memory), VMs run their own separate guest OS in non-root mode. When executing code in non-root mode, the hypervisor can restrict the processor from performing a pre-specified set of operations. These operations include, among other things, access to certain memory areas, reading or writing model-specific registers, and executing particular CPU instructions. Whenever a VM in non-root mode performs such an operation, the processor triggers a *VM-exit* event and passes the execution to the hypervisor, which decides how to proceed. This architecture grants the hypervisor full control over the VM's resources. However, the hypervisor initially does not know the semantic meaning of the resources, as it only has a low-level view of the guest OS' resources, such as memory or CPU context. In addition, it cannot trivially call functions from the guest OS to gather information about its state. This phenomenon is known as the *semantic gap* [10].

The term *Virtual Machine Introspection* (VMI) refers to methods for bridging this gap and gaining insights into the state and activities of a VM from the outside [25], [51]. To accomplish this task, the hypervisor must acquire knowledge of the OS's internal structure definitions and workflows and the location of these components in memory. Depending on the specific information the hypervisor requires, various approaches can be taken to overcome the semantic gap [16], [22], [29], [52]. For example, the hypervisor can configure the processor to cause VM-exits during specific events inside the guest OS, such as system calls or thread switches, and then make reliable assumptions about the CPU context depending on the cause for the VM-exit. Previous work has shown that it is possible to use VMI for intrusion detection [25] and Linux kernel fuzzing [31], [60]. In contrast, we intend to use VMI to enable fuzzing of complex, closed-source targets.

3. VMIGEN: Using VMI for Fuzzing

We now present the design and implementation of VMIGEN, a VMI-based method that enables fuzzing of complex, stateful targets based on an in-situ analysis of system interactions. Most importantly, we show that VMI enables us to observe the inner workings of closed-source components *without* requiring target-specific manual harnessing. VMIGEN's main advantage is the flexibility that allows arbitrary operating system components to be fuzzed with a one-time effort to develop VMI techniques for the underlying communication channels that control their interactions.

Before we outline VMIGEN, we discuss the technical challenges it needs to overcome. Then, we show a high-level overview of our VMI-based approach, followed by a detailed discussion of how the method is tailored to address the unique challenges associated with each use case.

3.1. Technical Challenges

Recall that our goal is to capture stateful interactions of some closed-source component. A naive approach is to reverse-engineer the target, understand its inner workings, and then write a custom harness specifically for this target. This is a highly complex manual effort that does not scale and requires a human domain expert. To address this limitation, we may consider sending *random* inputs to the component, thereby minimizing understanding of its internals. While this may cover superficial logic, it will fail to explore deeper code logic, as random inputs are unlikely to set up the required state correctly. Our key idea is to dynamically

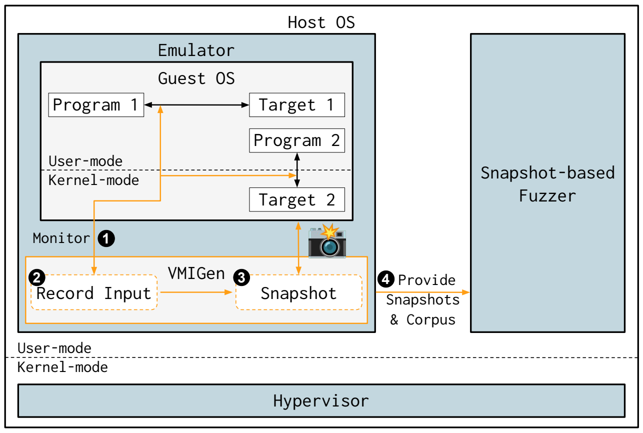


Figure 1: Overview of VMIGEN's architecture

intercept legitimate inputs sent to our target under test during regular operations performed by an analyst. These inputs will be well-formed, and the target is by design in the "right" state. Still, two challenges remain: First, it is unclear *when* and *where* another component interacts with our target. Second, the interaction may not be interceptable by traditional mechanisms, such as in a debugger or via code injection. This is the case, for example, for anti-virus agents that communicate with their driver modules or for parts of the Windows kernel protected by PatchGuard. In short, we need to address two challenges, namely to capture:

1) Global interactions between arbitrary processes and our target.
2) Interactions even if the communication is protected.

In the following, we discuss VMIGEN, our VMI-based approach to solve these challenges and enable fuzzing of complex, stateful closed-source components.

3.2. High-Level Overview

A general overview of our approach to overcome these challenges is presented in Figure 1. VMIGEN is capable of ❶ monitoring the runtime of the entire operating system, particularly the interactions of user-space programs with kernel components, and the communication between processes in user space. VMI enables us to ❷ automatically capture interesting inputs to a target component (e.g., we can record parameters of system or function calls). At the same time, VMIGEN can ❸ automatically create whole-system snapshots if it detects any process in the system that interacts with the designated fuzzing target. The snapshot captures the entire state of the full system and thus all potentially relevant preconditions necessary for executing deeper code logic. This snapshot ❹, combined with the collected input corpus, can then be used by an existing snapshot-based, whole-system fuzzer for comprehensive testing.

As shown in Figure 1, VMIGEN operates entirely within the host OS because it is designed to be an addition to an existing emulator that executes the VM. This design allows it to work independently, eliminating the need for additional components, such as an agent running inside the guest OS. By implementing various VMI techniques, which we describe in detail later in this section, VMIGEN can retrieve basic information about each process and thread executing within the guest. Furthermore, it can introspect calls to the fuzzing targets in real time, which enables it to extract input data and create snapshots on the fly.

3.3. Windows-Specific Design Choices

While our design is portable to different OSes, its practical implementation requires OS-specific adaptations. This is because any VMI mechanism is closely tied to the underlying OS semantics. We discuss portability in Section 5. In our work, we focus on Windows, which features a large attack surface due to its complex, closed-source components. We now discuss design details that enable VMIGEN to fuzz both kernel drivers and RPC interfaces running on Windows. Additionally, we explain how VMIGEN is able to obtain the required information about processes and threads of a VM running Windows.

Using VMI to Extract Process and Thread Information. To observe the relevant behavior of the VM in real time, VMIGEN must extract essential basic information from the guest OS. Particularly important is the knowledge of all running processes and their threads. VMIGEN automatically extracts all required information from the guest OS by intercepting the Windows thread scheduler. More precisely, each time the execution is about to switch to a different thread, the thread scheduler writes to a certain model-specific register (MSR) using the *wrmsr* instruction. When hardware-assisted virtualization is enabled, the hypervisor can be configured so that the execution of this instruction triggers a VM-exit event, allowing the guest to be intercepted at this point. Consequently, the hypervisor has full access to the CPU context and memory. When assuming that the processor is at a known location in the thread scheduler at the time of the VM-exit, VMIGEN can then extract pointers to OS-internal process and thread structures from the register state to perform the actual VM introspection.

Monitoring System Calls and Returns. Equipped with the ability to extract process and thread information, VMIGEN also utilizes the underlying hypervisor to intercept all system calls and returns, which is leveraged to monitor the respective channels for communicating with both kernel-mode drivers and user-mode RPC servers. In both cases, additional introspection steps must be taken to obtain all relevant semantic information and properly evaluate the context of the intercepted system call.

Kernel-Mode Driver Communication. IOCTL requests from a user-mode application to a kernel driver are ultimately made by the *NtDeviceIoControlFile* syscall, the prototype of which is shown in Listing 1. In the context of this work, the parameters *FileHandle*, *IoControlCode*, *InputBuffer*, and *InputBufferLength* are particularly important. To build up a usable and diverse input corpus for a driver, VMIGEN collects all inputs to a driver associated with an

```
NTSTATUS NtDeviceIoControlFile(
  HANDLE            FileHandle,
  HANDLE            Event,
  PIO_APC_ROUTINE   ApcRoutine,
  PVOID             ApcContext,
  PIO_STATUS_BLOCK  IoStatusBlock,
  ULONG             IoControlCode,
  PVOID             InputBuffer,
  ULONG             InputBufferLength,
  PVOID             OutputBuffer,
  ULONG             OutputBufferLength
);
```

Listing 1: NtDeviceIoControlFile function prototype [45]

IOCTL code not yet seen at that time. The corresponding input data resides at a virtual address in the current executing process's context, specified by the *InputBuffer* parameter. The size of this data buffer is specified in the parameter *InputBufferLength* accordingly. In addition to collecting the input data, VMIGEN also creates a snapshot of the CPU context and the entire memory state of the VM, which can then be passed to a snapshot-based fuzzer later on.

Note that this approach cannot match the *FileHandle* parameter to a driver device, because a file handle is an opaque reference to a device object that is only valid in the calling process. In contrast, the actual mapping to the driver uses internal structures, such as a handle table, in the kernel. As a result, it is impossible to trace which device an IOCTL request is sent to by only examining the *NtDeviceIoControlFile* syscalls without additional context from the guest OS. Moreover, it is infeasible to determine if another process, with a different handle referencing the same driver, is sending inputs. VMIGEN addresses this challenge by mapping filenames (including driver device names) to their corresponding handle values for each process, effectively managing its own handle table. This is accomplished by intercepting all *NtCreateFile* and *NtOpenFile* syscalls to extract the filenames that a program wants to open, while also hooking the respective returns to extract the handle value upon success. Concurrently, syscalls to *NtClose* are monitored to invalidate mappings when the handle is closed.

RPC Calls over ALPC. The mechanism commonly used in Windows to transport RPC communication within the boundaries of a system is the *Advanced Local Procedure Call* (ALPC) IPC mechanism. While other methods exist to transport the underlying RPC data, Microsoft considers ALPC to be the most efficient technology to perform calls between different processes on the same computer [46]. Hence, our proof of concept focuses on RPC communication based on ALPC. Similar to driver communication, ALPC-based data exchange relies on a central syscall: *NtAlpcSendWaitReceivePort*. Its prototype is shown in Listing 2 in the appendix. The essential parameters in this syscall are *PortHandle* and *SendMessage*. Using this system call, an application can send a message of arbitrary size to another process, provided that the two processes have previously established a connection through an ALPC port. Each message, referenced by the *SendMessage* parameter, is initially preceded in memory by a generic structure detailing information such as the message length. Then, specific data follows that can consist of any number of bytes, which the recipient interprets differently depending on the context of the message exchange.

Again, VMI is needed to provide the necessary semantic information, which gives important context to each call to the ALPC messaging syscall. Complementary to the logic for tracking driver device handles, VMIGEN records all ALPC handles and maps them to their corresponding port name by intercepting the *NtAlpcConnectPort* and *NtAlpcConnectPortEx* syscalls. Whenever a handle is closed using *NtClose*, the mapping is invalidated. At this point, however, all information about the context of the RPC connection is still missing and therefore needs to be extracted from the transmitted messages in a subsequent step. As described in Section 2.1, RPC clients are generally required to establish a binding to the interface of an RPC server prior to invoking its functions. This binding process involves the client transmitting a specialized packet via ALPC, which encapsulates the UUID of the server's interface to which the client intends to bind. Following ALPC messages, which then contain actual RPC call data, include the information on the binding in-use [20]. Knowing their structure, VMIGEN can track all bindings in real time and thus extract semantic information about the called RPC interface functions from the individual ALPC messages. This enables VMIGEN to collect the input data for every previously unrecorded call of a function from an RPC interface and simultaneously create a snapshot of the entire operating system at that point in time.

3.4. Integration with Fuzzing

Given our VMI-based capturing mechanism, which collects inputs and snapshots of "interesting" interactions of system components, we now outline how to integrate a snapshot-based fuzzer, such as *WTF* [61] or Nyx [56], to utilize this information for effective testing.

First, the fuzzer selects a snapshot provided by VMIGEN, which is then fuzzed for a user-specified time before proceeding to the next snapshot. Across the fuzzing campaign's runtime, we spread fuzzing time evenly across all snapshots. While fuzzing a particular snapshot, the fuzzer repeatedly follows a typical fuzzing loop. Each iteration proceeds as follows: The fuzzer randomly selects an input from the queue, which is then mutated and executed. Note that the initial queue is not empty (as is often the case) but contains all inputs that VMIGEN observed at runtime. While the particular mutations are fuzzer-specific, they usually do not include *structure-aware* mutations. For example, if VMIGEN observed nested pointers within a struct, the fuzzer may not be able to correctly mutate these nested objects; we leave this as interesting future work, as it is orthogonal to the goals of VMIGEN. During the execution of the input, the coverage is observed and reported to the fuzzer. After execution, the fuzzer saves the input to the queue if it is interesting, i.e., it leads to new behavior, or discards it otherwise. If the time budget assigned to the particular snapshot has not been exhausted, the fuzzer restores the state to it and chooses another input from the queue. In the (rare) case an input leads to a crash, the fuzzer saves this input for further analysis.

3.5. Implementation

To demonstrate the feasibility of the proposed approach, we implemented a prototype of VMIGen. All VM introspection features, including the mechanisms for context understanding and automated snapshot generation, are implemented directly in QEMU [5]. Consequently, VMIGen uses a modified KVM [34] in combination with an extended QEMU to intercept syscalls executed by the VM.

Snapshot-Based Fuzzing. To fuzz the individual snapshots with the help of the respective input corpus, we use a modified version of the coverage-guided and snapshot-based fuzzer *what the fuzz (WTF)* [61]. Specifically, we extended *WTF* to understand the custom memory snapshot files produced by VMIGen. Moreover, while we use a modified version of the harness provided by *WTF* to inject inputs into the *NtDeviceIoControlFile* syscall, we have developed an additional harness specifically for the *NtAlpcSendWaitReceivePort* syscall. This new harness enables us to fuzz RPC calls using ALPC, allowing communication across multiple processes.

Bochs Emulation. *WTF* allows users to execute the respective snapshots using three different backends, namely the Bochs emulator [36], the Windows Hypervisor Platform [48], and KVM. While the last two options allow faster executions by leveraging hardware virtualization, Bochs offers a crucial advantage for our use case: By effectively emulating all instructions in software, the Bochs backend can measure both basic blocks and edge coverage *without* the need for further instrumentation. This is particularly useful when it is unclear in advance which binaries are responsible for processing parts of the input sent to the component under test. In such cases, it is difficult to instrument them for coverage measurement beforehand without prior knowledge. Using Bochs as the backend, snapshots can be fuzzed directly, eliminating the need for manual interaction despite a significant slowdown.

4. Evaluation

To evaluate the effectiveness of VMIGen, we conduct a series of experiments involving real-world targets. We analyze the actual effect of the snapshots and inputs provided by VMIGen when fuzzing six closed-source kernel drivers and six RPC interfaces of Windows. Last, we study the ability of VMIGen to contribute to the discovery of bugs.

4.1. Experimental Setup

For our experiments, we used VMIGen to collect inputs and snapshots within a Windows 10 VM running with 4 GB RAM and one virtual CPU. We provide additional information on our targets in Table 2 in the appendix. When testing a target, we may need to manually interact with the target to trigger interactions that we can observe using VMI. We explain this preparation step for the different targets below. We performed the comparative evaluation using different fuzzer configurations on a machine with an Intel Xeon Gold 6230R CPU and 192GB of RAM. We fuzzed each target in each configuration for 24 hours on a single core. To account for the randomness of the fuzzing process, we repeated each fuzzing campaign ten times and followed the best practices by Klees et al [35] and Schloegel et al. [54].

Note that we evaluated our approach on Windows 10 rather than Windows 11, as Windows 10 remains the most widely used version of the operating system in practice, ensuring that any potential issues would have a broader impact [62]. We have verified whether security-relevant findings that might depend on the Windows version also affect Windows 11. With a few exceptions, we reproduced bugs using the latest version of Windows.

Ethics Considerations. As our work uncovers vulnerabilities in software—some of which could potentially be exploited by malicious actors to elevate local privileges or escape from highly restricted sandboxes—we conduct a *coordinated disclosure* and follow best practices. All findings classified as security-relevant by vendors have been responsibly disclosed, and we worked with the vendors towards a patch (e.g., by providing technical details on how the software faults can be triggered). We did not request CVEs [55], but vendors have assigned five CVEs as they saw fit.

Open Science. We will release our code and scripts at https://github.com/MPI-SysSec/VMIGen.

4.2. Fuzzer Configurations

To evaluate VMIGen's impact on fuzzing, we need to pair it with a snapshot-based fuzzer. We have chosen *WTF*, a state-of-the-art Windows snapshot fuzzer, as baseline. An alternative implementation could be based on kAFL [57] or Nyx [56], even though this would require significant engineering effort and a partial reimplementation of VMIGen. Implementing VMIGen on top of both also offers comparably little value, as our goal is not to compare kAFL with *WTF* but to assess the benefits introduced by using VMIGen. Comparing the extended fuzzer against a state-of-the-art tool in the domain proves difficult: The only other state-of-the-art Windows kernel fuzzer, *NtFuzz* [11], focuses on extracting and mutating syscall parameters for 32-bit systems. There is also *IOCTL Fuzzer* [49], but it was last updated in 2011 and does not support Windows 10. Finally, *Syzkaller* [15] provides basic support for Windows, even though it is primarily designed for Linux. However, we observe that its support is so limited that it does not even feature code coverage. Hence, we do not believe any of these tools can provide a meaningful comparison to VMIGen, underlining the dire need for more research in this area.

Instead, we resort to a comparison with *WTF* to quantify the impact of VMIGen and design several ablations to isolate the impact of different components. By using the same underlying fuzzer (*WTF*) across all configurations, we ensure that differences in the observed performance can be directly attributed to the specific improvement (i.e., snapshots or informed inputs) rather than general fuzzer implementation details. This enables a fair evaluation that precisely measures our contributions. In detail, we use these configurations:

(i) BASELINE. Essentially *WTF*, our baseline has neither access to snapshots containing special states nor access to the informed inputs collected via VMI. As *WTF* by design requires a snapshot for fuzzing, we provide the baseline with a single initial snapshot that we manually created while performing a dummy interaction with the component under test. When testing kernel drivers, we ensure that, at the time this snapshot was taken, our own test application had opened a handle to the device object and sent an IOCTL request with an input composed entirely of null bytes to the driver. During early testing, we found that the fuzzer failed to identify valid IOCTL codes, significantly limiting its capability to explore code. This is due to the fact that the fuzzer would have to guess a 32-bit value correctly to unlock new code coverage, a known fuzzing roadblock [2]. Therefore, we opted to include an initial input with one valid IOCTL code. While both of these changes provide the baseline with an advantage compared to an uninformed fuzzer, they allow for precisely measuring the impact of having access to multiple snapshots or informed inputs.

We use a similar setup for RPC targets: We provide a single snapshot to fuzz in the context of a test application, ensuring it has been previously bound to the desired RPC interface (without any further state). The input corpus contains a single ALPC message consisting solely of null bytes. Hence, the fuzzer cannot derive any information about the structure of valid interface calls from this initial input.

Furthermore, we executed the test applications with administrator privileges, as in some cases the use of kernel drivers and RPC interfaces can be restricted to privileged users only. Neglecting this restriction could negatively impact the coverage achieved by the baseline.

(ii) Inputs-Only Configuration: INPUTGEN. Our second configuration extends the baseline with the input corpus collected by VMIGEN. In terms of the starting point, it uses the same snapshot as the baseline. For drivers, having access to the input corpus means INPUTGEN has access to all the IOCTL codes and associated input buffers sent at runtime. For the RPC targets, this corresponds to the input corpus containing all serialized calls to the respective RPC interface that VMIGEN has observed at runtime.

(iii) Snapshots-Only Configuration: SNAPSHOT-GEN. In contrast to the previous configurations, this one uses several different snapshots, each originating from different real process contexts. VMIGEN created these snapshots when a process sent a previously unseen input to the respective component. In the context of kernel drivers, the IOCTL code serves as a decision criterion as to whether an input has already been seen or not; in the case of RPC interfaces, each process that calls a function that has not yet been called is snapshotted. Since multiple snapshots enable the fuzzer to initiate fuzzing from different starting points, a decision must be made regarding the duration of fuzzing each snapshot before moving to the next. For our evaluation, we allocate equal time for fuzzing each snapshot, dividing the 24h runtime by the number of available snapshots.

SNAPSHOTGEN has no access to the inputs collected via VMI, like the BASELINE. Again, we provide a valid IOCTL code with the input corpus when fuzzing the drivers.

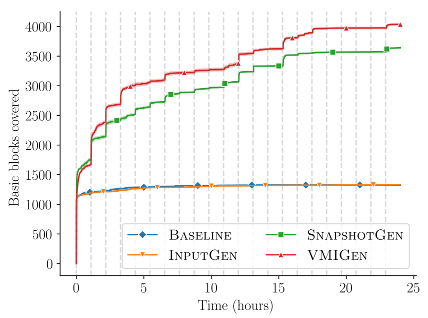
(iv) Full VMI Configuration: VMIGEN. Representing all our contributions, VMIGEN has access to both snapshots and inputs collected via VMI. To reiterate, its attention is distributed fairly across all snapshots by dividing the available evaluation time by the number of snapshots while giving the fuzzer access to all VMI-derived seeds. Thus, in this configuration, the respective targets are fuzzed in potentially different states while the input corpus helps the coverage-guided fuzzer to generate valid inputs.

4.3. Kernel-Mode Drivers

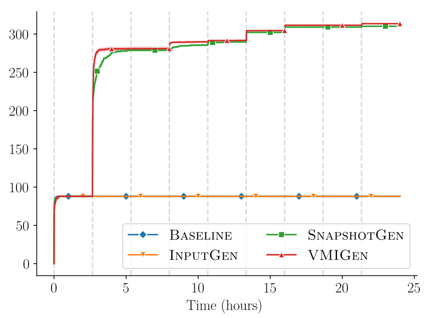
In our first experiment, we use VMIGEN to target six Windows drivers (*afd.sys, ksecdd.sys, spaceport.sys, mfehidk.sys, mfencbdc.sys, miniicpt.sys*). Three of our targets are drivers available on a default installation of our test system. The other three drivers are part of trial versions of two widely used anti-virus programs, namely *McAfee Total Protection* [43] and *G DATA Total Security* [23].

Preparation. Before we can fuzz the drivers, we need to obtain the snapshots and extract the corresponding inputs using our VMI-based approach. We collected all snapshots and inputs for the *afd.sys* driver and the *ksecdd.sys* driver automatically within 15 minutes using VMIGEN, no manual interaction was needed. According to our observations, user-mode processes frequently communicate with these drivers even without specific human interaction with the OS by an analyst. Similarly, for the three anti-virus drivers, we automatically retrieved all data during normal system use after the respective software was installed, again within approximately 15 minutes. In contrast, no out-of-the-box communication occurs for *spaceport.sys* without a user manually initiating various interactions. This driver is utilized by the OS when a user manages *storage spaces* to store multiple copies of data on different drives, aiming to protect against drive failure in a manner similar to RAID systems [47]. Ultimately, we observed a variety of different requests sent to the driver by manually creating, modifying, and deleting storage spaces through the corresponding user interface of the OS.

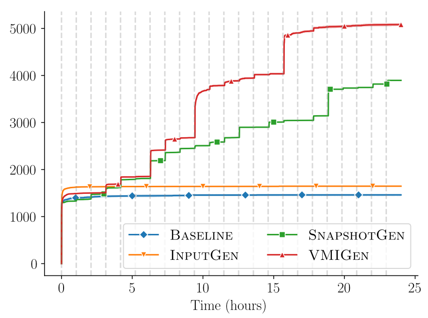
Coverage Analysis. After running all four fuzzer configurations ten times for 24 hours, we plot the coverage in Figure 2. These results show that the BASELINE and INPUTGEN configurations exhibit comparable performance, with both demonstrating suboptimal outcomes. An exception here is the *miniicpt.sys* driver. In Figure 2f, we see that both configurations reach at least 50% of the coverage accomplished by VMIGEN. Generally, BASELINE and INPUTGEN achieve their peak coverage almost immediately and quickly stagnate. For the target in Figure 2e, we were unable to provide a functioning initial snapshot because we could not create a handle for the driver in a custom test application.



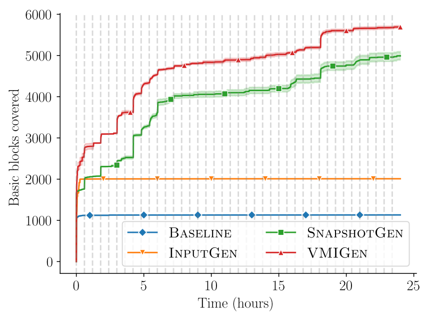
(a) afd.sys



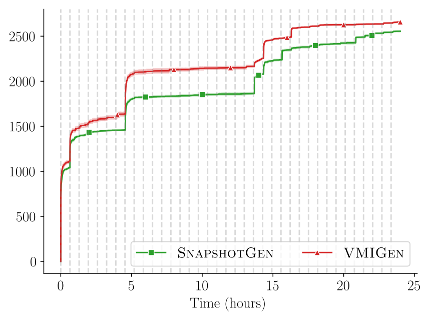
(b) ksecdd.sys



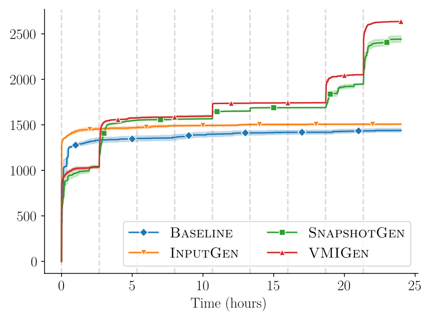
(c) spaceport.sys



(d) mfehidk.sys



(e) mfencbdc.sys



(f) miniicpt.sys

Figure 2: Coverage plots of driver targets. Vertical lines represent the time when a new snapshot is scheduled.

TABLE 1: Overview of unique bugs found in drivers and RPC targets during our ablation study. InputGen and SnapshotGen are ablation studies of VMIGen.

	Target	**Description**	**#Bugs per Configuration**				**#Total Bugs**
			Baseline	InputGen	SnapshotGen	VMIGen	
Driver	afd.sys	Ancillary Function Driver for WinSock	0	0	0	0	0
	ksecdd.sys	Kernel Security Support Provider Interface	0	0	2	1	2
	spaceport.sys	Storage Spaces Driver	2	2	2	4	4
	mfehidk.sys	McAfee Anti-Virus Driver	0	0	7	7	7
	mfencbdc.sys	McAfee Anti-Virus Driver	–	–	1	5	5
	miniicpt.sys	G Data Anti-Virus Driver	2	2	2	2	2
RPC	wevtsvc.dll	EventLog Remoting Protocol RPC Interface	1	1	1	1	1
	localspl.dll	Print System Remote Protocol RPC Interface	0	0	0	3	3
	lsasrv.dll	Local Security Authority RPC Interface	0	0	0	0	0
	schedsvc.dll	Task Scheduler Service Remoting Protocol	1	1	1	1	1
	samsrv.dll	Security Account Manager Remote Protocol	0	0	0	0	0
	audiosrv.dll	Undocumented RPC Interface	–	–	0	4	4

In contrast, configurations with access to snapshots perform significantly better. To visualize the impact of switching to a new snapshot, we mark these points in time with gray vertical lines in Figure 2. We clearly see that switching to a new snapshot leads to an immediate increase in coverage in most cases. This occurs as the input stored in memory when a snapshot is taken—which is not part of the input corpus but of the snapshot itself—often leads directly to new coverage when executed by the fuzzer as-is.

Furthermore, we observe that combining the input corpus with snapshots has a more pronounced impact. The coverage achieved using the full capabilities of VMIGen is significantly greater for four of the six targets, as shown in Figures 2a, 2c, 2d, and 2f, compared to the coverage achieved by SnapshotGen. This indicates that the recorded input corpus can also be used more effectively in the presence of snapshots, as without first reaching the correct context, parts of the inputs (e.g., object IDs or handles) do not result in valid references, and, as a result, the inputs do not lead to any further coverage, even when having the correct structure.

We statistically evaluate our results using a bootstrap-based two-sample t-test and Vargha and Delaney's $\hat{A}_{12}$ test [54]. We find a statistically significant difference between all of them and measure a large effect size ($\hat{A}_{12} = 1.00$). Comparing VMIGen to the best-performing ablation, it finds a statistically significant higher coverage in all cases.

Bugs. We also analyzed the bugs found by different configurations. The deduplication and triaging of bugs in anti-virus drivers is a difficult process due to their proprietary nature, complexity, and the lack of debugging symbols, which resulted in an immense reverse engineering effort. Note that the bugs we found in the McAfee drivers were not confirmed by the vendor, as they can only be triggered by a privileged account and are therefore not considered security-relevant. Consequently, while we cannot guarantee the accuracy of our bug count in this regard, we estimate—based on our best efforts—that we discovered 20 unique bugs across five of the drivers. For *afd.sys*, the fuzzer found no crashes. Table 1 presents an overview of our findings. For the *ksecdd.sys* driver, the fuzzer was unable to trigger crashes in the BASELINE and INPUTGEN configurations. However, with the help of SNAPSHOTGEN, it was able to identify two bugs, one of which was also found using VMIGEN. This initially counterintuitive outcome—after all, VMIGEN includes the same snapshots as SNAPSHOTGEN, and both achieve roughly the same coverage, see Figure 2—can be explained by the random nature of the fuzzing process. Regarding *spaceport.sys*, the fuzzer found at least two bugs that could be triggered without requiring any specific state in all configurations. Moreover, VMIGEN uncovered two additional bugs that required a specific context to be set up via previous IOCTL requests. While we could not apply BASELINE and INPUTGEN to test the anti-virus driver *mfencbdc.sys*, we also did not discover crashes in the other McAfee driver, *mfehidk.sys*, without our snapshots. Only for the *miniicpt.sys* driver, all configurations found all crashes.

Ultimately, all bugs found can only be triggered by privileged administrator accounts, as they reside in code only reachable with extended privileges. We stress this does not hold in general: During recent testing, we also found a heap corruption in a widely used kernel-mode driver of Kaspersky that did not require elevated privileges.

Case Study. One of the bugs found in McAfee Total Protection's [43] *mfencbdc.sys* driver is particularly interesting. VMIGEN captured runtime interactions between the client and the kernel driver that, when fuzzed, led to overwriting a stack buffer with arbitrary data passed in the IOCTL buffer. Based on our manual reverse engineering, we understand that data is copied from the IOCTL buffer with a length controlled by the sender to a fixed-size stack buffer.

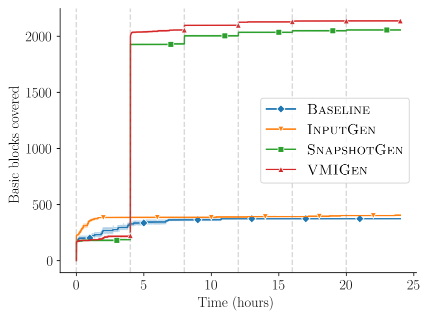
We could not run BASELINE and INPUTGEN for this driver due to the inability to open a device handle, which is required to send requests to the driver. Through reverse engineering, we discovered that when the driver device is opened, a specific function determines whether the corresponding user-mode process is allowed to access it. However, the underlying logic of this function was found to be rather complex, calling for a deep manual analysis beyond the scope of this paper. Therefore, we can only speculate if external programs (such as a fuzzer harness) could communicate with the driver. So far, we could only reproduce the bug by issuing the IOCTL request via process injection from one of the user-mode anti-virus processes itself. This requires elevated rights and a certain security option in the anti-virus software to be disabled. Although this bug is not critical to security, it illustrates the benefits of VMIGEN: Conventional fuzzing would never find this bug unless significant manual work is committed to analysis and harnessing.

4.4. RPC Interfaces

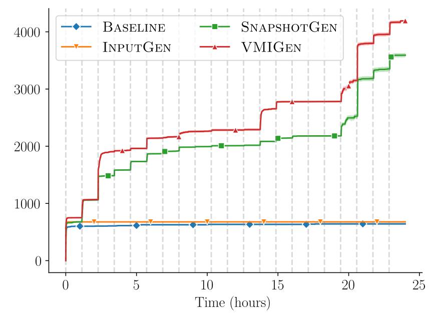
We performed the same experiment with six RPC interfaces to evaluate the benefits of VMIGEN for another set of targets. It is worth noting that many RPC servers run with elevated privileges and make their functionality available to less privileged users. This privilege boundary renders them as a particularly interesting target for attackers. For our evaluation, both the RPC client and server run on the same system; however, in real environments, they can even reside on separate devices with the same network domain. Thus, bugs can potentially be triggered remotely on other systems within the same network.

Preparation. For the RPC interfaces in *wevtsvc.dll*, *lsasrv.dll*, and *samsrv.dll*, VMIGEN managed to observe all calls without any manual interaction by an analyst in about 15 minutes. Regarding the *Print System Remote Protocol* RPC Interface, which is registered by the *spoolsv.exe* binary but has large parts of the actual logic implemented in the *localspl.dll* library, no interaction could be observed without additional manual steps. To retrieve the snapshots and inputs, we manually initiated a printing job by converting a text file to a PDF using Microsoft's virtual *Print to PDF* printer. The RPC interfaces in *schedsvc.dll* and *audiosrv.dll* also required manual interaction to collect as much information as possible. The *schedsvc.dll* interface handles scheduled tasks, managed through the *Task Scheduler*. All snapshots and inputs were gathered via basic operations such as creating, deleting, modifying, starting, and stopping scheduled tasks. While the remaining interface is undocumented, the name of the respective binary, *audiosrv.dll*, suggests that it is related to audio processing. Here, we have manually triggered the observed interaction by modifying the audio settings via the taskbar and playing a video in a web browser.

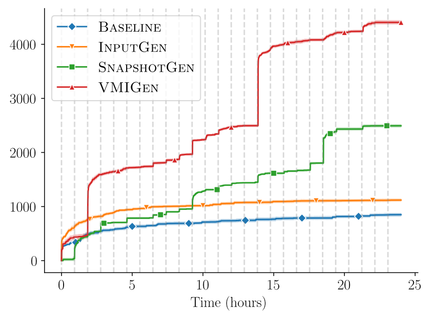
Coverage Analysis. The results of our coverage analysis for the RPC interfaces are shown in Figure 3. Similar to the previous experiment, we can clearly see that for the majority of the targets, the configurations without access to snapshots perform the worst. Interestingly, Figures 3a, 3c, and 3d show that—other than for the drivers—the maximum coverage achieved by the fuzzers is not attained immediately. We hypothesize that this is due to the inherently more complex input format, as the fuzzer needs to produce correctly deserializable RPC call data. Additionally, another contributing factor is that each RPC call typically involves the execution of more instructions, as it requires at least one context switch. In contrast, code running in the kernel is generally designed to be as lightweight as possible. This effectively reduces the speed of a single fuzzing iteration. To prevent the underlying fuzzer from generating unnecessarily large inputs, we impose a size limit on the mutator that is based on the size of the real inputs recorded with VMI. Compared to the other interfaces, we observed smaller inputs



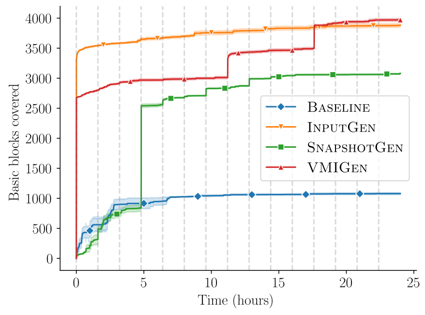
(a) wevtsvc.dll



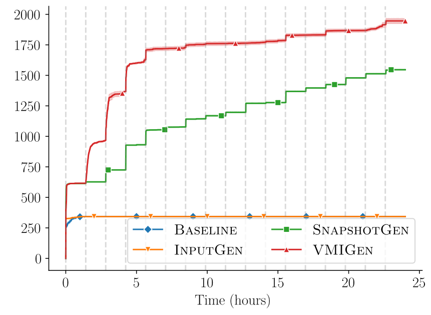
(b) lsasrv.dll



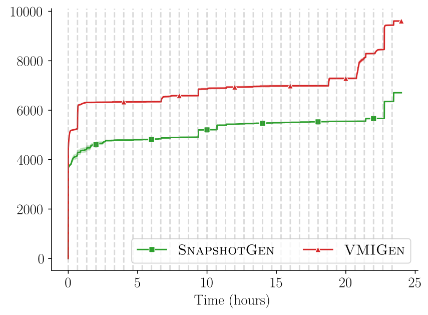
(c) localspl.dll



(d) schedsvc.dll



(e) samsrv.dll



(f) audiosrv.dll

Figure 3: Coverage plots of RPC targets. Vertical lines represent the time when a new snapshot is scheduled.

for those whose coverage is shown in Figures 3b and 3e. We assume that the coverage achieved in these cases was reached more quickly due to the stricter size limitations, which reduce the number of possible mutations.

For the targets depicted in Figures 3a, 3c, and 3d, we also observe that the fuzzer has benefited from the input corpus generated by InputGen. In Figure 3a, InputGen ultimately did not achieve additional coverage compared to Baseline but reached the final coverage more quickly. Furthermore, in Figure 3c—and especially in Figure 3d—it achieved more coverage overall. We believe this improvement stems from the fact that these two interfaces also export methods that can be called without a valid context handle, yet still require the call data to be correctly serialized. This effect is particularly visible for the target depicted in Figure 3d. At the same time, we suspect that the interfaces shown in Figure 3b and Figure 3e do not benefit solely from the collected inputs, due to their more restrictive input size. This is evident as there is no significant difference in the coverage achieved by the fuzzer between the Baseline and InputGen configurations. As it is undocumented, we could not provide a test application that binds to the interface depicted in Figure 3f, preventing us from evaluating Baseline and InputGen.

Overall, we observe a significant improvement in coverage due to the snapshots collected by VMIGen. This is an anticipated result, as many of the RPC functions expect a context handle as the first parameter, which is not present in the initial snapshot. Invoking such a function without a context handle will result in the RPC runtime rejecting the call. On the other hand, combining the input corpus with snapshots leads to noticeable improvements for four targets. This is because the fuzzer needs to guess the value of a valid context handle in the current snapshot context if it was not included in the initial input corpus. Again, we statistically evaluate our results and find a statistically significant difference between the Baseline and VMIGen for all RPC targets, with a large effect size of $\hat{A}_{12} = 1.00$. Comparing VMIGen to the best-performing ablation, results are statistically significant in all cases but *schedsvc.dll*.

Bugs. While evaluating the RPC targets, we found a total of nine unique bugs, summarized in Table 1. Only for *lsasrv.dll* and *samsrv.dll*, we found no bugs.

Interestingly, for the respective interfaces in *wevtsvc.dll* and *schedsvc.dll*, a single crash was observed for all four configurations. This behavior can be explained by the fact that the affected code does not require any special state to be executed (the affected RPC function does not receive a context handle). In contrast, all three bugs located in the RPC endpoints in *localspl.dll* and *audiosrv.dll* were triggered exclusively when the fuzzer profited from the entire feature set of VMIGen. Through manual analysis, we have confirmed that the respective crashes can only be triggered if the calling process holds a valid context handle, which the fuzzer then uses to craft new inputs. Consequently, both a certain state of the snapshot *and* specific information in the input corpus are essential in these cases, requiring the VMIGen configuration. At least on the local system, all bugs could be triggered by unprivileged users.

Case Study. While fuzzing the *Print System Remote Protocol* RPC interface, we observed an interesting crash. For specific inputs, an access violation occurred inside the *spoolsv.exe* process while trying to write to an invalid address. Further investigation revealed that the crash resulted from calling the *RpcIppGetJobAttributes* function, which requires a context handle as the first parameter. This handle can only be obtained by opening any existing printer using another RPC call. Thus, the crash only occurred on the third snapshot, as the required context handle was created by the second snapshot's RPC call. The crash occurs when the RPC function attempts to resolve an exported function with a hardcoded ordinal number from a DLL referenced in the internal printer structures. Based on our reverse engineering efforts, the referenced DLL appears to be a *port monitor* that takes over the communication of the user-mode print spooler to the respective drivers, which then communicate with the actual printer hardware. In our experiments, the DLL referenced in the structures is the *localspl.dll* itself (which seems unintended), and the given ordinal number resolves to a function that writes four null bytes to the address in register *R8*. When the function is called, *R8* contains a 32-bit value that is passed to the function as an argument by the RPC client and can be fully controlled.

This effectively allows an unprivileged attacker to write zero bytes to any address in the lower 4GB range, including both the heap and the thread stacks. Since the affected process *spoolsv.exe* runs with system privileges, when combined with another vulnerability (such as an information leak allowing the prediction of the address layout), it could potentially be exploited to perform a privilege escalation. Microsoft has also confirmed that this bug can, under certain circumstances, be exploited to escape the *Less-Privileged App Container (LPAC)* sandbox [27]. Consequently, this vulnerability poses a greater threat compared to typical privilege escalation bugs.

4.5. Additional Findings and Impact

As part of our evaluation, we applied VMIGEN to several other components to demonstrate the broad applicability of our approach. Specifically, we tested another RPC interface, a Windows kernel driver, and the kernel drivers included in the *Kaspersky Endpoint Security for Windows* [33] product. Through this process, we uncovered 4 additional bugs. These findings include a null pointer dereference and a heap corruption in two undocumented RPC interfaces within the *lsass.exe* process and an arbitrary pointer dereference in Microsoft's *storport.sys* kernel driver. Moreover, VMIGEN identified a heap corruption in one of the kernel drivers shipped with the installation of Kaspersky's solution. Notably, all these bugs can be triggered by low-privileged users, highlighting VMIGEN's effectiveness in identifying vulnerabilities across a diverse set of targets. Across the found bugs, five have been assigned a CVE (CVE-2024-43502, CVE-2024-43529, CVE-2024-49072, CVE-2025-24072, and CVE-2024-13614).

5. Discussion

Next, we discuss limitations and future work.

Human Effort. If the OS does not automatically generate VMI interactions, we rely on human assistance, which is often more cost-effective than reverse engineering. An analyst only needs to trigger actions without understanding the underlying processes. However, if simple interactions are insufficient or it is unclear how to trigger the component our approach offers limited improvement, as it can only extract knowledge from observed interactions. In addition, there was a manual one-time effort to develop both the VMI techniques and the generic harness for the communication methods on which the specific interactions rely.

Lack of Diversity. The effectiveness of our approach depends on the diversity of snapshots and their inputs. For manually written harnesses, the developer must bring the target into the desired state. In contrast, our approach relies on the assumption that sufficient state can be accumulated at runtime by the OS on its own or through limited user interactions that do not require in-depth knowledge. If this assumption does not hold, or if only a few interactions are observed, the snapshots and inputs offer limited value and may only marginally increase coverage compared to an uninformed fuzzer.

Limitations due to the Underlying Fuzzer. Our work focuses on *enabling* fuzzing rather than *improving* the fuzzer implementation details. For our evaluation, we rely on *WTF*, as its support for full-system snapshot emulation with Bochs best met our needs. However, this also means inheriting its limitations, such as difficulties handling paged-out memory and emulating devices. Due to the inherently complex nature of an OS, individual snapshots may perform worse due to uncontrollable factors. For instance, snapshots taken just before a thread or process switch may interrupt execution, reducing fuzzing efficiency.

Lack of Target-Specific Knowledge. VMIGEN's goal is to reduce human effort, while enabling fuzzing components that depend on stateful interactions. Consequently, we do not assume a human expert to provide target-specific knowledge required to tailor a fuzzer to the component under test. In some edge cases, this generic approach may miss target-specific bugs. This can occur when the component processes inputs with a complex structure, such as nested pointers in a struct. Despite VMIGEN capturing the interaction, the default mutators of fuzzers are structure-unaware and will fail to mutate such inputs meaningfully. Other targets may process inputs asynchronously, which breaks the fuzzing loop we rely on, where the assumption is that the target immediately processes received input. Similarly, some targets may contain bugs that require sending multiple consecutive inputs before restoring a snapshot. Future work could analyze how the information retrieved by VMIGEN can be harnessed by other tools than a fuzzer, for example, to identify the structure of inputs or asynchronous processing. This, in turn, could enable the fuzzer to be tailored to the component under test without requiring manual effort.

Applicability to other Components and OSes. We implemented introspection for two specific Windows mechanisms: IOCTL-based kernel driver interfaces and RPC servers communicating over ALPC. Our approach could be extended to other input/output OS mechanisms (e.g., named pipes). Since VMI itself is OS-agnostic, our design can be ported with some engineering effort to other OSes, including macOS and Linux. For example, Linux similarly implements many interactions between different components (e.g., between user and kernel drivers) as system calls, which boil down to `syscall`/`sysret` instructions that a hypervisor can intercept. As a consequence, the idea of VMIGEN can be applied here as well; however, its current Windows-specific implementation would need to be adapted. Among others, it would need to use Linux-specific VMI to extract process and thread semantics, require information about file descriptors, and need to adhere to Linux calling conventions. Additionally, a new base fuzzer needs to be selected, as *WTF* only fully supports Windows. We believe our approach provides the most value for a closed-source OS; for open-source software techniques such as *DIFUZE* [12] or *AFGen* [38] can help address these challenges.

Snapshot Strategy. Our prototype allocates equal time to all snapshots for a fair evaluation, but this may waste fuzzing time on less promising states. Similarly, the current snapshot creation and input collection logic could be optimized. Estimating snapshot fruitfulness or input value at runtime is challenging, suggesting the need for advanced heuristics and metrics in future work.

6. Related Work

Our work relates to fuzzing and Windows testing.

Snapshot-Based Fuzzing and Harnessing. Using snapshots in fuzzing is a widely accepted approach to address various challenges. For instance, Dong et al. used snapshots to fuzz Android apps [17], while Schumilo et al. applied incremental full VM snapshots for hypervisor-based snapshot fuzzing, targeting both regular and network applications [56], [58]. *Agamotto* [60] employed VM checkpointing to skip previously observed operations to speed up fuzzing. Geretto et al. proposed *Snappy* [26], which leverages adaptable and mutable snapshots, placing them deep in the execution trace. *Snapfuzz* [1] and *FIRM-AFL* [68], demonstrate the use of snapshots for fuzzing network functions or firmware images. All these works aim to either increase the efficiency of fuzzing (by avoiding rerunning costly initialization steps) or provide the user with a new primitive, namely, to place snapshots. In contrast, our contribution is not a *new* snapshot-based fuzzer, but VMIGEN *guides* an existing fuzzer to snapshotting interesting, relevant OS interactions. To this end, we use VMI to monitor the execution of the entire OS to create snapshots whenever complex interactions with the target of interest occur. Our method thus complements and further enhances the effectiveness of snapshot-based fuzzers.

In the domain of *harnessing*, *Fudge* [4] was one of the first works that proposed a design for generating *fuzz drivers* automatically to find bugs in libraries. Building on *Fudge*, *AFGen* [38] uses a bottom-up approach to automatically create harnesses that include proper initial program context. Galland and Böhme proposed the concept of *in-vivo fuzzing*, a technique where a valid execution context of a library interaction is fuzzed without requiring custom fuzz drivers [24]. In contrast, Ispoglu et al. proposed a top-down approach that analyzes the whole system for library calls, uses this information to extract an API dependency graph, and finally generates fuzzer stubs [30]. VMIGEN similarly intends to enable efficient and effective fuzzing; however, we focus on components beyond libraries and do not require static analysis of source code (which is not available on Windows).

Testing on Windows. Despite its prevalence, little research targets Windows. *WinAFL* [66] has ported AFL to Windows on a best-effort basis. *Winnie* [32] improved upon this by reverse engineering Windows internals to implement a fork-server, while also designing an efficient method for fuzzing GUI applications by generating harnesses for underlying library functions. More recently, *WinFuzz* [63] introduced a kernel-agnostic snapshotting approach that enables saving and restoring the entire program state directly from within an application. For kernel fuzzing, *NtFuzz* [11] was the first approach to target the Windows kernel, using static binary analysis for type-aware fuzzing of system calls. Furthermore, *POPKORN* [28] uses taint analysis and symbolic execution, instead of fuzzing, to examine IOCTL handlers for security-critical API functions. In addition to academic works, tools like *IOCTL Fuzzer* [49] and *IoAttack* [44] exist. However, they use rather primitive approaches by modern standards.

7. Conclusion

This work introduced VMIGEN, the first scalable approach towards automated fuzzing of complex, closed-source components. Using VMI, we observe actual runtime interactions with the component under test, taking full-system snapshots of the target to capture the state, and saving informed inputs that reflect an interaction with the target under test. This process enables a fuzzer to start from a valid execution context with meaningful inputs, avoiding the inefficiencies of existing methods. Using an existing snapshot-based fuzzing framework, we demonstrate that VMIGEN significantly enhances the fuzzing process by providing the fuzzer with informed initial knowledge and context, while only requiring a limited amount of manual work. As part of our evaluation, we detected several security vulnerabilities in components of Windows and anti-virus products.

Acknowledgments

This work was supported by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA — 390781972).

References

[1] Anastasios Andronidis and Cristian Cadar. Snapfuzz: High-throughput fuzzing of network applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2022.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful Greybox Fuzzing. In *USENIX Security Symposium*, 2022.

[4] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz Driver Generation at Scale. In *Joint Meeting on Foundations of Software Engineering*, 2019.

[5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[6] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and Reflections. *IEEE Software*, 2020.

[7] Jean-Marie Borello, Julien Boutet, Jeremy Bouetard, and Yoanne Girardin. RpcView. https://www.rpcview.org/authors.html.

[8] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Joint Meeting on Foundations of Software Engineering*, 2020.

[9] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[10] Peter M Chen and Brian D Noble. When Virtual Is Better Than Real. In *USENIX Workshop on Hot Topics in Understanding Botnet*, 2001.

[11] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. NtFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[13] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for Stateful Systems: Survey and Research Directions. *ACM Computing Surveys (CSUR)*, 2024.

[14] Zhen Yu Ding and Claire Le Goues. An Empirical Study of OSS-Fuzz Bugs. In *IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021.

[15] Dmitry Vyukov and Google. Syzkaller – Kernel Fuzzer, 2015.

[16] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[17] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-Travel Testing of Android Apps. In *International Conference on Software Engineering (ICSE)*, 2020.

[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[19] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[20] James Forshaw. NtApiDotNet: A Basic Managed Library to Access NT System Calls and Objects. https://github.com/googleprojectzero/sandbox-attacksurface-analysis-tools.

[21] Open Software Foundation. DCE 1.1: Remote Procedure Call. https://publications.opengroup.org/c706.

[22] Yangchun Fu and Zhiqiang Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[23] G Data. G Data Total Security. https://www.gdata.de/total-security.

[24] Octavio Galland and Marcel Bohme. Invivo Fuzzing by Amplifying Actual Executions. In *International Conference on Software Engineering (ICSE)*, 2025.

[25] Tal Garfinkel, Mendel Rosenblum, et al. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Symposium on Network and Distributed System Security (NDSS)*, 2003.

[26] Elia Geretto, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. Snappy: Efficient fuzzing with adaptive and mutable snapshots. In *Annual Computer Security Applications Conference (ACSAC)*, 2022.

[27] Google. Chromium Docs – Sandbox. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md#Less-Privileged-App-Container-LPAC.

[28] Rajat Gupta, Lukas Patrick Dresel, Noah Spahn, Giovanni Vigna, Christopher Kruegel, and Taesoo Kim. POPKORN: Popping Windows Kernel Drivers At Scale. In *Annual Computer Security Applications Conference (ACSAC)*, 2022.

[29] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud. Virtual Machine Introspection: Techniques and Applications. In *International Conference on Availability, Reliability and Security (ARES)*, 2015.

[30] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic Fuzzer Generation. In *USENIX Security Symposium*, 2020.

[31] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[32] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[33] Kaspersky. https://www.kaspersky.com/small-to-medium-business-security/endpoint-windows.

[34] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Linux Symposium*, 2007.

[35] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[36] Kevin P Lawton. Bochs: A Portable PC Emulator For Unix/X. *Linux Journal*, 1996.

[37] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. In *ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[38] Yuwei Liu, Yanhao Wang, Tiffany Bao, Xiangkun Jia, Zheng Zhang, and Purui Su. AFGen: Whole-Function Fuzzing for Applications and Libraries. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[39] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, 2019.

[40] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. EMS: History-Driven Mutation for Coverage-based Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[41] Debasish Mandal. iofuzz. https://github.com/debasishm89/iofuzz.

[42] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 2019.

[43] McAfee. McAfee Total Protection. https://www.mcafee.com/en-us/antivirus/mcafee-total-protection.html.

[44] Microsoft. How to Perform Fuzz Tests with IoSpy and IoAttack. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-perform-fuzz-tests-with-iospy-and-ioattack.

[45] Microsoft. NtDeviceIoControlFile Function. https://learn.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntdeviceiocontrolfile.

[46] Microsoft. RPC Protocol Sequences. https://learn.microsoft.com/en-us/windows/win32/rpc/selecting-a-protocol-sequence.

[47] Microsoft. Storage Spaces overview. https://learn.microsoft.com/en-us/windows-server/storage/storage-spaces/overview.

[48] Microsoft. Windows Hypervisor Platform. https://learn.microsoft.com/de-de/virtualization/api/hypervisor-platform/hypervisor-platform.

[49] Dmytro Oleksiuk. IOCTL Fuzzer. https://github.com/Cr4sh/ioctlfuzzer.

[50] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[51] Bryan D Payne. Simplifying Virtual Machine Introspection Using LibVMI. Technical report, Sandia National Laboratories and Livermore, 2012.

[52] Jonas Pfoh, Christian Schneider, and Claudia Eckert. A Formal Model for Virtual Machine Introspection. In *ACM Workshop on Virtual Machine Security*, 2009.

[53] Process Hacker. NtAlpcSendWaitReceivePort. https://processhacker.sourceforge.io/doc/ntlpcapi_8h.html.

[54] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[55] Moritz Schloegel, Daniel Klischies, Simon Koch, David Klein, Lukas Gerlach, Malte Wessels, Leon Trampert, Martin Johns, Mathy Vanhoef, Michael Schwarz, Thorsten Holz, and Jo Van Bulck. Confusing Value with Enumeration: Studying the Use of CVEs in Academia. In *USENIX Security Symposium*, 2025.

[56] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[57] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[58] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems (EuroSys)*, 2022.

[59] Dongdong She, Abhishek Shah, and Suman Jana. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[60] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *USENIX Security Symposium*, 2020.

[61] Axel Souchet. what the fuzz. https://github.com/0vercl0k/wtf.

[62] StatCounter. Desktop Windows Version Market Share Worldwide: Dec 2023 - Dec 2024. https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/#monthly-202312-202412.

[63] Leo Stone, Rishi Ranjan, Stefan Nagy, and Matthew Hicks. No Linux, No Problem: Fast and Correct Windows Binary Fuzzing via Target-embedded Snapshotting. In *USENIX Security Symposium*, 2023.

[64] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[65] Michał Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[66] Google Project Zero. *WinAFL*, 2016.

[67] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States. In *ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[68] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.

[69] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR)*, 2022.

Appendix

Table 2 provides a detailed summary of the different kernel drivers and RPC targets tested during our evaluation. Listing 2 shows the the function prototype of the `NtAlpcSendWaitReceivePort` function.

TABLE 2: Detailed information on our evaluation targets.

	Target	**Vendor**	**Size**	**Version**
Driver	afd.sys	Microsoft	644 KB	10.0.19041.3803
	ksecdd.sys	Microsoft	148 KB	10.0.19041.3636
	spaceport.sys	Microsoft	668 KB	10.0.19041.3636
	mfehidk.sys	McAfee	928 KB	22.12.0.279
	mfencbdc.sys	McAfee	700 KB	22.12.0.472
	miniicpt.sys	G Data	920 KB	1.0.23311.36
RPC	wevtsvc.dll	Microsoft	1.8 MB	10.0.19041.3636
	localspl.dll	Microsoft	1.3 MB	10.0.19041.3636
	lsasrv.dll	Microsoft	1.7 MB	10.0.19041.3930
	schedsvc.dll	Microsoft	796 KB	10.0.19041.3636
	samsrv.dll	Microsoft	920 KB	10.0.19041.4474
	audiosrv.dll	Microsoft	1.8 MB	10.0.19041.4355

```
NTSTATUS NtAlpcSendWaitReceivePort (
  HANDLE                      PortHandle,
  ULONG                       Flags,
  PPORT_MESSAGE               SendMsg,
  PALPC_MESSAGE_ATTRIBUTES    SendMsgAttributes,
  PPORT_MESSAGE               ReceiveMsg,
  PSIZE_T                     BufferLength,
  PALPC_MESSAGE_ATTRIBUTES    ReceiveMsgAttributes,
  PLARGE_INTEGER              Timeout
);
```

Listing 2: NtAlpcSendWaitReceivePort function prototype [53]