Designing Robust API Monitoring Solutions

Daniele Cono D'Elia, Simone Nicchi, Matteo Mariani, Matteo Marini, and Federico Palmaro

Abstract—Tracing the sequence of library calls and system calls that a program makes is very helpful to characterize its interactions with the surrounding environment and, ultimately, its semantics. However, due to the entanglements of real-world software stacks, accomplishing this task can be surprisingly challenging as we take accuracy, reliability, and transparency into the equation. In this article, we identify six challenges that API monitoring solutions should overcome in order to manage these dimensions effectively and outline actionable design points for building robust API tracers that can be used even for security research. We then detail and evaluate SNIPER, an open-source API tracing system available in two variants based on dynamic binary instrumentation (for simplified in-guest deployment) and hardware-assisted virtualization (realizing the first general user-space tracer of this kind), respectively.

Index Terms—API monitoring, API hooking, anti-analysis, call interposition, binary instrumentation, hardware virtualization, malware

1 Introduction

A modern operating system (OS) typically comes with large, heterogeneous software components that developers can build on when writing a program. Consequently, to expose their functionalities the OS provides Application Programming Interfaces (APIs) that any compiled code can access through well-defined calling conventions and prototypes.

The sequence of APIs that a piece of code may invoke during its execution characterizes its externally observable behavior and, ultimately, its semantics. Therefore, monitoring API calls is a common practice for analyzing programs.

In security research, API monitoring is popular for instance in malware analysis and code reverse engineering activities in order to track how an untrusted piece of software interacts with the surrounding environment [1]. Similarly, in dependability research API monitoring is useful for, e.g., run-time monitoring [2] and troubleshooting [3] tasks.

Researchers have often explored specialized forms of API monitoring, tailoring the implementations to different contexts. For instance, malware sandboxes interpose most of the time on system calls [4] to capture the events of interest for sketching the behavior of a sample. Thanks to this design, sandboxes can collect in a single spot the events originating from the many alternative library-level APIs that a sample can use (sometimes in unexpected or undocumented ways) from the upper layers of the software stack. Several academic works [5], [6], [7] have then explored how to implement system call interposition in increasingly covert and efficient ways, benefiting from the hardware virtualization extensions available in modern CPUs.

However, whenever a notable sample calls for manual intervention for its dissection [8], malware analysts often resort to monitoring solutions capable of tracing first and foremost library calls. Such fine-grained information can be very useful to understand how a sample achieves some functionality. Consider, for instance, an attempt to connect to an HTTP server via some OS feature (e.g., `WinHttpConnect`): intercepting a generic TCP packet transmission event down in the software stack may not be as informative as logging the API call that originated it. Furthermore, for several APIs, the sole observation of system call events would not be sufficient to infer their use [9].

Tracing high-level facts from library code is in general valuable for many other tasks involving monitoring, troubleshooting, and reverse engineering of programs. Apparently though, prior literature seems to have given little attention to the problem of designing robust API monitoring solutions, especially for the *accuracy*, *reliability*, and *transparency* dimensions that characterize the tracing process.

This gap seems to be reflected in how current mainstream API monitoring systems fall short in one or more of these three respects. Common flaws that we observed are an incomplete symbol and argument tracing, missed calls to dynamically solved APIs, and instrumentation artifacts that are easy prey of adversaries [10]. Furthermore, most systems overwhelm users with calls that originate within the implementation of a high-level API, since they recursively log all the APIs that its code invokes for its working. Unfortunately, logging such calls does not bring users any actionable information but only reflects how the OS implements some functionality.

Motivated by these observations, in the first part of this article we identify six key challenges along the way towards accurate, reliable, and transparent API monitoring solutions. We analyze the problem space and discuss a general-purpose design that can address such challenges and work with different instrumentation technologies. We then present our SNIPER system, detailing common traits and distinctive features of two implementation variants.

The first variant builds on dynamic binary instrumentation [11]. It can operate as a standalone in-guest agent or

- *Daniele Cono D'Elia, Simone Nicchi, Matteo Mariani, and Matteo Marini are with the Sapienza University of Rome, Rome, RM 00185, Italy. E-mail: delia@diag.uniroma1.it, simone.nicchi@gmail.com, {mariani.1815188, marini.1739106}@studenti.uniroma1.it.*
- *Federico Palmaro is with Prisma S.r.l., Rome, RM 00185, Italy. E-mail: f.palmaro@prismaprogetti.it.*

Manuscript received 11 March 2021; revised 1 December 2021; accepted 2 December 2021. Date of publication 10 December 2021; date of current version 16 January 2023.
(Corresponding author: Daniele Cono D'Elia.)
Digital Object Identifier no. 10.1109/TDSC.2021.3133729

work as an extension for the many program analysis systems from the literature that make use of this technology.

The second variant builds on hardware virtualization extensions. While much literature used them for system call tracing [5], [7], the article will detail how we automatically locate and instrument an arbitrary selection of library functions and how we filter the events belonging to the monitored program or an execution flow derived from it.

SNIPER targets the Windows platform, covering a large collection of libraries and system calls, and can be extended to other systems. We evaluate its capabilities using API testing suites, real-world programs, and complex malware.

While our implementations can work in different contexts, we incubated them as part of our malware analysis research: the tricky patterns found in this realm, combined with quirks of Windows internals, have been a tough training ground for their development. The code is available at:

`https://github.com/dcdelia/sniper`

2 Background

This section covers the main traits and underpinnings of the API handling process for Windows programs and libraries; then, it compares instrumentation technologies available to date for implementing an API monitoring system.

2.1 Windows API Resolution and Internals

Windows programs can access functionalities of the surrounding software environment by *importing* API functions from *dynamic-link library* (DLL) modules. To this end, the Portable Executable format provides for an `.idata` section for listing any required external API symbols. The section contains an array called *import address table* (IAT) that the Windows loader will populate at run-time with pointers to the desired functions, importing them from known DLL modules. The program can then call such an API function by simply retrieving its address from the IAT.

DLL modules come with an `.edata` section for their public functions and storage, commonly known as *exports*. Each *export address table* (EAT) entry hosts a relative virtual address (RVA), i.e., the offset of the export from the beginning of the module. For an executable importing a function from a DLL, the loader will typically populate the involved IAT entry by looking up the RVA of the export in the EAT of the DLL and adding it to the base address that the system chose for the DLL when loading it.

However, there are alternative methods to locate API addresses. A program may manually load a DLL and retrieve its exports using the `GetProcAddress` Windows API that does not touch the IAT. Furthermore, regardless of automatic or manual DLL loading, a program may covertly solve symbols by manually navigating the code modules in memory and parsing their `.edata` section. As we will discuss in Section 3.3.1, this behavior can be found in malware and in programs shielded with executable protectors.

When it comes to the internal structure of a DLL, an exported API can be of different kinds. The recurrent case is when its logic is entirely contained in the code starting at the given RVA. For other APIs, the code may be partial, ending with a tail jump to another function (private or exported) in the module or imported from another DLL[1]. Finally, in other cases, the RVA does not point to code but represents a *forwarder* export [12]: this instructs the loader to silently rewire any IAT entry referencing it to point to another export from another DLL. Due to these factors, determining for an export what are the "exit points" to the caller function is therefore not always immediate.

As for API prototype information, the Windows SDK provides header files that list for each API the calling convention [13] (typically *stdcall*) and the *input modifier* of each argument, that is, when it identifies an input (IN) or an output (OUT) value, or both (INOUT) [14]. In practice, for an output argument the caller provides a pointer to a buffer where the API can write data. Finally, headers also define a large number of primitive types and data structures.

System calls (or *syscalls* for short) operate differently. Programs normally access them through user-space wrappers accessible as exports of `ntdll.dll`. Each wrapper writes in the EAX register the ordinal corresponding to the syscall and triggers a software interrupt. However, adversaries can elude the monitoring of such wrappers by identifying the ordinals for the current Windows version (e.g., by parsing `ntdll.dll`) and then triggering an interrupt with a custom assembly stub. This technique is known as *direct syscall*. Furthermore, experienced writers may use undocumented syscalls to complicate the analysis, as their prototypes may be available only in reverse engineering forums.

2.2 Instrumentation Technologies

As we will see throughout the article, the type of instrumentation that a system uses to interpose on API calls affects several dimensions of the overall monitoring efficacy.

A possible avenue is to patch the in-memory image of the program under analysis or the libraries of interest. For instance, in the approach known as *IAT hooking*, one can overwrite an IAT entry to make it point to a tracing stub that logs the call and then invokes the intended API. A more reliable alternative is to instrument library code, modifying the prologue of monitored functions with the insertion of a *trampoline* (also known as *inline hook*) to a tracing stub.

Mainstream commercial API tracers follow either approach. As we will also in Section 4.5, a major weakness of both schemes is that an adversary can easily recognize the modifications they introduce [10]. Generally speaking, instrumentation artifacts are a well-known problem for dynamic analyses that operate through patching or rewriting: for this reason, other instrumentation technologies have gained popularity in security research [11].

Dynamic binary translation (DBT) systems can trap execution at arbitrary instructions based on their type (e.g., control transfer instructions) or address while providing the running code with the illusion that instrumentation is not present. A popular DBT technique for user-space monitoring of programs is *dynamic binary instrumentation* (DBI) [11], [15], [16]. When a dynamic analysis has to deal with kernel-level or system-wide execution flows, researchers have used instead whole-system emulators like QEMU [17] as a DBT engine to instrument an entire virtual machine (VM).

[1] We observed this pattern, for instance, with APIs partially implemented in `kernel32.dll` that rely in turn on `kernelbase.dll`.

In such out-of-VM scenarios, *virtual machine introspection* (VMI) tools come into play to overcome the semantic gap [18] from having to retrieve high-level features of the underlying OS and processes by accessing the raw memory of the VM.

The advent of CPU virtualization extensions like Intel VT favored new instrumentation primitives with better performance and transparency. By maintaining a split view of code and data pages in the Extended Page Table, the authors of SPIDER [6] create *invisible breakpoints* for registering analysis callbacks at physical page addresses. This design defeats introspective attacks: to insert a breakpoint, SPIDER creates a code page for instruction fetching and modifies it, leaving the original code untouched in a data page visible to read and write operations. The DRAKVUF [7] sandbox uses a variant of this mechanism to hook kernel code for syscalls and few selected user-space APIs from DLLs. However, as we will explain later in the article, lazy loading mechanisms and other OS entanglements get in the way when one wishes to use VT-based instrumentation to trace arbitrary user-space APIs from heterogeneous libraries.

3 Design Space of API Monitoring Systems

In this section, we identify and analyze common problems of API monitoring systems, discussing design alternatives that are not bound to specific instrumentation technologies. The possibilities that we outline are readily actionable, as Section 4 will detail for our DBI and VT-based implementations.

3.1 Challenges in API Monitoring

We identified six challenges along the way to accurate, reliable, and transparent API monitoring solutions:

[C1.] **Transparency**. Adding probes or other instrumentation for intercepting calls to an API may introduce artifacts that an adversary can look for [1]. Also, they may collide with recently introduced OS mitigations against API hijacking in exploits and malware [19].

[C2.] **Recall**. The points in the software stack where instrumentation takes place also determine how many of the actual calls a tracer can capture.

[C3.] **Coverage**. Tracing argument values for an API call is more informative than its function name alone. With an ample universe of libraries, a programmatic approach to extract prototypes and data type declarations can avoid incomplete information retrieval.

[C4.] **Output values**. A tracer should capture the return value of an API call and any data that the API wrote to output locations supplied by the caller.

[C5.] **Relevant calls**. An API call from program code may lead to many intra- and inter-component API calls down the software stack. These *internal calls* bloat monitoring logs but are hardly informative: therefore, a tracer should filter them out.

[C6.] **Derived flows**. A tracer should cover derived execution flows like child processes and remote threads (i.e., created in other processes). Adversaries may also use such flows to hide API calls from the analysis [20].

TABLE 1
Mainstream Tools and Research Systems for API Monitoring

		C1						
Tracing system	**Technology**	**Test T1**	**Test T2**	**C2**	**C3**	**C4**	**C5**	**C6**
API Monitor v2α-r13	IAT hooking	✗	✓	◔	◕	●	○	◕
SpyStudio v2.9.2	Trampolines	✓	✗	◕	◔	●	○	◔
WinAPIOverride 6.6.6	Trampolines	✓	✗	◕	◕	●	◕	◔
drltrace [31]	DBI	✓	✓	◕	◔	○	◕	◔
PyREBox [32]	QEMU-TCG	✓	✓	●	●	●	○	◕
SNIPER	DBI, VT	✓	✓	●	●	●	●	●

Circles are filled to indicate if an aspect is met to a basic, good, or optimal extent.

We observe that **[C1]** and **[C6]** are compelling aspects in many security and dependability settings; **[C2-4]** impact the soundness and completeness of high-level analyses that require API monitoring; finally, **[C5]** affects the temporal and spatial overhead of a monitoring system as well as its usability when a human agent is involved.

We reviewed publicly documented API monitoring solutions and identified[2] three commercial products popular among security professionals and two DBT-based research systems. After careful analysis and testing, we found all of them to fall short in one or more of the six **[C1-6]** respects (Table 1). We defer the discussion of each system to Section 4.5 to allow for a detailed comparison with our solutions.

To come up with robust solutions for API monitoring, in this article we reason on the key design choices that impact these respects. The coming sections will discuss possible alternatives (where applicable) and motivate our choices. Also, one of our goals is to pursue the accuracy and reliability of the tracing process without tying our design to an instrumentation technology and its transparency properties.

3.1.1 *Threat Model*

As we devote special attention to security uses, we assume that the program under analysis may run introspective sequences to reveal API monitoring mechanisms. These sequences may verify the integrity of its code and data (Test T1) or of DLL code in memory by using pre-computed signatures or reading its counterpart on disk (Test T2). We craft an adversarial program for testing purposes **[C1]** that:

- for T1, it compares each IAT entry of the running program with the expected address for its symbol (found through the EAT of the DLL exporting it);
- for T2, it reads the DLL from the disk, applies relocations to its contents (to match where Windows loaded the original DLL), and compares the first 8 code bytes of every imported DLL symbol with our copy.

Eventually, T1 will expose stubs for IAT hooking, while T2 will reveal hooking trampolines in API code prologues. From a technical standpoint, these tests closely resemble techniques used in defensive literature to detect hooks that rootkits and other malware use to alter API results [21].

3.2 Scope of Monitoring

We find that two elements help determine the scope of an API tracer: the breadth and level of detail of API prototype

[2] We involved in the selection an independent malware analyst with ten years' experience as a principal malware analyst in security firms.

information available to it and the treatment of internal calls.

3.2.1 Prototypes

Windows offers an extensive collection of DLLs to programmers. Each DLL file comes with information only for its exported symbols and their relative locations (Section 2.1). A sound way to obtain information on their arguments **[C3-4]** is to cross-reference DLL symbols with function declarations from the header files of the Windows SDK[3]: the Deviare system [22] offers an infrastructure to this end.

This approach is general and can also apply to third-party libraries if their headers are available. Note that a programmatic extraction shall include the size of each argument so as to be able to fetch their values at run-time; also, pointer types require a recursive valuation. Input modifiers are not present in header files and should be retrieved from the MSDN documentation (e.g., using a crawler [23]).

3.2.2 Relevant Calls

We reasoned on what makes a traced call informative for a user **[C5]**. Internal calls happening within one or more DLLs do not provide valuable insights to the user but only describe how the OS implements any API from the upper layers that exercises them.

Eventually, internal calls make up a large fraction of the logs and drain resources for their tracing. To spot them, we define the scope of a call to be *relevant* only when the call is made in code belonging to the program under analysis and eventually returns to it.

This definition rules out internal calls and jumps to other exports and redirections within API code (Section 2.1); also, it captures syscalls that programs invoke directly. For derived flows **[C6]**, by “program under analysis” we mean the process where the code first runs, the child processes and remote threads it creates, and any recursive byproduct.

3.3 Hook Insertion

As part of its working, an API monitoring system must interpose on specific events: the invocation of an API (*API entry*) and the moment it returns to its caller (*API exit*). The placement of hooks through instrumentation impacts the recall of a tracer **[C2]** and its call argument extraction **[C3,4]**.

3.3.1 API Entry Events

In principle, to interpose on API entry events, one may target either the means for a program to reach external symbols or the code of the corresponding functions.

Locating with a static analysis the code places where a program may make an API call faces known soundness and completeness issues (e.g., with code pointers) that obfuscation and other anti-analysis measures make only worse [1].

One possibility is to interpose on control transfer instructions dynamically. For instance, PyREBox captures every `call` (optionally `jmp` and `ret` too) instruction in the execution and checks its target against a list of API addresses.

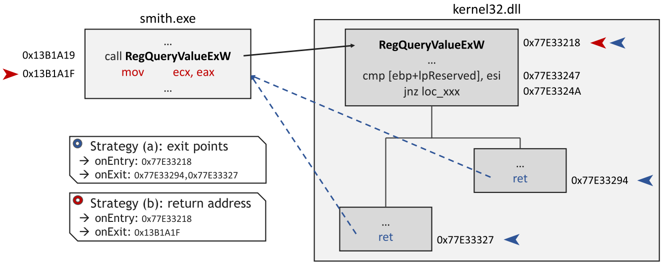

Fig. 1. API call handling with strategy (a) or (b) for exit events. Arrows placed next to instruction addresses represent hooks.

Unfortunately, this approach introduces unnecessary overhead for control transfers unrelated to APIs, which are dominant in practice. It also requires an instrumentation facility that can hook instructions by type as the CPU sees them during fetching, ruling out static rewriting (self-modifying code breaks it [11]) and VT-based instrumentation.

As we anticipated in Section 2.2, IAT hooking adds instead one level of indirection to API calls done through the IAT of a module by redirecting them to an analysis stub. Typically, a distinct stub is required for each function of interest in order to track the intended target. In a tracing scenario, the stub logs any input arguments from the caller, issues a control transfer to the desired API, and eventually logs any output arguments before returning to the caller. While this strategy also helps in ignoring internal calls, it captures only calls that a program makes using the IAT. Programs, however, can resort to dynamic symbol resolution through the `GetProcAddress` API for many reasons[4]. Even worse, they may use custom implementations of `GetProcAddress` that go unnoticed by monitoring tools. Such sequences are common among covert malware and software shielded with packers and executable protectors (which are a popular choice for both malicious and benevolent programs [1]) and are a standard feature in shellcode and other injected payloads [24]. Furthermore, an adversary may use a custom IAT-like lookup table for API addresses during compilation and populate it covertly when the program starts.

We believe that the only reliable and efficient choice to capture all the API entry events of interest is to intercept when execution touches code belonging to APIs. Therefore, in the design of a tracer, we argue for placing instrumentation in the prologue of API implementations: interposition will happen only and every time an API function is invoked, either by the program or internally down the software stack.

To locate addresses for probe insertion, we target every unique RVA that appears in the EAT of a loaded DLL and is not a forwarder. For forwarders, we postpone the instrumentation until we process the APIs that serve the forwarded calls. This design maximizes recall **[C2]**.

3.3.2 API Exit Events

Intercepting when the execution returns to the caller of an API is necessary to log return values and output arguments **[C4]**. Fig. 1 depicts two viable options to this end:

[3] And hand-written headers for uncounted structures and syscalls. Those may be found in reverse engineering forums and in ReactOS.

[4] We can name, among others, locating symbols in a library loaded at run-time with `LoadLibrary`, checking if an API is available in the current Windows version before using it, and accessing low-level `ntdll.dll` functions that are not available in common header files.

(a) at DLL load time, placing hooks on the single or multiple exit points for the code that implements the API;

(b) on API entry, placing a hook on the instruction that represents the return address for the call.

Strategy (a) of chasing exit instructions is not immediate to pursue due to the redirections present in many API implementations (Section 2.1). Based on the insights from analyzing many DLLs, we wrote a static analysis that processes partial implementation stubs and tail calls in APIs to determine their exit points. We found exotic cases where an export makes a tail call to an export from another DLL that eventually leads to a tail call to another export from a third DLL.

Strategy (b) of chasing return addresses looks easier at first, but the hooking logic should be carefully designed to process only real API exits. In fact, the instruction following the call may be a join point in the control flow graph later reached from basic blocks that do not end with an API call. We found many instances of this pattern in our tests (we provide an example in the supplementary material Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2021.3133729).

Unless a program contains such instances in a pathological number, strategy (b) is more attractive as it brings fewer invocations of the analysis callback for API exit events since our design discards internal calls. Instead, strategy (a) would trigger the callback even for such latter calls, requiring the analysis code to ignore them. Choosing one scheme over the other has no impact on other design components but depends mainly on the capabilities and efficiency of the instrumentation technology in use: we detail this aspect in Section 4.3.2, explaining why and when the two may also profitably coexist when implementing an API tracer.

3.3.3 *Argument Extraction*

For retrieving input and output arguments of API calls, the instrumentation facility should grant the tracer access to CPU register values and program memory.

Upon API entry events, the stack pointer value suffices to locate the arguments under 32-bit *stdcall* (Windows) and *cdecl* (UNIX-like systems) calling conventions. 64-bit code requires accessing dedicated registers for the first 4 arguments, then any additional one is passed on the stack. Similar considerations apply to floating-point arguments, which we omit in the remainder of the article for brevity.

For exit events, the return value is available in register EAX, plus EDX for wider data types. The tracer can save at API-entry time any pointer values for 64-bit output arguments passed via registers, which would otherwise be lost. We also remark that prototype information is essential on API entry and exit events for computing offsets for stack arguments by taking into account their size and order.

4 Our Tracing Solutions

Fig. 2 portrays the architecture of SNIPER, which embodies the design points that emerged from the discussions of Section 3. We detail next its DBI and VT-based implementations in their common traits and distinctive features.

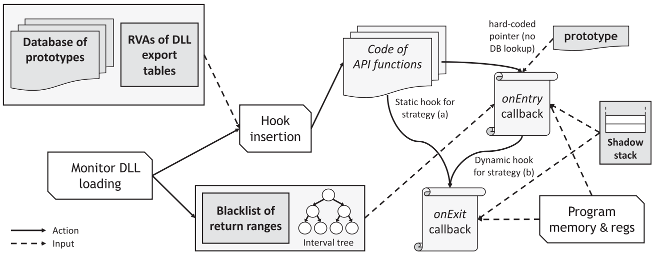

Fig. 2. Bird's eye-view of the proposed SNIPER system.

4.1 Instrumentation Technologies

The design options outlined throughout Section 3 are general: therefore, one can implement them using different instrumentation technologies. However, choosing one technology over others can lead to transparency concerns **[C1]**.

We implement the first variant of SNIPER in Pin [16], a popular DBI choice in programming language, software testing, and security research. The DBI abstraction behind it ensures that every address or value manipulated by the program matches the one expected in a native execution. Under the hood, Pin operates by JIT-compiling and instrumenting code in a designated cache area: any hook we insert will not be visible to introspective attempts from the threat model (Section 3.1.1) of **[C1]**. We then shield Pin for DBI artifacts by using the mitigations of [11]. Ultimately, these factors contribute to making our tracer less conspicuous than commodity systems based on binary patching.

The second variant brings a new piece to the research landscape: an API tracer compatible with modern designs for efficient out-of-VM analyses via VT extensions. For placing hooks, we build on the implementation of invisible breakpoints from DRAKVUF [7], a whole-system analysis framework based on the Xen hypervisor. As we detail next, we extend it with a VMI component to restrict monitoring to selected processes and identify their derived flows.

Our implementation variants can serve complementary tasks and provide different trade-offs in terms of ease of deployment, breadth of monitoring, and transparency. The DBI variant ships as a high-level library suitable either for standalone usage as an in-guest API monitoring tool (like current mainstream API tracers) or for being plugged in existing analysis systems based on Pin (which are numerous in security research [11] as well as in other communities). The VT-based variant is particularly suitable for scenarios where minimal invasiveness is desirable (e.g., with code sensitive to environment artifacts [7] or slowdowns [25]) and for monitoring groups of unrelated processes and system-wide flows. However, it must execute in a modified hypervisor.

4.2 Relevant Calls and Execution Units

In Section 3.2 we mentioned the advantages of restricting tracing to calls explicitly made by the program under analysis **[C5]**. In this section, we first describe how to identify relevant processes and threads **[C6]**—a task that requires different provisions for different instrumentation technologies—and then move to the filtering policies for internal calls.

Pin naturally operates on a single process but offers APIs to intercept the creation of child processes and of a remote thread in an existing process [11]. We use such APIs to extend the instrumentation to derived flows automatically.

In the VT-based scenario, the subject of instrumentation becomes the entire system. While libraries belong to the private virtual address space of a process, the OS can share physical pages for DLL modules among multiple processes as long as their contents do not change. Therefore, an invisible breakpoint in API code may also be triggered by processes unrelated to the one(s) of interest.

To identify relevant execution units, we wrote a VMI component that, starting from a process under analysis, tracks the creation of child processes and remote threads recursively. The component maintains a pool of IDs for monitored threads and their enclosing processes. To this end, we hook the kernel code of syscalls `NtCreateThreadEx` and `NtTerminateThread` when they return to user mode. For thread creation, we read the `ThreadHandle` output argument of the syscall: since its value is valid only for the caller, we look into the Windows Object Manager to locate the referred `_ETHREAD` data structure and extract from there the process ID (different than the caller's when dealing with remote threads) and the thread ID. For thread termination, we discriminate whether the caller thread is terminating itself or another thread, accessing the related `_ETHREAD` data in the latter case.

With those IDs, we maintain and update the pool of relevant units. Whenever execution hits an invisible breakpoint from an API hook, the analysis callback checks whether the current execution unit belongs to the pool. In the case of code injection patterns, this design also allows us to ignore activities from the "authentic" threads of a victim process.

To filter out internal calls happening in Windows components **[C5]**, we then check, under any instrumentation, in which region the return address of an API call falls.

We observe that a whitelisting policy that logs only calls that return to code regions belonging to the program can be a slippery road. Not only malware and protected executables, but even COTS programs can exercise exotic behaviors such as executing code from the heap or changing section permissions [25].

We find it safer to build a *blacklist of return ranges* for calls to discard, populating it with code section ranges of Windows DLLs (such addresses would indicate an internal call) as those get loaded and unloaded. This scheme turns out to be robust and efficient: as intervals are disjoint, we use an interval tree with a lookup cost that is logarithmic in the number of Windows DLLs in use. We then complement the range lookup operation with ad-hoc measures for tail jumps (Section 2.1) that we present in the next section.

4.3 Hook Insertion and Callbacks

This section details the registration and functioning of analysis callbacks that we use to log API calls from program code along with their input and output values.

4.3.1 API Entry

For the reasons discussed in Section 3.3.1, we target the RVAs of the function symbols exported by Windows DLLs to place the hooks needed to interpose on API entry events.

DBI engines offer facilities to intercept loader activities. Once we identify the base address of a DLL module of interest, we locate its EAT, cross-reference the names of exported functions with a database of prototypes[5], and compute the run-time addresses of functions using their RVAs. We instrument the first instruction in each function with *onEntry* analysis callback and hard-code the address of the API prototype information as its argument, thus avoiding an expensive lookup at run-time. This approach is independent of the Windows version in use and has performance advantages as we can use Pin's IMAGE mode to place efficient ahead-of-time instrumentation [26].

In the VT-based scenario, we can equally parse EATs for RVAs or load Rekall profiles [27] precomputed for the current Windows version. In a similar way to the DBI case, we insert an invisible breakpoint at the first instruction in each function, associating to it an *onEntry* analysis callback with the address of its prototype information as hard-coded argument. The difference with the other variant is that the callback will first determine whether the intercepted thread belongs to the pool of units that we wish to monitor.

Following the approach of SPIDER [6], an invisible breakpoint (Section 2.2) takes the form of an `int 3` sequence replacing the original instruction: when hit, the hypervisor intercepts the raised #BP exception, invokes the callback, executes the original instruction in single-step, and re-enables the breakpoint. As we anticipated in Section 2.2, de-synchronization of data and code views for pages hides the presence of such a breakpoint from an adversary.

Invisible breakpoints operate on physical pages: adding instrumentation to logical addresses requires their translation to physical ones. Unfortunately, when we intercept the loading of a DLL, due to the *lazy loading* mechanism of Windows, not all the DLL pages may be amenable to hook insertion when we intercept its loading: put in other words, for a logical address there might be no physical page yet [28]. To mitigate this issue, we wrote a component that loads DLLs of interest in a separate process and reads code from their sections, forcing page materialization: since such pages are normally shared among processes, we can place hooks also for the program of interest.

This scheme still misses a few corner cases, but luckily other researchers concurrently developed a mechanism [9] to force page faults and materialize pages upon DLL loading [29], using a new feature of libvmi [30] that meanwhile became available. We have started to extend our implementation to integrate their technique.

onEntry Callback: This analysis routine takes as input the ESP register value (to access the return address and stack arguments), a pointer to the prototype information for the current API, and other register values where needed (e.g., with 64-bit code). Its simplified pseudocode provided in Fig. 3 is agnostic to the instrumentation technology.

The *onEntry* callback maintains a thread-local *shadow stack* of currently monitored functions[6]. Line 1 restricts

[5] The authors of PyREBox released a large one that we use and refine in a few aspects, e.g., to distinguish INOUT arguments from OUT ones.

[6] We say *functions* as in a thread the concurrently active functions that return to user code may be multiple: this happens, for instance, when a program loads a custom DLL (hence we equate it with program code that we seek to monitor) with *LoadLibrary* and its *DllEntryPoint* function invokes one or more APIs before *LoadLibrary* returns.

logging to calls made from program code, otherwise we would be mirroring also API calls happening within DLLs. Line 5 discards internal jumps and tail calls to other exported functions, which would see the same return address from program code of their caller (current top stack entry). Line 6 deals with hooking the return address when we use strategy (b) for handling API exit events. Line 7 sanitizes stale stack frames in case of instrumentation glitches: if ESP is at higher addresses than the ones stored, those calls have likely returned already since the stack grows downwards to lower addresses. Finally, lines 8 and 9 update the shadow stack and log the call, respectively.

4.3.2 API Exit Events

In Section 3.3.2, we presented two strategies for tracing API returns: hooking exit instructions (a) or return addresses (b). For strategy (a), we place hooks at DLL load time at the exit points identified with static analysis, while for (b) we place them dynamically upon API entry events (line 6).

We implement strategy (a) in Pin using the IMAGE mode as we did with entry events. For strategy (b), Pin lacks a neat way to place callbacks on instruction addresses during execution without resorting to heavy-duty features like TRACE mode (while its ROUTINE mode is unreliable for exit events [26]). We instead build on the recently introduced *PIN_RemoveInstrumentationInRange* primitive to force Pin to recompile and reanalyze only the instruction located at the return address. Recompilation is required only the first time a given return address occurs at line 6. As we will explain shortly, the *onExit* callback ignores subsequent spurious raises of the hook, so we are not forced to unregister the callback once the handling of an exit event completes.

In the VT-based scenario, invisible breakpoints naturally back both strategies, as they can target arbitrary addresses.

On a different note, in Section 3.3.2 we mentioned that strategy (b) reduces the fraction of times *onExit* is invoked for uninteresting events that must be discarded. However, there could be cases where strategy (b) may be more intrusive for the program under analysis (e.g., due to recompilation events in Pin), or an adversary knowing the details of the system may tamper with return addresses on the stack. SNIPER retains support for both schemes, which can coexist seamlessly and profitably.

In more detail, let us consider a DLL or function for which we did not precompute exit points: we can still instrument the return addresses for them and use the other scheme for the rest of the APIs. Similarly, this flexibility helped us when developing the VT-based version: DRAKVUF failed the breakpoint insertion at some exit instructions with no apparent reason, but we could fall back to hooking the return addresses for the related APIs.

```
function onEntry(threadID, ESP, prototype, ...):
1  if *ESP ∈ RangeBlacklist then return
2  SStack = getTLS(threadID) // thread-local storage
3  if not SStack.empty() then
4      cInfo = SStack.peek() // recorded call information
5      if *ESP == cInfo.ra && ESP == cInfo.esp then return
6  hookReturnAddress(*ESP) // skip under strategy (a)
7  removeStaleEntries(SStack, ESP) // from top while cInfo.esp ≤ ESP
8  SStack.push(<*ESP, ESP, prototype>) // <ra, esp, proto> call info
9  parseArgsOnEntry(ESP, prototype, ...)

function onExit(threadID, ESP, EIP, EAX, ...):
10 SStack = getTLS(threadID)
11 if SStack.empty() then return
12 idx = SStack.size() - 1
13 cInfo = SStack[idx]
14 while true do
15     if cInfo.ra == EIP || --idx < 0 then break
16     cInfo = SStack[idx]
17 if idx == -1 then return
18 if ESP == cInfo.esp + 4 + cInfo.prototype.retN then
19     parseArgsOnExit(cInfo.esp, cInfo.prototype, EAX, ...)
20     SStack.resize(idx) // pops one or more elements
```

Fig. 3. Analysis callbacks executed upon API entry and exit events.

onExit Callback: The pseudocode in Fig. 3 of the analysis routine for exit events is instrumentation-agnostic and embodies strategy (b). Initially, the routine looks up the most recent shadow stack entry that matches the instruction pointer EIP, which corresponds to a return address under this strategy. It then invokes a routine for processing the return value and output arguments. The routine takes the ESP value and the prototype stored in the shadow stack during the entry event for the call (and any saved output arguments passed via registers for 64-bit code), along with registers EAX and (where needed) EDX for the return value.

The checks at lines 11 and 17 intercept when a previously hooked return address is reached by blocks that did not make an API call (Section 3.3.2, Appendix A, available in the online supplementary material). This logic is semantically equivalent to disabling instrumentation at the return address, which may not always be cheap (e.g., requires extra recompilations in Pin). A sanity check at line 18 compares the current ESP value against the one stored by *onEntry* for the frame, "undoing" the effects of the `ret N` instruction[7]. In practice, we observed that simply checking for $ESP > cinfo.esp$ is a reliable approximation of the condition.

For strategy (a), since the callback would trigger on an exit point, we would see that the instruction pointer EIP has not been diverted yet to the return address (which can be found however at *ESP) and that the stack pointer has not been adjusted with the displacement associated with the return sequence. We adapt the code for *onExit* from Fig. 3 to this strategy by replacing EIP with *ESP at line 15 and the condition at line 18 with simply $ESP == cinfo.esp$.

4.3.3 Argument Extraction

The subroutines that *onEntry* and *onExit* call at lines 9 and 19, respectively, are conceptually similar.

Both may have to locate data from the stack, computing offsets based on the size of each previous argument in the prototype. Logging primitive types is a straightforward task. With pointers, we need to distinguish the type and size of a pointed object. Even before, we should verify whether a pointer is meaningful—that is, if it points to valid data.

[7] After the instruction, the stack pointer value will be higher than the ESP value seen by *onEntry* by r+N bytes: r is 4 for 32-bit code and 8 for 64-bit code, while N is the stack space used for argument passing for *stdcall* APIs (we remind our readers that in *stdcall* functions the callee has to clean the stack for the caller) and zero for *cdecl* ones.

Ideally, a sound way would be to take into account the API semantics (e.g., checking its return code to discriminate errors), but this may be unrealistic for a general-purpose tracer. Instead, we can check whether the pointed object falls into valid memory and then call a print helper for its contents. This operation is immediate for fixed-size objects such as primitive types or structures. For variable-size objects like strings, we cannot rely on the presence of some terminator when an API fails: we conservatively fetch a predefined amount of bytes from the address, reducing it if the chunk would span two pages with the second being invalid.

4.4 System Calls

In the presentation flow, we postponed the discussion of how our implementations address syscalls because there are some unique aspects to their handling.

From the program's perspective, syscalls are self-contained: they happen over a privilege mode change (to kernel mode when invoked, back to user mode upon termination). Hence, no shadow stack update is needed.

A good source of prototype information is the database from the drsyscall module of DynamoRIO [15]. We build on it as it covers many undocumented syscalls and provides auxiliary data needed to determine an argument type for cases where the latter depends on another syscall argument.

For syscall instrumentation, a DBT system naturally intercepts the privileged sequences involved in their invocation and return [16]. In Pin, we register two callback routines for syscall entry and exit events: from there, we extract the syscall ordinal, use it as array index to retrieve the corresponding prototype from the drsyscall database, and extract the current syscall arguments.

In the VT-based scenario, as we cannot interpose on instructions by type, we follow the design proposed by the authors of DRAKVUF in [7]—that is, we instrument the entry and exit instructions of the syscall implementations found in the code of the Windows NT kernel.

Finally, we deem a syscall relevant if it returns either to program code directly (as with direct syscalls found in malware) or to some *Nt* library wrapper from `ntdll.dll` called in turn from program code. For the latter case, we walk back the stack mimicking the effects of the epilogue instructions of the wrapper: we check if the stack frame of the method returns to an address in the blacklist of return ranges and discard the call accordingly.

4.5 Comparison With Other API Tracing Systems

Having completed the presentation of our tracing solutions, we can now discuss the qualitative assessment of existing systems given in Table 1. In Section 5.1, we will substantiate our claims further with figures from an experimental comparison on heterogeneous benchmarks.

The first three entries represent commodity monitoring systems. Some evident limitations are present in all of them: the introduction of classic artifacts **[C1]**, the inability to intercept direct syscalls **[C2]**, and the incomplete handling of derived flows **[C6]**. Each system then suffers from additional limitations that we describe next. On the positive side, all of them can trace output arguments **[C4]** as they capture API exit events by issuing API calls from logging stubs.

API Monitor applies IAT hooking to the main program module and also to libraries, which—besides Windows APIs—may host third-party code or other application components. To deal with dynamically solved APIs **[C2]**, API Monitor rewrites the output of `GetProcAddress` invocations so that the caller will receive the address of a logging stub instead of the actual API. This strategy, however, is not only trivial to detect (with the stub even placed in a different code module) but is already defeated by the standard means for covert API resolution that we discussed in Section 3.3.1. With ~15K DLL functions supported, API Monitor has one of the richest sets of API prototypes **[C3]** among all the tested systems. Discarding internal calls **[C5]** is left to the user upon termination of the logging. Finally, API Monitor automatically follows child processes and injected threads **[C6]**. However, it traces the own activities of the injected process too and, above all, misses all the APIs solved with covert methods routinely used in injected payloads.

SpyStudio adds trampolines to API code **[C2]** using the Deviare In-Proc hooking library (we will discuss it in Section 7). In our understanding[8], SpyStudio predominantly targets DLL APIs of common usage, resulting in a coverage of prototypes more limited than in other systems **[C3]**. As with API Monitor, discarding internal calls **[C5]** is left to the user upon termination. Finally, SpyStudio automatically follows child processes but misses remote threads **[C6]**.

WinAPIOverride works by adding trampolines to API code too **[C2]**, with a prototype collection of ~7.7K DLL APIs **[C3]**. To cope with internal calls **[C5]**, the user can enable an online whitelisting filter to track only calls from the main module of an executable (we discussed common pitfalls of this strategy in Section 4.2) or specify custom inclusion or exclusion lists for specific modules. To cope with derived flows **[C6]**, they can add rules to extend the monitoring to other processes **[C6]** (e.g., by their name). WinAPIOverride also offers orthogonal capabilities for process manipulation that may be useful for binary reverse engineering tasks.

Both the DBT-based systems of Table 1 do well with classic artifacts **[C1]** thanks to the instrumentation technology in use: drltrace builds on the DynamoRIO DBI engine[9], while PyREBox uses QEMU-TCG for whole-system emulation.

drltrace places instrumentation on API prologues. Its authors left syscalls out **[C2]** since DynamoRIO offers the drsyscall component to this end. drltrace comes with much fewer prototypes than most systems **[C3]** (~2K APIs) and does not trace return values and output arguments **[C4]**. It has automatic filtering capabilities for internal calls by whitelisting the text and heap regions of the main module but does not expunge tail transfers from internal calls **[C5]**. Finally, its support for derived flows is limited as it can automatically follow only child processes **[C6]**.

[8] Through reverse engineering work on its main executable, we identified controls for tracing ~700 functions. We believe that more may still be added by its developers (or enabled in ways that we did not foresee in the GUI) using the Deviare2 DB that comes with the bundle.

[9] The instrumentation mechanism of drltrace passes Test T2 for all but a few APIs: the test incidentally reveals artifacts of DynamoRIO, which alters several `kernel32.dll` APIs for its working [34].

PyREBox interposes on every `call`, `jmp`, and `ret` instruction to hook API entry events **[C2]**. While this choice guarantees high recall, it can add an important overhead (Section 3.3.1) to the one already high [33] from the full-system emulation of QEMU. PyREBox has a remarkable collection of ~19K prototypes **[C3]** that we borrow in SNIPER. It logs return values and output arguments by hooking the `ret` instructions in the execution **[C4]**. For internal calls, users may only encode manual filtering policies that discard calls between specific pairs of DLLs **[C5]**. Finally, for derived flows **[C6]**, PyREBox follows child processes and conservatively monitors, by tracking `NtOpenProcess`, any process that the program interacts with. Our solution is generally less noisy as it tracks selected threads, ignoring the API calls from the legitimate activities of a victim process.

TABLE 2
API Call Count for Subjects Chosen From the Wine Conformance Tests

Test	SNIPER	API Monitor	SpyStudio	WinAPIOv.	drltrace
kernel32	2 853 159	2 853 160	51	-	2 853 159
gdi32	499 755	500 184	127	-	63 682
user32	791 233	791 558	266 139	2 314	791 105
ole32	69 476	69 481	92 418	31 474	64 157
oleaut32	299 101	299 105	0	20	23 132
shell32	8 947	8 943	15 008	2 466	6 944
crypt32	14 741	14 782	57	3 227	12 960
advapi32	3 215	3 257	-	760	3 001
wininet	5 668	5 709	1 057	984	4 718
iphlpapi	1 614	1 646	83	59	1 342
ws2_32	1 253	1258	24	350	927
kernelbase	138	139	41	119	138

The count refers to calls that we could identify as from program code, albeit some internal calls are still included for SpyStudio (especially in tests `ole32` and `shell32`) due to unreliable caller module information.

5 Evaluation

This section evaluates the capabilities of SNIPER and its performance and security aspects. We ran our implementations on Pin 3.15 and Xen 4.12 (DRAKVUF commit 376c03d), respectively. To collect the experimental figures reported in the following, we used an Intel i9-8950HK CPU, 3 GB of RAM, Windows 7 SP1 build 7601 32-bit, and strategy (b) for handling API exits. The DBI and VT-based variants yielded consistent results in the events recorded for each experiment. We observed no significant changes when repeating a subset of the tests on Windows 10. Our tracers currently support 446 syscalls and ~19K APIs from 194 DLLs. Users can select on startup the DLLs or groups of APIs to monitor.

5.1 Validation and Assessment of Other Systems

We initially tested our implementations using system utilities and programs shipped with Windows since they use heterogeneous, numerous APIs and occasionally make syscalls from program code. We then stressed them using:

- the extensive conformance tests [35] of the Wine emulator, which cover ~10K Windows APIs **[C3, C4, C5]**;
- tools popular in the context of malware sandbox testing like Al-Khaser [36] as they use many low-level, seldom undocumented primitives **[C2, C3]**;
- synthetic programs that we shielded with state-of-the-art executable protectors **[C1, C2]**; some exercised derived flows using injection patterns **[C6]**.

In short, the outcome of the testing backed our expectations for all the dimensions behind **[C1-6]**. As a byproduct of this validation process, we were also able to corroborate our qualitative claims from Section 4.5 on other tracers with reproducible tests for accuracy (in addition to the adversarial tests of Section 3.1.1). We make available the programs and configurations that we used on the project web page of SNIPER.

More in detail, we selected 12 popular DLLs that cover different Windows features: `advapi32`, `crypt32`, `gdi32`, `iphlpapi`, `kernel32`, `kernelbase`, `ole32`, `oleaut32`, `shell32`, `user32`, `wininet`, and `ws2_32.dll`. We then chose for an experimental comparison the Wine conformance tests specific to such libraries, as they exercise in their code a large deal of APIs. As most Wine tests come with multiple workloads per library, we selected the one that would at once: i) yield a deterministic sequence of API calls from program code, ii) not cause a crash on more than one of the tracers involved in the comparison, and iii) yield the highest count of API calls from program code among the workloads that met the two previous conditions.

We configured each system to simultaneously track all the APIs they support from the 12 DLLs (if any), as the Wine tests for a DLL frequently invoke also APIs from other DLLs for their working—and those calls are of interest as well. We then attempted to discard internal calls from the logs of each tracer by using caller module information, directly from the tracer when possible or via offline processing. Unfortunately, we were unable to test PyREBox due to excessive resource usage and high slowdowns that did not allow many of our tests to terminate in a reasonable time.

As our readers can see from Table 2, API Monitor stands out among existing systems as the most accurate tracer on these benchmarks. As we test benevolent programs, hooking IATs and rewriting the output of `GetProcAddress` (Section 4.5) is sufficient to capture all the APIs called from program code as long as their prototypes are known. The smaller values for the other tracers are a direct consequence of their more limited API databases **[C3]**. For SpyStudio, our filtering attempts were insufficient to discard a high number of internal calls on some subjects. For instance, for hundreds of internal API calls involving the Windows registry in the `shell32` test, SpyStudio erroneously reports the main executable as the caller module, while those calls originated internally in APIs that were truly called from program code. As for WinAPIOverride, the call counts are often the lowest among all tracers: while its database of prototypes is larger than the ones of drltrace and SpyStudio (Section 4.5), it misses key APIs that the Wine tests routinely invoke.

The call counts for SNIPER are almost identical to the ones for API Monitor. By manual investigation of the logs (details available in the supplementary material Appendix A, available in the online supplemental material), we found that most of the differences came from a single API that SNIPER did not instrument: `HeapAlloc` from `kernel32.dll`. The function turned out to be a forwarder export to `RtlAllocateHeap` from `ntdll.dll`, which was not part

TABLE 3
API Call Counts for the Subjects of Table 2 When Including Also Internal Calls

Test	SNIPER				API Monitor			
	Total # of DLL calls	From prog.	Log size	Log time	Total # of DLL calls	From prog.	Log size	Log time
kernel32	5 953 607	47.9%	850.0	51.9	3 148 988	90.6%	792.7	124.6
gdi32	3 346 379	14.9%	343.0	26.6	666 752	75.0%	186.3	37.4
user32	2 395 059	33.0%	307.0	17.0	1 573 131	50.3%	429.5	207.0
ole32	968 831	7.2%	31.0	4.7	567 052	12.2%	188.5	32.5
oleaut32	343 784	87.0%	74.0	3.9	313 281	95.5%	138.2	3.8
shell32	342 873	2.6%	2.6	3.7	182 529	4.9%	62.0	8.2
crypt32	179 716	8.2%	4.6	1.6	66 999	22.1%	21.8	3.2
advapi32	108 764	3.0%	1.1	2.5	66 519	4.9%	22.0	5.1
wininet	91 633	6.2%	1.7	2.2	65 502	8.7%	21.5	3.6
iphlpapi	66 731	2.4%	0.5	1.2	55 898	2.9%	19.0	4.1
ws2_32	37 533	3.3%	0.3	0.7	45 664	2.7%	17.9	3.0
kernelbase	2 260	6.1%	0.1	0.3	1 793	7.7%	0.6	1.4

Calls from program code are reported here as the percentage of all calls. Log sizes are in megabytes, log times are in seconds.

of the tracer configuration. SNIPER also found a few additional calls missed by API Monitor in the `gdi32` (13 from 6 functions), `user32` (28 from 3 functions), and `shell32` (7 from 2 functions) tests. Therefore, SNIPER was just as accurate as the best performer among existing systems while offering higher transparency and recall guarantees in the face of adversarial code (Section 4.5).

Table 3 provides the total call counts for the benchmarks and two resource usage metrics that help us highlight other architectural advantages of SNIPER also for non-security scenarios. We considered only API Monitor as a reference since the other systems traced too few API events for a meaningful comparison. We tracked the total number of DLL calls[10] that the tracer witnesses, the size of the logs that it produces, and the time spent logging during the execution, reporting the median value from 5 trials. Since we do not have access to the code of API Monitor, we measured its logging time by taking the difference between the first and the last timestamp in the logs and subtracting from it the execution duration with no DLLs selected for tracing. We followed the same approach for SNIPER and made a sanity check on the measurement by also computing it directly, adding counters to the analysis callbacks.

SNIPER witnesses a number of internal calls significantly higher than for API Monitor. This is a direct consequence of our design choice of maximizing recall **[C2]**, as hooks placed in API code see also intra-module calls done without resorting to the IAT, and to a lesser extent of our larger collection of prototypes. However, our analysis callbacks quickly discard them, resulting in a logging time that is smaller by a factor of 2 or more for 8 of the 12 subjects.

Discarding internal calls also results in much smaller logs for SNIPER except for the `kernel32` and `gdi32` tests. By manual inspection of the logs, we found out that the two (especially `gdi32`) make use of APIs for which input modifier information is not available for some of the arguments: in these cases, SNIPER conservatively logs them on both API entry and exit, while API Monitor often does not.

5.2 Capabilities and Performance

Whereas the Wine benchmarks are well-suited for an analytical comparison with other systems, for a more thorough evaluation of the capabilities and the performance of SNIPER, we studied 11 complex malware samples analyzed in [4] for their assorted anti-analysis patterns and 6 classic productivity programs. Table 4 reports the results.

We remark that the nature of these subjects is mainly non-deterministic. Therefore, to allow for an indirect quantitative comparison with other tracers, the table comes with dedicated columns for the respects where those would struggle most (i.e., internal calls, syscalls, output arguments) under the fictitious assumption that they survived the anti-analysis provisions from malware **[C1, C2, C6]**. We select for monitoring the same DLLs used for the experiments of Section 5.1.

The collected figures back our claim on the importance of distinguishing calls originating in program code from internal ones **[C5]**: the latter may be orders of magnitude more numerous. For DLL APIs, we further qualify internal calls as *normal* or *tail call* invocations: the ability to discard tail calls in *onEntry* (line 5) proved useful, as those can be as numerically relevant as the calls from program code (e.g., *Gootkit* and *Gozi-ISFB*). Syscalls from program code were few compared to those made within DLL code, yet they can reveal interesting details: for instance, for the *Furtim* malware, they proved vital for analysts to understand its adversarial strategies according to [4], [37].

Table 4 also reports how many distinct DLL APIs program code invoked, how many of them had output arguments, and the average number of arguments of all kinds from their prototypes **[C3, C4]**. Output arguments turned out relevant in practice as well, as they were present in 15–40% of the APIs we observed.

We also studied in greater detail the time spent executing our callbacks. The numbers shown in Table 4 refer to analysis code only, as probe insertion is a well-studied problem in DBI and VT-based research [2], [6], [15] that leaves us little optimization room. The average processing time for DLL APIs called from program code was 5-31 μs for each API entry or exit event, with a high correlation with the number of arguments to process (0.94 Pearson correlation between the *onEntry* processing time and the average number of arguments handled by *parseArgsOnEntry*), and with a minor role played by variable-size arguments.

Filtering out internal calls is cheaper than logging them. In particular, *onEntry* took 3–15 μs to terminate after the range (line 1) or the tail call (line 5) checks; *onExit* executed faster (< 1 μs) likely due to locality effects and, therefore, we omit its figures since hardly significant. For syscalls, we report figures only for internal ones as they are dominant; note that their enter analysis includes the verification step for the return address (Section 4.4). Finally, the additional cost for logging arguments in case of syscalls from program code was around a couple of dozens of μs.

[10] For some benchmarks, the calls seen by both systems can change slightly between trials. However, this difference comes exclusively from internal calls that Windows DLLs issue depending on the system state.

TABLE 4
API Calls Recorded on Complex Malware Samples and Common Productivity Programs

	# of syscalls		# of DLL calls			DLL APIs (from program code)			Avg call processing time (μs)				
	from program	internal	from program	internal		distinct	write to output args	avg # of args	program code		internal	syscalls (int.)	
Subject				tail call	normal				onEntry	onExit	onEntry	enter	exit
APT28	0	408 045	50 577	1 934	1 153 200	130	29	3.20	14.38	15.76	3.16	3.28	2.33
BlackSquid	0	12 172	4 667	988	55 715	151	38	2.82	17.42	17.81	14.27	10.76	3.71
Furtim	88	1511	541	371	2 365 887	71	25	2.49	16.53	30.79	2.69	3.61	4.27
Gootkit	0	3 068	4 737	4 478	31 507	79	23	1.37	5.61	8.27	8.69	4.17	2.86
Gozi-ISFB	19	1 509	13 449	11 019	22 180	75	28	1.54	5.60	6.45	3.18	4.13	9.45
Grobios	4	419	225	144	1 275	27	10	2.97	19.18	27.17	5.39	6.36	2.40
Olympic	0	1 129	434	298	4 726	64	26	3.26	18.38	23.36	4.59	8.22	16.83
SmokeLoader	15	485	49	27	1 019	28	10	3.94	29.50	21.20	9.86	5.39	2.53
Softpulse	1	1 552	1 163	628	10 702	83	26	2.82	15.37	20.65	8.61	6.39	2.44
Swisyin	0	7 058	1 392	451	81 456	22	8	4.38	27.52	20.76	3.47	17.17	1.83
Untukmu	0	105 646	23 978	21 195	4 459 691	25	8	2.19	10.63	11.29	2.51	3.46	1.92
7zip	0	28 398	5 922	294	139 152	112	26	3.80	24.36	23.37	4.55	3.89	2.41
BitTorrent	0	113 742	268 804	109 608	913 214	366	109	2.73	16.44	16.87	4.98	6.83	3.26
Chrome	1 821	263 839	1 586 718	684 236	755 054	398	145	2.99	19.68	21.56	8.83	9.21	2.64
Foxit Reader	2	150 490	946 903	205 319	818 568	396	93	3.45	17.78	19.69	4.33	5.23	2.12
Notepad++	0	315 440	2 955 873	1 645 034	725 638	231	36	2.04	8.63	10.57	3.90	3.15	2.24
TeamViewer	0	307 126	489 341	52 778	1 795 308	328	87	3.36	21.75	23.95	4.22	6.75	7.80

As for the end-to-end logging time, we report on the two samples with the highest counts of DLL calls from program code (APT28) and internal ones (Untukmu), respectively. For APT28, SNIPERspent 1.52s to record ~50K DLL calls from program code, 3.65s to discard ~1.1M internal ones, and 2.29s to discard ~408K internal syscalls. For Untukmu, it took 11.77s to process ~4.4M internal calls, 0.52s for those from program code, and 0.57s for ~105K internal syscalls.

5.3 Security

We conclude our evaluation by discussing the security aspects of the design and its implementations. As threat model for the transparency challenge **[C1]** (Section 3.1.1), we considered introspective attempts on the in-memory contents for the executable (T1) and the DLLs (T2): both our variants successfully pass those tests. By design, the analysis code and data structures are kept separate from the monitored application: they reside in the VM monitor in the VT-based scenario and we use the shielding techniques of [11] for Pin. We make no visible changes to the executable or the DLLs.

Let us consider a sophisticated malware scenario where an adversary aware of the inner workings of SNIPER tries to either evade or break its tracing process.

By evasion, we mean that the program makes a call that SNIPER misses or erroneously discards as internal. We remark that our instrumentation is exhaustive in terms of recall and copes with derived execution flows. Therefore, to deceive the tracer, the only option we foresee for an adversary is a TOCTTOU (time of check to time of use) attack: at API call time, the adversary push on the stack a fake return address falling in the blacklist of return ranges and later replaces it from a concurrent thread with the intended return address before the API terminates. While the reliability of such an attack is yet to be explored in practice, strategy (a) for API exits would come to the rescue: the intended return address has to be visible when an API reaches one of its exit points, so we could use that moment to decide whether or not to log the call.

By breaking the tracing process, we mean that the adversary tries to make SNIPER crash or to subvert its control flow. As analysis code and data are in regions distinct from program ones, the adversary should aim for data handling bugs by feeding poisonous data to the tracer. We picture two avenues: exhausting the shadow stack and targeting print helpers for argument types. We rule out the first possibility because, irrespectively of the size chosen for the stack (we also remark that our implementations use a resizable C++ vector), constructing an arbitrarily long sequence of nested API calls is not plausible, plus our callbacks discard internal calls. As for arguments, we only copy them: primitive types have fixed sizes, and we make conservative provisions for variable-length data (Section 4.3.3). We thus foresee no direct way to break the print helpers either.

6 Discussion

This section sets out reflections on the generality of the design principles behind SNIPER and examines its residual attack surface, highlighting further research opportunities.

Instrumentation Technologies

The design points that we recommend do not make use of primitives that are exclusive to specific instrumentation technologies. For this reason, they are amenable to different instrumentation solutions: for SNIPER, we chose two technologies that can cope well with the threat model of **[C1]** and the other requirements **[C2-6]** of Section 3.1.

While those technologies are well-known in security research, their deployment required careful implementation work to pursue accuracy and efficiency in the tracing process. For instance, we had to augment invisible breakpoints with a filtering mechanism for execution units, for which we trace creation and termination events in a separate VMI component. This aspect was not handled in DRAKVUF, while the authors of SPIDER only discuss the case of single processes. Furthermore, prior literature did not cover the lazy loading of Windows DLLs (Section 4.3.1).

We did not consider using QEMU as a whole-system DBT engine because VT-based schemes execute at native speed and bring fewer artifacts. However, we foresee no major obstacle to a QEMU-based implementation: infrastructure to hook instruction addresses can be found in projects like [33], while the VMI component for tracking execution units would need only QEMU-specific adaptations.

Compatibility with Other Platforms

Early in the article, we claimed that the design points behind SNIPER are compatible with other mainstream operating systems. In the following, we detail the implementation changes required for Linux and briefly discuss macOS.

Linux programs and libraries rely on the *Global Offset Table* (GOT) and the *Procedure Linkage Table* (PLT) to solve, respectively, position-independent address calculations and function calls to absolute locations. Unlike the IAT mechanism of Windows, a Linux program typically solves external functions via lazy binding. On the first invocation of a dynamically linked API, the executed PLT stub references a GOT entry value that will cause the dynamic linker to retrieve the absolute address of the function and write it to the GOT before calling it. In this way, any subsequent API invocations will see the PLT stub jump directly to the previously found function address. To insert our instrumentation, it would suffice to intercept at load time any external module that needs monitoring and parse its symbol table to locate the addresses of its public functions. A similar strategy may be used for statically linked libraries. As for instrumenting syscalls, DBI runtimes naturally interpose on their execution (Section 4.4); for a VT-based solution, we may place hooks in kernel code as we did in the Windows case.

For API prototypes, one may start from existing collections of symbols for the popular user-space `ltrace` tracer[11], as they include type and modifier information for each argument. The analysis callbacks must then account for the different calling conventions: for instance, in Fig. 3 at line 18 the field *cInfo.prototype.retN* would be always zero. Finally, the VMI component responsible for tracking any derived flows should monitor (at least) the `fork`, `clone`, and `execve` syscalls and the `do_exit` kernel function, while DBI runtimes can follow child processes also on Linux.

While our experience with macOS is more limited, we observe that the `dyld` loader for Mach-O files supports static symbols, weakly linked symbols, and lazy binding for dynamic linking. In all these cases, one can always locate destination symbols upon program/module loading and place instrumentation in the body of the involved functions. As for analysis callbacks, macOS follows the same calling conventions of Linux. At the time of writing, we are not aware of public implementations of invisible breakpoints on macOS, while the developers of Pin are actively working on providing full macOS support [38].

On a related note, one may explore our design also on other architectures, adapting the callbacks in the stack analysis and CPU register contents retrieval parts. In the ARM realm, Proskurin *et al.* in [39] present implementations of invisible breakpoints for ARMv7 and ARMv8, whereas DynamoRIO currently supports AArch32 and AArch64 Linux programs for DBI. We leave this possibility to future work.

Residual Attack Surface

In the arms race of anti-analysis techniques and countermeasures, a principled approach to API monitoring may give defenders an edge against sophisticated malware. Researchers and practitioners may devise our design principles over ever-improving instrumentation technologies and compose them with other techniques to cope with present and future attacks that are outside the current threat model of SNIPER.

The design of SNIPER does not address evasions targeting the peculiarities of the underlying instrumentation technique. For this well-studied problem [34], [40], [41], the implementation can resort to existing mitigations, such as patching the Time Stamp Counter in the VM monitor upon VM exit events or hiding artifacts of a DBI runtime as we did by using the mitigation library of [11].

Kawakoya *et al.* in [20] use taint analysis for an adversarial model for API monitoring where an attacker can evade hooks by emulating with own instructions the initial portion of an API before jumping in the middle of its canonical implementation. We may cope with popular forms of such *stolen code* attacks by moving the hook from the initial instruction of an API to a later basic block (for instance, one that post-dominates the entry block in the control flow graph) where the arguments are still visible. The authors also consider code injection attacks to elude monitoring using other processes: we tackled this surface by tracking child processes and remote threads. We can extend our implementations to recognize new exotic injections [42] since SNIPER captures the API calls needed to mount them.

Our system is deceived by attacks against API name resolution based on non-standard loading of DLLs. One described in [20]—and countered by the authors using disk-level taint analysis—copies a system DLL to a non-standard path and alters its exported symbols before loading it. A more fragile and conspicuous variant copies from a loaded DLL the code of individual APIs to program RWX memory; note that internal calls may still leak, as the outermost ones would appear as from program code. In a more complex attack [43], the adversary reimplements the Windows loader and recursively rewires every import referencing other DLLs to use stealth copies of such libraries so that the program never calls "standard" API functions. As future work, we look at the countermeasures suggested in [43] to extend our system for coping with these orthogonal attacks.

7 Other Related Works

To dynamically analyze executables, researchers have used for a long time instrumentation and analysis schemes operating alongside the object of the analysis.

In the context of monitoring the interactions of a program with the surrounding environment, analysis systems have ranged from operation-specific tracers (e.g., [44]) to fully-fledged sandboxes. A common strategy was to patch the OS API functions of interest [1]. For instance, in its day, the pioneering CWSandbox [45] replaced the first 5 bytes of each API of interest with a trampoline to an analysis callback. At present, Cuckoo Sandbox inserts similar trampolines to monitor ∼320 API functions for its analyses [46].

When developing a program, it may be useful to interpose on specific functions for a variety of reasons, such as adding

[11] Also `ltrace` faces transparency issues: for interposition, it places a conspicuous `int 3` instruction in PLT stubs with `PTRACE_POKETEXT`.

debugging instrumentation or sanitizing sensitive arguments. Frameworks like Microsoft Detours [47], Deviare In-Process [22], and EasyHook [48] allow users to insert trampolines in functions selected among DLL exports and program code (provided that debugging information is available for the latter). Typically, the user writes a code module that invokes framework primitives to register arbitrary callbacks for the functions of interests. The framework then takes care of code injection, argument marshaling, thread safety, and other low-level aspects. While these frameworks may even support the implementation of a general-purpose API tracer (as we mentioned in Section 4.5, SpyStudio builds on Deviare In-Proc), they remain lackluster in terms of transparency due to their conspicuous trampolines. Furthermore, as we reported in Section 3.1, patching-based mechanisms like trampolines or IAT hooking (Section 2.2) may conflict with recent OS mitigations for hardening processes [19].

The code changes and artifacts introduced by all the approaches above have worried researchers and practitioners already for some time [1]. In a seminal work [49], Garfinkel and Rosenblum propose to move an intrusion detection system from the guest to the VM monitor, using VMI techniques to inspect the guest with better transparency and isolation. VMI was later adopted in many other scenarios, first and foremost malware analysis and memory forensics.

Ether [5] pioneered low-artifact malware analysis with a system based on VT extensions with syscall tracing capabilities. For interposition, Ether used bogus values in the `SYSENTER_EIP_MSR` register and in the 0x2e interrupt descriptor table entry to cause a page fault whenever the guest triggered a syscall using `sysenter` or `int 2e`. However, it could not capture calls to user-space code.

SPIDER [6] brought new capabilities to the table with invisible breakpoints for instrumenting selected instruction addresses (Section 2.2, Section 4.3.1). The paper also presents two case studies, attack provenance and confidential data acquisition, that see a manual hook placement. Invisible breakpoints are a fundamental building block of our VT-based tracer, which comes with original additions (Section 4.2, Section 4.3.1, Section 6) required for the usage scenario that we pursue in this article.

To the best of our knowledge, this article is the first work to identify and propose solutions for the challenges that arise when trying to monitor the vast universe of APIs offered by Windows. Also, it represents the first attempt to extend VT-based monitoring to arbitrary user-space API calls in an automated, general-purpose manner.

Our system shares similarities with hprobes [2], a framework that uses VT extensions for hook insertion in user space. The work discusses three software dependability case studies: an emergency exploit detector, a watchdog, and an infinite loop detector. To insert a hook, hprobes overwrites the instruction at the address of interest with `int3` to have the VM monitor kick in and carry out the analysis. We find hprobes to serve a purpose orthogonal to ours, as it means to back generic, user-supplied analyses for specific events. Unlike the invisible breakpoint from SPIDER, hprobes makes no provisions for hiding code changes, so its technique is not transparent to checksumming attempts (Section 3.1.1).

This limitation is shared by designs for secure hook insertion in a VM with a modified OS (e.g., [50], [51]), which are an alternative to invisible breakpoints. Recent developments in this area (e.g., [52], [53]) feature efficient isolation using the `VMFUNC` feature of VT extensions but introduce distinguishable code artifacts. Lately, the OASIS system [54] has made promising improvements in this direction.

We conclude by discussing DBT systems. Those are a popular choice for implementing security analyses that require fine-grained instrumentation capabilities (such as tracking instructions by type) or when substantial code modifications are needed. DBT systems usually offer better transparency and flexibility than binary patching [11], although they may incur emulation artifacts [55].

A recent work [56] uses DBI for real-time call detection of functions from program code. Unlike DLLs with their export information, an executable does not declare the entry points of those functions. The authors show how to scrutinize control transfer instructions to identify function calls reliably. We believe it would be interesting to compare this approach in terms of recall and efficiency with a solution combining our design for entry events with recent advances for function detection in binaries [57].

8 Conclusion

API monitoring is a valuable technique in many research scenarios. In this article, we identified and addressed key challenges towards robust API monitoring, a task for which existing systems reveal several shortcomings. We discussed how to build tracing solutions for Windows APIs and their multifaceted universe of challenges, suggesting general design points amenable to different instrumentation technologies. Our techniques are general: they make no assumption on how a program is compiled or obfuscated, but only on the calling conventions in use and the availability of function prototypes. Therefore, they may also be applied to other OSes or to the own components of an application.

References

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008.

[2] Z. J. Estrada, C. Pham, F. Deng, L. Yan, Z. Kalbarczyk, and R. K. Iyer, "Dynamic VM dependability monitoring using hypervisor probes," in *Proc. 11th Conf. Eur. Dependable Comput.*, 2015, pp. 61–72.

[3] V. Golender, I. Ben Moshe, and S. Wygodny, "System and method for troubleshooting software configuration problems using application tracing," U.S. Patent 7386839B1, Jun. 10, 2008.

[4] D. C. D'Elia, E. Coppa, F. Palmaro, and L. Cavallaro, "On the dissection of evasive malware," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 2750–2765, Feb. 2020.

[5] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. 15th ACM Conf. Comput. Commun. Secur.*, 2008, pp. 51–62.

[6] Z. Deng, X. Zhang, and D. Xu, "SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proc. 29th Annu. Comput. Secur. Appl. Conf.*, 2013, pp. 289–298.

[7] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, 2014, pp. 386–395.

[8] D. Plohmann, S. Eschweiler, and E. Gerhards-Padilla, "Patterns of a cooperative malware analysis workflow," in *Proc. 5th Int. Conf. Cyber Conflict*, 2013, pp. 1–18.

[9] M. Leszczyński and K. Stopczański, "A new open-source hypervisor-level malware monitoring and extraction system - current state and further challenges," in *VB2020 localhost. Virus Bull.*, 2020, Accessed: Sep. 7, 2021. [Online]. Available: https://vb2020.vblocalhost.com/uploads/VB2020-Leszczynski-Stopczanski.pdf

[10] S. Z. Mohd Shaid and M. A. Maarof, "In memory detection of Windows API call hooking technique," in *Proc. Int. Conf. Comput., Commun., Control Technol.*, 2015, pp. 294–298.

[11] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed)," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2019, pp. 15–27.

[12] Microsoft developer blogs, "Exported functions that are really forwarders," Accessed: Mar. 11, 2021. [Online]. Available: https://devblogs.microsoft.com/oldnewthing/?p=30473

[13] Microsoft, "Argument passing and naming conventions," Accessed: Mar. 11, 2021. [Online]. Available: https://docs.microsoft.com/en-us/cpp/cpp/argument-passing-and-naming-conventions

[14] Microsoft, "Header annotations," Accessed: Mar. 11, 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/winprog/header-annotations

[15] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proc. Int. Symp. Code Gener. Optim.*, 2003, pp. 265–275.

[16] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200.

[17] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2005, Art. no. 41.

[18] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. IEEE Symp. Secur. Privacy.*, 2011, pp. 297–312.

[19] Microsoft, "Compatibility considerations for Import address filtering," Accessed: Mar. 11, 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/exploit-protection-reference

[20] Y. Kawakoya, M. Iwamura, E. Shioji, and T. Hariu, "API Chaser: Anti-analysis resistant malware analyzer," in *Proc. Res. Attacks, Intrusions, Defenses.*, 2013, pp. 123–143.

[21] A. Case *et al.*, "HookTracer: A system for automated and accessible API hooks analysis," *Digit. Investigation*, vol. 29, pp. S104–S112, 2019.

[22] Nektra, "Deviare API hook," Accessed: Mar. 11, 2021: [Online]. Available: https://www.nektra.com/products/deviare-api-hook-windows/

[23] Zynamics, "MSDN crawler," Accessed: Mar. 2021. [Online]. Available: https://github.com/zynamics/msdn-crawler/

[24] D. C. D'Elia, L. Invidia, and L. Querzoni, "Rope: Covert multi-process malware execution with return-oriented programming," in *Proc. Eur. Symp. Res. Comput. Secur.*, E. Bertino, H. Shulman, and M. Waidner, Eds., Oct. 2021, pp. 199–217.

[25] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proc. 8th ACM SIGPLAN/SIGOPS Conf. Virtual Execution Environ.*, 2012, pp. 133–144.

[26] Intel, "Instrumentation granularity," in *Pin Official Documentation (release 97998)*, Accessed: Mar. 11, 2021. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/97998/Pin/html/

[27] Rekall, Accessed: Mar. 11, 2021. [Online]. Available: http://www.rekall-forensic.com/

[28] DRAKVUF project page, "Discussion on improvements around usermode hooking," Accessed: Mar. 11, 2021. [Online]. Available: https://github.com/tklengyel/drakvuf/issues/669

[29] DRAKVUF project page, "memdump: Dumps based on user mode API calls," Accessed: Mar. 11, 2021. [Online]. Available: https://github.com/tklengyel/drakvuf/pull/675

[30] LibVMI, Accessed: Mar. 11, 2021. [Online]. Available: https://github.com/libvmi/libvmi

[31] M. Shudrak, D. Bruening, and J. Testa, "Drltrace," Accessed: Mar. 11, 2021. [Online]. Available: https://github.com/mxmssh/drltrace

[32] Cisco Talos, "PyREBox: Python scriptable reverse engineering sandbox," Accessed: Mar. 11, 2021. [Online]. Available: https://talosintelligence.com/pyrebox

[33] A. Davanian, Z. Qi, Y. Qu, and H. Yin, "DECAF++: Elastic whole-system dynamic taint analysis," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses.*, 2019, pp. 31–45.

[34] D. C. D'Elia, L. Invidia, F. Palmaro, and L. Querzoni, "Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts," *Digit. Threats Res. Pract.*, 2021.

[35] Wine project, "Writing conformance tests (Wine's developer guide)," Accessed: Mar. 11, 2021. [Online]. Available: https://wiki.winehq.org/Wine_Developer%27s_Guide/Writing_Conformance_Tests

[36] "Al-Khaser," Accessed: Mar. 11, 2021. [Online]. Available: https://github.com/LordNoteworthy/al-khaser

[37] SentinelOne, "SFG: Furtim malware analysis," 2016, Accessed: Mar. 11, 2021. [Online]. Available: https://www.sentinelone.com/blog/sfg-furtims-parent/

[38] Intel, "Release notes for Pin 3.20," Accessed: Sep. 2021. [Online]. Available: https://software.intel.com/sites/landingpage/pintool/docs/98437/README

[39] S. Proskurin, T. Lengyel, M. Momeu, C. Eckert, and A. Zarras, "Hiding in the shadows: Empowering ARM for stealthy virtual machine introspection," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 407–417.

[40] M. Brengel, M. Backes, and C. Rossow, "Detecting hardware-assisted virtualization," in *Proc. Detection Intrusions Malware, Vulnerability Assessment.*, 2016, pp. 207–227.

[41] A. Dos Santos Fh, R. J. Rodríguez, and E. L. Feitosa, "Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks," *Digit. Threats: Res. Practice*, 2021.

[42] A. Klein and I. Kotler, "Windows process injection in 2019 (Process injection techniques - gotta catch them all)," *Black Hat USA*, 2019, Accessed: Mar. 11, 2021. [Online]. Available: https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf

[43] Y. Kawakoya, E. Shioji, Y. Otsuki, M. Iwamura, and T. Yada, "Stealth loader: Trace-free program loading for API obfuscation," in *Proc. Int. Symp. Research Attacks, Intrusions, Defenses*, 2017, pp. 217–237.

[44] Microsoft, "Sysinternals suite," Accessed: Mar. 11, 2021. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/downloads/regmon

[45] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Secur. Privacy*, vol. 5, no. 2, pp. 32–39, Mar. 2007.

[46] Cuckoo Sandbox, Accessed: Mar. 11, 2021. [Online]. Available: https://github.com/cuckoosandbox/cuckoo/wiki/Hooked-APIs-and-Categories

[47] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 functions," in *Proc. 3rd USENIX Windows NT Symp.*, 1999, pp. 135–143.

[48] C. Husse and J. Stenning, "EasyHook," Accessed: Sep. 2021. [Online]. Available: https://easyhook.github.io/

[49] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2003. [Online]. Available: https://www.ndss-symposium.org/ndss2003/

[50] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. IEEE Symp. Secur. Privacy*, May 2008, pp. 233–247.

[51] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 477–487.

[52] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1607–1619.

[53] B. Shi, L. Cui, B. Li, X. Liu, Z. Hao, and H. Shen, "ShadowMonitor: An effective in-VM monitoring framework with hardware-enforced isolation," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses.*, 2018, pp. 670–690.

[54] J. Hong and X. Ding, "A novel dynamic analysis infrastructure to instrument untrusted execution flow across user-kernel spaces," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 402–418.

[55] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, and D. Bruschi, "A methodology for testing CPU emulators," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 1–26, Oct. 2013.

[56] F. de Goër, S. Rawat, D. Andriesse, H. Bos, and R. Groz, "Now you see me: Real-time dynamic function call detection," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 618–628.

[57] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proc. 2nd IEEE Eur. Symp. Secur. Privacy.*, Apr. 2017, pp. 177–189.

Daniele Cono D'Elia received the PhD degree in computer science engineering from the Sapienza University of Rome, in 2016. He is currently a postdoctoral with Sapienza. His research interests include software and systems security, with contributions in the analysis of adversarial code and in the design of program analyses and transformations that help in making software more secure.

Simone Nicchi received the MSc degree in computer science engineering from the Sapienza University of Rome in 2018. His research explores how program analysis techniques can help in countering anti-analysis behavior from the malware domain.

Matteo Mariani reseived the MSc degree in computer science engineering from the Sapienza University of Rome in 2019. He is currently with Leonardo as member of the Cyber Security Research Lab working on endpoint security systems.

Matteo Marini received the BSc degree in computer and system engineering in 2020 from the Sapienza University of Rome, where he is currently working toward the MSc degree in computer science engineering.

Federico Palmaro received the MSc degree in computer science engineering from the Sapienza University of Rome in 2018. He is currently with Prisma researching evasive malware and related dynamic analysis systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.