

AN IN-KERNEL FORENSICS ENGINE FOR INVESTIGATING EVASIVE ATTACKS

Javad Zandi

Florida International University

Lalchandra Rampersaud

Florida International University

Amin Kharraz

Florida International University

ABSTRACT

Over the years, adversarial attempts against critical services have become more effective and sophisticated in launching low-profile attacks. This trend has always been concerning. However, an even more alarming trend is the increasing difficulty of collecting relevant evidence about these attacks and the involved threat actors in the early stages before significant damage is done. This issue puts defenders at a significant disadvantage, as it becomes exceedingly difficult to understand the attack details and formulate an appropriate response.

Developing robust forensics tools to collect evidence about modern threats has never been easy. One main challenge is to provide a robust trade-off between achieving sufficient visibility while leaving minimal detectable artifacts. This paper will introduce LASE, an open-source Low-Artifact Forensics Engine to perform threat analysis and forensics in Windows operating system. LASE augments current analysis tools by providing detailed, system-wide monitoring capabilities while minimizing detectable artifacts. We designed multiple deployment scenarios, showing LASE's potential in evidence gathering and threat reasoning in a real-world setting. By making LASE and its execution trace data available to the broader research community, this work encourages further exploration in the field by reducing the engineering costs for threat analysis and building a longitudinal behavioral analysis catalog for diverse security domains.

1 INTRODUCTION

Modern attacks against critical infrastructure are continuously getting cheaper, faster, and more consequential. For instance, in December 2021, just a few days after the Log4j vulnerability was disclosed [21], adversaries started to adapt their code to locate exposed log4j services on the Internet and actively exploited this vulnerability. Very recently, the Clop ransomware group claimed responsibility for attacking the MOVEit transfer service on several known financial, education, U.S. federal, and state governments over the Memorial Day holiday. The attack was so consequential that the U.S. State Department announced a \$10 million bounty for information on the actors and involved threat campaign [67]. These incidents, on the one hand, suggest that adversaries are opportunistic and have asymmetric power in repurposing their tools to target new vulnerabilities in exposed services. However, a more concerning issue is that collecting necessary forensic evidence about these incidents at the early stages of the attacks before they cause consequential damage to the target systems is often challenging.

Over the years, the security community has invested significant effort to close this gap by developing solutions to extract insights from large volumes of raw network [7, 33, 41] or default logging traces [36, 41, 46, 49, 62, 73, 129] that primarily reflect temporal usage of systems resources (e.g., CPU, RAM, Network, Disk). While these efforts have been instrumental in detecting specific forms of attacks with predictable patterns (e.g., Ransomware [59], crypto mining [120]), they often fail to manifest fine-grained information on attack techniques that implement not evident or less known evasive techniques to deliver their payloads. Consider an attack where the offending process injects a payload into another process and uses that as a proxy to establish a backdoor to a remote server. This incident, as a sequence of system-wide temporal data, is not collected effectively by engines that rely on default logging mechanisms. This insufficient visibility over the dynamic behavior of malicious operations can put defenders in a highly disadvantaged

position to understand the attack tactics and lateral movements, perform root cause analysis, and formulate a proper response.

The core **insight** in this paper is today's threat intelligence against modern evasive attacks has to provide fine-grained system-wide visibility, leave minimal detectable artifacts, and be portable. These design goals have been historically difficult to attain at the same time. For instance, having fine-grained temporal visibility requires looking into very low-level components of the systems, which could make the solution less portable. On the other hand, approaches such as API hooking offer great portability, but there are several ways to bypass them. This paper aims to move towards addressing this need by introducing LASE— an open-source Low-Artifact ForenSics Engine, that aims to define a balance between offering *fine-grained visibility* while *minimizing detectable* artifacts. These two design requirements are critical to staying effective over time in a landscape where adversaries actively try to identify defense solutions. To this end, LASE is deployed as a system-wide in-kernel engine that operates in high-privileged mode, making it almost impossible for user-mode applications to fingerprint, tamper, or kill the engine. At the same time, it offers system-wide visibility, temporal data on processes and threads, I/O requests, synchronous and asynchronous I/Os, fast I/Os, which are essentials to record the behavior of various forms of evasive attacks.

We highlight the portable design of LASE through two distinct security-motivated case studies: (1) an in-kernel baremetal-assisted threat analysis environment: we deployed the engine on several live-running physical machines to perform large-scale malware analysis on baremetal Windows machines (Section 4.1), (2) a distributed deception-based infrastructure: We deployed the LASE-enabled images as in-cloud deception-based threat infrastructure for 46 days and collected attack artifacts about real-world attacks on intentionally vulnerable systems in the wild (Section 4.2). In the following, we summarize the key findings of the paper.

The experiment shows that LASE-enabled baremetal analysis environment can complement current malicious code analysis tools on large-scale threat analysis and measurement. We created Windows 10 OS images enabled by LASE and deployed them across eight physical machines, forming one of the first scalable baremetal-assisted analysis environments. We analyzed 79,544 malicious binaries over 30 months, from June 2022 to December 2024. Our analysis of recorded activities shows that a low-artifact engine can increase visibility over the behavior of malicious code. For instance, we collected 5,284,059 files after executing 34,438 malware samples that have file write activities in baremetal and virtualized environments. We observed that 3,981,555 (75.35%) of the files were delivered only in the baremetal environment, and 884,301 (16.74%) were only dropped in the virtualized environment. This leaves 418,203 files that were delivered in both – showing that the samples fetched different payloads. We have shared 2% of the dataset (117GB) for this submission, which contains filesystem artifacts for bare-metal and virtualized analysis environments.

The analysis shows the emerging trend of using hardware-based fingerprinting makes malicious code analysis increasingly more challenging. In particular, among all forms of fingerprinting techniques we observed in the dataset, 1,187 of the samples were calling Windows Management Instrumentation (WMI) to detect advanced virtualized analysis systems (e.g., Hypervisor-based sandbox), 9,254 samples from Sabsik, Wacatac, AgentTesla families checked for direct CPU clock access, and 7,147 samples checking the BIOS hardware, firmware version, manufacturer, and configurations loaded during the system boot process as shown in table 5. While advanced sandboxes may attempt to mask these artifacts and report different driver versions or hardware information, such modifications significantly reduce the robustness of the target system. That is, these changes may result in a complete system crash or disruption of legitimate drivers because critical services often need exact information of hardware information for critical operations (e.g., integrity checking, patch and update, driver management).

Finally, LASE was deployed as a portable forensics engine in a deception-based threat intelligence environment for 46 days by intentionally exposing vulnerable Windows services on the LASE-enabled cloud hosts. During the course of the experiment, we identified 734 successful intrusions. Our analysis shows that it takes approximately 6 hours for adversaries to discover the exposed vulnerable service and 13 hours to compromise the service. The analysis of the process and filesystem traces shows the execution of 1,221 executables, a collection of 401 shell scripts, 235 installers, 14,706 source code files, 438 digital certificates, and 4,586 custom dynamic libraries.

The proposed forensics engine and discussed use cases aimed to highlight a critical gap in security defense for portable yet robust forensics tools for attack analysis, bringing more behavioral visibility without introducing trivial detectable artifacts. Our hope is that this work serves to raise awareness about the importance of forensics analysis frameworks for global visibility and intelligence gathering in today’s attack landscape. We also hope that our approach will prove helpful to the security community in identifying emerging threats that go beyond what

is routinely observed today and open the door to future research on threat intelligence and malicious code analysis.

Contributions. The paper makes the following contributions:

- We propose an in-kernel open-source forensics engine that can be deployed in all modern versions of Windows operating systems. The forensics engine collects almost all forensically relevant information at the process and thread level as well as I/O activities for analyzing run-time behavior of malicious.
- We deployed the engine in two case studies, showing the benefits of a low overhead security forensics engine in providing insights into the threat landscape. We collected artifacts about 734 successful attacks on our deception environment. We also analyzed over 79K samples from hundreds of malware families and collected over 2.6 TBs of compressed threat artifacts over 30 months of experiments.
- The source-code¹ as well as the artifacts used in the analysis of the paper (2% of the data catalog)² are made available. The distribution of malware families for the artifacts submitted can be found in Table 6.

2 BACKGROUND / MOTIVATION

In this section, we begin by describing the threat model to describe attackers’ capabilities in designing and developing malicious code and evasion mechanisms employed to bypass possible defense solutions. We then motivate the work by describing what is lacking in the current solutions to bring more visibility into the attack landscape.

2.1 Threat Model

In this paper, we assume that adversaries have significant freedom to develop evasive malicious code, as documented in prior work and community reports. In particular, our assumptions about attackers’ capabilities are as follows:

Environment Sensitive: There is no lack of evidence that modern evasive attacks are armed with various forms of anti-analysis techniques [4, 14, 16, 17, 28, 62, 63, 70, 92, 103, 112]. A long list of those artifacts (e.g., specific processes, registry keys, services, and network adapters) is publicly reported [26]. These techniques are often used by malicious code developers to successfully intrude on a system where they plan to launch code.

Debugging Resistant: Adversaries have historically incorporated anti-debugging techniques to complicate forensics analysis. In particular, one common debugging approach relies on code injection techniques [5, 13, 66, 124] in the context of the target payload. That is, analysis agents (i.e., DLL) are injected into the context of a malicious process for behavioral monitoring. There have been several techniques to automatically detect such analysis techniques by simply listing active processes on target systems to identify debuggers or creating snapshots of their own memory structures, such as their heap and modules

¹<https://tinyurl.com/LASECode>

²<https://tinyurl.com/LASEArt>

(DLLs loaded in the process’s virtual address space), to identify if they are being analyzed. Adversaries can incorporate anti-debugging techniques at different layers, including CPU registers, in-memory data structure checks, or calling specific APIs or native functions [13, 34, 74, 79, 102, 124].

Launching Proxy Operations: Malicious processes often inject the actual malicious code into the context of legitimate processes to bypass reputation-based services [94, 95]. This is an important assumption as it makes most debugging techniques that target particular processes ineffective. Hence, our threat model in this project will consider these and other similar risks, and the solutions that will be developed should be robust and tested against these malicious activities [48, 53, 94, 95].

In this study, we also assume that user-initiated processes solely operate in user mode. Consequently, interactions with lower-level system resources are channeled through operating system API calls, which can be intercepted by a low-level forensic engine operating in the kernel space. In addition, we also assume that the trusted computing base includes the OS kernel and underlying software and hardware stack, and that normal user-based access control prevents attackers from running malicious code with superuser privileges.

2.2 *The Need for a Low-Artifact Forensics Engine*

To respond effectively to highly evasive methods described in the threat model, the security community has innovated at various layers to define their analysis frameworks [8, 41, 46, 57, 61, 62], depending either on whether they found it the most effective way or where it was easier to incorporate the proposed method. In this section, we describe the limitations of those methods and make the case for how the forensics engine should be implemented to analyze the majority of threats.

User-mode Hooking Techniques. Classic hooking techniques [13, 48, 74, 79, 111] often rely on inline overwriting of APIs which requires injecting custom-formulated payloads into the context of the target process address space to identify the invoked functions, which inherently leaves several artifacts (e.g., checking the integrity of the API function, monitoring the list of loaded processes). Furthermore, this is not a scalable approach due to the engineering efforts required to overwrite the target APIs. These modifications often are not without problems and could cause serious issues (e.g., system crashes) because of unexpected exceptions and unhandled cases. Lastly, function hooking or debugging mechanisms are inherently designed to be single process-centric and often lose their effectiveness rapidly when a more system-wide behavioral analysis is required. Furthermore, subverting these mechanisms is a common practice among adversaries by copying the desired malicious code into the address space of another process or using other common hooking evasion techniques such as customized code, stole code [19, 55, 56, 64, 115], and sliding calls [56].

Built-in Logging and Diagnosis Platforms. Modern operating systems are often equipped with integrated logging platforms designed for event monitoring and application performance tracking (e.g., syslog) [75]. In Windows, Event Tracing for Windows (ETW) [125] serves as a logging tool for system-wide monitoring and performance data collection. While ETW trace recording has been useful for collecting security events

(e.g., authentication logs), it was not primarily designed to stay undetected. Adversaries can enumerate active ETW providers that use it for analysis/inferencing purposes to list defense tools on target machines. This technique is often used to detect anti-malware tools on the target system. Furthermore, a user in an active session can disrupt the logging process by temporarily modifying ETW environment variables to redirect logs, disable logging in a specific context, or modify behaviors in an ETW-reliant application. That said, ETW offers significant visibility over the overall dynamic behavior of systems. However, the approach is vulnerable to fingerprinting techniques and disruption [22]. In Table 2, we also discuss a set of open-source forensics engine projects that are built on top of ETW service and compare with LASE.

Hypervisor-based Approaches. Hypervisor-based methods [12, 32, 37] have been proposed to address the above shortcomings by putting the analysis module out of the operating system to achieve increased visibility and analysis capabilities (e.g., memory extraction, low-cost analysis, and debugging). While these techniques offer significant flexibility for code analysis, there are still low-cost techniques to fingerprint hypervisor-based solutions by, for instance, abusing critical Windows core APIs (e.g., Windows Management Instrumentation [116, 121]) to extract hardware information or direct access to the CPU for detecting instruction execution delay. Disabling or overwriting these APIs is possible. However, these changes can be consequential because the normal operation of the many critical software components depends on accurately extracting hardware system information. Consequently, modifying these APIs can result in frequent system crashes and legitimate service disruptions.

2.3 *Design Requirements*

Given the benefits and shortcomings of current methods, we believe that an in-kernel behavioral recording method is an appropriate place to collect forensic data for a wide range of applications in threat analysis. In the following, we describe the security requirements we are seeking to achieve acceptable visibility without leaving detectable artifacts.

Low Artifact Operations. Low-artifact behavioral monitoring is a critical property in analyzing modern threats (e.g., analyzing malicious code, recording techniques, tactics, and procedures) because, in almost all of these incidents, adversaries’ goal is to identify signs of an analysis environment. As mentioned earlier, approaches such as user-mode hooking techniques or hypervisor techniques introduce specific forms of artifacts that can be weaponized by adversaries to analyze the environment. New threat intelligence frameworks should make the target systems more robust against these fingerprinting efforts.

System-wide Behavioral Monitoring. System-wide behavioral monitoring refers to the property that enables run-time behavioral monitoring in a multi-process environment. This is a critical property in modern forensic analysis towards achieving complete mediation since malicious code can launch the actual malicious payload via proxy operations (as mentioned in the threat modeling). A system-wide view can provide sufficient visibility about the interaction of a process not only with the

operating system resources but also with other processes [25]. From a threat intelligence standpoint, system-wide monitoring helps to establish a clear and comprehensive audit trail of all actions taken on a system by users or processes.

OS Support. It is critical that monitoring mechanisms operate in a high-privileged mode while introducing low integration costs. To this end, in-kernel forensics layer techniques are viable design choices to achieve these goals due to the significant OS support. An in-kernel forensics engine inherently runs as a privileged operation in the kernel-level [44]. Therefore, termination or manipulation of core functionalities will not be possible by the malicious process that operates at the user level. In particular, Windows officially supports the concept of protected services, called Early Launch Anti-Malware (ELAM), to allow critical services to run as protected services [65]. After the service is launched, Windows incorporates a code integrity mechanism to only allow trusted code to load into a protected service. Windows also protects these services from code injection [86, 91]. This approach will guarantee that the forensics engine is protected from common forms of availability, memory corruption, tampering, and code injection attacks.

Considering all design options and the proposed threat model, an in-kernel behavioral recording method appears to be a suitable layer for enabling a portable and low-artifact forensics layer to study common threats in the wild. That is, the in-kernel module does not modify any Windows APIs or inject any code into running processes. Furthermore, the flexibility to deploy the engine in baremetal systems makes the entire analysis system robust against advanced fingerprinting techniques that target hardware specifications. *That said, this design should not be considered as an alternative solution to the current hypervisor-based solution that can offer significant reverse engineering flexibility. Rather, it can serve as the first line of defense in threat analysis by satisfying fingerprinting checks that are impossible or costly in other defense systems.*

3 MONITORING RUN-TIME BEHAVIOR

In this section, we discuss the engineering decision and details of Low-Artifact ForenSics Engine (LASE). We elaborate on how we implemented the forensics engine. We provide examples of the collected artifacts and the deployment details of LASE to better analyze modern threats.

3.1 Forensics Agents and Major Components

This section describes the main modules of LASE that enables run-time trace recording.

I. Process and Thread Monitoring. Modern malicious code often employs code injection techniques such as DLL injection or process hollowing to inject code into legitimate processes [10, 11, 18, 56, 64, 93, 106]. This allows the malicious code to execute in the context of a legitimate process and run proxy attacks. Detecting such evasive operations has never been trivial since adversaries have significant freedom on what process to choose for injection, and making any assumption about the target process can impact the visibility factor. To this end, LASE records system-wide monitoring for processes and threads by recording process-related events

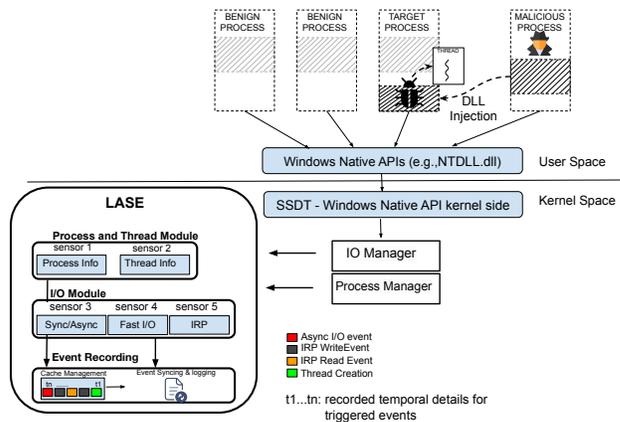


Figure 1: The high-level view of LASE’s architecture. LASE provides a system-wide multi-process forensics engine by defining three in-kernel modules: (1) Process and thread module, (2) Filesystem and I/O module, (3) Event recording module. In this figure, LASE records run-time behavior of the malicious process that uses DLL injection to launch a proxy attack.

such as *initiation and termination*, thread-level events such as *creation and deletion*, and process image loading events such as *load and unload*. This driver takes advantage of the Windows kernel APIs located in the `NtosKrnL.lib` [45, 52, 85] library and invokes embedded Windows Kernel functions: `PsSetCreateProcessNotifyRoutineEx()` [82], `PsSetCreateThreadNotifyRoutineEx()` [83] and `PsSetLoadImageNotifyRoutineEx()` [84] for process, thread and image activities respectively. Each of these functions allows the LASE driver to register a callback routine, which is registered in the kernel data structure within the process management subsystem and called by the Process Manager [128] whenever its corresponding event occurs. These functions are part of the process and thread management in Windows and provide mechanisms for kernel-mode components to monitor process lifecycle events.

II. I/O Request Packets. The execution of input and output operations in the Windows operating system, specifically related to driver operations, is facilitated by the utilization of packets. These packets, known as I/O packets or IRPs (I/O Request Packets), include encoded information that directs and controls the driver’s actions [104, 107]. The self-contained structure of the Input/Output Request Packet (IRP) encompasses all the necessary information for a driver to manage an I/O request. To enhance operational efficiency and contextual relevance, IRPs are classified into major and minor operations. The major function code instructs the driver on what action to take to fulfill the I/O request. Minor codes refer to a specific category of major operations and supply further detailed information to the driver. A major function code specifies the specific type of I/O operation in conjunction with an optional minor function code [87, 104]. Recording the forms of IRPs shown in table 8 in the Appendix is critical to understanding different classes of system interactions. That is, all filesystem activities, such as fetching files into the memory or interacting with cached files, are translated into a set of IRP requests that can be captured in the kernel. LASE

monitors all 103 IRPs, including 45 major and 48 minor I/O request packets (IRPs) based operations encompassing persistent and cached files.

III. Synchronous and Asynchronous I/O. In Windows, most of the I/O operations generated by processes are synchronous [89]. That is, a process waits until the OS returns a status code when the I/O is complete. For instance, ReadFile and WriteFile functions are executed synchronously. However, programs can modify the default behavior and issue asynchronous I/O requests by generating multiple I/O requests and continue executing while the OS performs the I/O operations. Our analysis shows that asynchronous behavior is largely ignored by almost all the current solutions [57, 61, 62, 114]. This is an important property because it allows processes to launch concurrent malicious operations, evading defense solutions that wait for the immediate return of the status code.

IV. Fast I/O. Programs can be designed to use Fast I/O requests for reading and writing to cached files, enabling rapid access synchronously [88]. In Fast I/O operations, the data is transferred directly between user buffers and the cache. This operation can bypass all current mechanisms that generate an IRP. For instance, if a process generates a Fast I/O operation requesting access to the content of an opened file in the memory, the requests will be processed immediately without generating any IRP events. This is an important observation since adversaries or malicious code can generate Fast I/O requests to access open files in the system buffer without generating any IRP artifacts [113].

V. Event Recording. The event recording module operates as a producer-consumer architecture [15] where producers generate filesystem and process events, and consumers use the events for processing, which include storing on disk and sending high-priority events to the network. We developed a trace recording library that offers a set of Application Programming Interfaces (APIs) to store the events and manage I/O event monitoring. The library provides a mechanism to create data containers to maintain I/O traces in the memory for further processing. Generating the data containers requires close interaction with the OS cache manager to allocate the necessary memory space. One approach to implement the in-memory data cache is a write-back cache [39] where the data is updated only in the cache, and permanent storage is updated later. The write-back cache is a straightforward caching mechanism and is an excellent fit for general read and write I/O events. However, continuous I/O monitoring can significantly increase the required cache size on machines with limited memory and storage capacity. As an alternative approach, we implement a circular buffer mechanism using a queue, where I/O events are broken into smaller and more manageable data chunks for processing and consumption. The caching policy is implemented as a multi-threading model to deliver data from multiple producers to multiple consumers concurrently.

3.2 I/O Benchmarks

LASE operates as an in-kernel module to record several forensics events. Consequently, it is important to evaluate the imposed overhead in a quantifiable fashion. To this end, we conducted an experiment to evaluate the overhead on filesystem

operations as they are the more common and intense operations due to the customized instrumentation layer and log collection process. We generated workloads by creating 500 small (10Kb) and large (10Mb) files to test the throughput of block write, rewrite, and read operations. We used these operations as the major events that require significant filesystem interaction. The disk I/O performance was assessed using the popular Windows file system benchmarking tool, IOZone [2] by measuring the time to process the creation and access of generated files. Each experiment was repeated ten times for 4.42 minutes, and the average score was calculated to yield the final results presented in table 1. The experiments show that LASE performs well with both small and large files and imposes an overhead between 1.74% and 5.31%. We consider the 2.01% overhead for reading an existing small file for the first time. One reason for the relatively lower overhead for these critical operations is the design decision to monitor forensics events with minimal changes to the standard subsystems. Note that LASE leverages an event-driven approach and does not require injecting code into the context of target processes. This approach has two main benefits: (1) it is a passive approach and avoids intrusive operations that may require any modification to the filesystem or standard OS functions, introducing lower I/O overhead compared to inline API modifications, (2) it minimizes possible interference with the normal operation of processes and reduces the risk of crashes and memory corruption.

Table 1: Disk I/O performance in a standard and host running LASE

Operation	Standard	LASE	Overhead
Writer			
Small	788,641 KB/s	808,474 KB/s	2.51%
Large	893,921 KB/s	909,448 KB/s	1.74%
Re-writer			
Small	1,036,665 KB/s	1,05,9468 KB/s	2.20%
Large	1,054,297 KB/s	1,092,756 KB/s	3.65%
Reader			
Small	3,564,507 KB/s	3,492,892 KB/s	2.01%
Large	2,841,287 KB/s	2,928,931 KB/s	3.08%
Re-Reader			
Small	4,228,550 KB/s	4,452,886 KB/s	5.31%
Large	3,643,169 KB/s	3,791,870 KB/s	4.08%

3.3 Resistance Against Common Dynamic Evasion Mechanisms

In the following, we briefly describe common fingerprinting checks used by adversaries in the wild and discuss the robustness of LASE against each method.

VM Checks. Modern attacks are sensitive to virtualized environments and may not run correctly in those systems [8]. Checking registry keys (e.g., reg_key, reg_key_value) [1], firmware details (e.g., firmware ACPI, RSMB), CPU information (e.g., cpuid) [1] and hardware information (e.g., model_computer_system_wmi) are just a few examples of such fingerprinting techniques [92]. LASE is portable and can be deployed on baremetal machines (see case study 1). Consequently, none of the VM checks would apply to the threat infrastructure enabled by LASE. We should also highlight that

Table 2: Comparing LASE with current forensics engines

Recorded Activities	LASE	BestEDR [126]	Fibratus [96]	Osquery [101]	Wazuh [123]	BlueSpawn [51]	Whids [105]
Technology	kernel driver	API Hooking	ETW	App log collection	App log collection	Win32 API	ETW, Sysinternal
Run-time Behavior							
Process-level Information	system-wide	single proc.	system-wide	single proc.	single proc.	single proc.	system-wide
Thread-level Information	system-wide	no thread	system-wide	no thread	no thread	single proc.	system-wide
DLL Injection	system-wide	single proc.	not supported	not supported	not supported	not supported	not supported
Inter-process Communication	system-wide	not supported	not supported	not supported	not supported	not supported	not supported
I/O Activities							
Filesystem I/O	system-wide	not supported	system-wide	not supported	not supported	single proc.	system-wide
Synchronous I/O	system-wide	not supported	system-wide	not supported	not supported	single proc.	system-wide
Asynchronous I/O	system-wide	not supported	not supported	not supported	not supported	not supported	not supported
Fast I/O	system-wide	not supported	not supported	not supported	not supported	not supported	not supported
Forensics Agents							
Logging Service	kernelmode	usermode	usermode	usermode	usermode	usermode	usermode
Scheduler Agent	kernelmode	usermode	usermode	usermode	usermode	usermode	usermode

LASE-enabled infrastructure is robust against advanced fingerprinting techniques (e.g., device information retrieval through Windows Management Instrumentation) [90] used to fingerprint hypervisor-based environments given that the reference platform does not use any virtualization technologies vulnerable to such fingerprinting attempts.

Anti-Debugging Checks. Anti-debugging techniques are strategies to thwart efforts to understand and dissect malicious software. Anti-debugging aims to make debugging more difficult, time-consuming, or outright impossible. For instance, it is quite common in malicious payloads to check for debuggers in various levels (e.g., SharedUserData.KernelDebugger) by checking ports or debugging objects (e.g., processDebugPort, processDebugObject), interrupts, and hardware breakpoints (e.g., Interrupt_0x2d, SystemKernelDebuggerInformation) [35, 43, 71]. LASE is an event-driven engine that relies on filter driver design to achieve its design goals. It does not require injecting the debugging agent into the context of target processes to record activities, leaving almost no traces for anti-debugging checks initiated by the malicious process.

Resource Profiling Checks. Resource profiling checks complement anti-debugging and VM checks [71]. For instance, checking the Windows data structure for processes (e.g., process_enum) [1, 92], or information on Disk (disk_size_getdiskfreespace, disk_size_wmi) [90] allows adversaries to identify sandboxes or virtual environments [1, 35]. Since LASE is deployed in a baremetal environment, these checks will not likely assist adversaries.

3.4 Comparison with Other Open-Source Forensics Tools.

As the last comparison, we analyzed publicly available forensics tools by comparing their architectural details, recorded artifacts, and fingerprintable details that could impact the effectiveness of these engines.

Selection Metrics. We evaluated forensics engines using three key criteria for a head-to-head comparison: the engine (1) must be open-source, (2) must be compatible with Windows operating systems, and (3) must be installed and operate without compilation or fundamental deployment errors. Our initial analysis included 10 engines, from which we selected six for this evaluation following these metrics.

Analysis. Table 2 illustrates the summary of the analysis. We observed that user-mode API hooking is still a common practice. For instance, BestEDR [126], OSquery [101], Wazuh [123], BlueSpawn [51] are all developed on top of API hooking or standard application logging mechanisms. Implementing this design strategy results in easier evasion and tampering since the forensic engine leaves detectable traces. Furthermore, tampering or terminating the engine is quite straightforward when the artifact recording engine operates at the same privilege level as other processes. Whids [105] approximates LASE in terms of gaining visibility by augmenting Microsoft ETW. However, the log processing and the scheduling module reside in the user space, making the tool vulnerable to evasion and fingerprinting. As previously noted, ETW was initially developed for performance monitoring, and minimizing artifacts was not a primary design consideration. Furthermore, it is not designed to capture specific run-time security events, such as remote code injection, that can have significant security consequences. That said, LASE offers system-wide visibility over processes, threads, and I/O activities while maintaining all functionalities (e.g., logging, monitoring tasks) within the kernel. This can potentially make LASE a more generalizable solution for the same problem space.

3.5 Collecting Run-time Artifacts

In this section, we discuss the format and details of the recorded artifacts. To this end, we provide an example of a macro-based malware sample where the malicious code loads and compiles visual basic code. Table 3 depicts the low-level temporal artifacts collected by LASE in this attack. The sequence of commands provided illustrates a multi-faceted approach used by the macro-based malware to infiltrate a system, execute malicious payloads, and maintain persistence. In particular, the malicious code uses Dynamic Data Exchange (DDE) to pass commands to Excel, a method known to execute code within an Excel document. The subsequent command (line 2) indicates the use of embedded mode, likely to run a macro embedded within the Excel document.

The repeated use of ‘eqnedt32.exe -Embedding’ in the trace suggests the exploitation of vulnerabilities in the Equation Editor. Older versions of EQNEDT32.EXE have known security flaws that can be exploited to execute arbitrary code [31, 42]. Lines 14 to 25 show that the malware incorporates Windows command prompt and script hosts such as wscript.exe and

cscript.exe to execute scripts. The filesystem activity shows that these scripts were used to download additional payloads (i.e., xx.vbs, Podaliri4.exe), modify system settings, and establish persistence. Using commands such as ‘WmiPrvSE.exe -secured -Embedding’ invokes Windows Management Instrumentation (WMI) to perform system management tasks, including creating persistence and gathering system information. The artifact also shows that the malware uses Alternate Data Streams (ADS), a technique to hide malicious scripts within legitimate files, making them harder to detect. This multi-stage attack resulted in the launch of the final process, i.e., Podaliri4.exe, which delivered ransomware to the system. Figure 2 depicts the attack tree generated from the data collected.

4 CASE STUDIES

LASE is designed to be portable, low artifact, and robust against common evasion mechanisms. In this section, we provide two case studies to illustrate how the engine can achieve these goals under different settings and deployment scenarios. In case study 1, we deployed LASE in a baremetal malware analysis environment, aiming to address common fingerprinting techniques in the wild. The analysis is based on 79,544 malware samples. In the second case study, we deploy LASE as an in-cloud deception-based threat intelligence service with the intent to collect real-world artifacts on how adversaries exploit vulnerable machines and how they use those hijacked.

4.1 Case Study 1: A Baremetal-assisted Analysis Infrastructure

Virtualized environments have been used significantly in prior research [24, 57, 60, 119] to analyze malicious code due to the ease of deployment, scalability, isolation, and hardware utilization. However, this approach often comes with important visibility costs. In particular, environmentally sensitive samples launch several forms of fingerprinting checks to analyze the target host. They rarely load their actual malicious payloads in these environments, making analysis and reverse engineering a challenging task.

In this case study, we study the feasibility of LASE as a low-artifact engine and build a scalable baremetal analysis environment to collect artifacts about modern malware payloads. The core motivation behind this case study was to answer the question of how to build threat monitoring platforms that can satisfy common fingerprinting techniques often used in modern malware samples. We started by asking how much visibility could be gained by running the evasive samples in the baremetal analysis environment.

This experiment was performed by executing 79,544 malware samples from across 941 malware families and analyzing the results. Table 4 shows the distribution of malware samples across different families for the 40,050 labels obtained. Each sample was executed on both the virtual and baremetal machines for five (5) minutes [66], during which run-time behavior artifacts were collected. The infrastructure used is equipped with high-bandwidth network switches to enable transferring Windows 10 images to the device in each run. That is, in each run, a new LASE-enabled OS image was distributed and loaded into each machine. The baremetal machines had similar processing, memory, and storage specifications.

Figure 3 shows the summary of our analysis. While LASE records 45 various low-level OS operations, we have provided six major operations to show the run-time behavior of a given sample. As shown, while there are samples with a similar number of key operations in both environments, there are still a large number of samples that show significantly more activities in baremetal compared to virtualized environments. For instance, the number of write operations (Figure 3b) is almost 64% more than the ones in a virtualized environment for 40,050 samples. The difference in the type of dropped payloads in the two environments suggests that the malware might have taken different execution paths in the two environments. We noticed

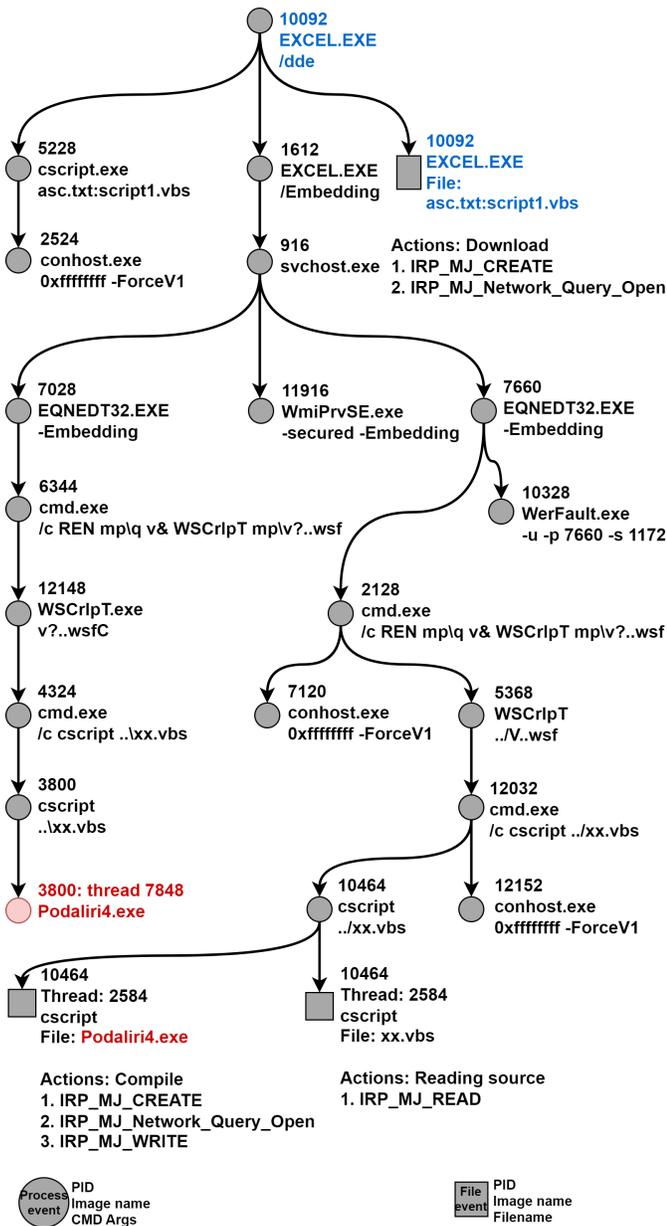


Figure 2: The collected artifacts for a micro-based malware attack. The malicious process invokes 15 processes in the background to launch a successful attack.

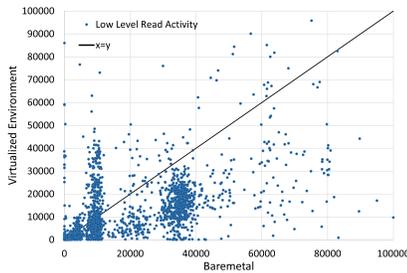
Table 3: LASE artifacts for a multi-stage macro-malware attack (md5:7ccf88c0bbe3b29bf19d877c4596a8d4). The malware incorporates several techniques for fetching malicious payload, information grabbing, persistence, and camouflage.

#	Operation	Time	Global #	PPID	PID	TID	Image path	Command-line args	File path
1	Pr Create	20:51:45:628	183668	5480	10092	0	%MSOffice%\EXCEL.EXE	/dde	
2	Pr Create	20:51:48:141	205076	10092	1612	0	%MSOffice%\EXCEL.EXE	/Embedding	
3	Pr Create	20:52:16:301	346364	916	7028	0	%MSOfficeCommon%\eqnedt32.exe	-Embedding	
4	Pr Create	20:52:16:579	348340	7028	6344	0	%SysWOW64%\cmd.exe	/c REN mp\q v& WSCripT mp\?.wsf C	
5	Pr Create	20:52:16:668	350587	6344	12148	0	%SysWOW64%\wscript.exe	%TEMP%\?.wsf C	
6	Pr Create	20:52:17:084	358524	12148	4324	0	%SysWOW64%\cmd.exe	/c csript %TEMP%\xx.vbs	
7	Pr Create	20:52:17:149	359955	4324	3800	0	%SysWOW64%\cscript.exe	%Temp%\xx.vbs	
8	Pr Exit	20:52:17:520	376991	916	7028	0	%MSOfficeCommon%\eqnedt32.exe		
9	Pr Exit	20:52:18:638	380165	10092	1612	0	%MSOffice%\EXCEL.EXE		
10	Pr Create	20:52:19:456	381227	916	11916	0	%SysWOW64%\wbem\WmiPrvSE.exe	-secured -Embedding	
11	Pr Create	20:53:05:751	407732	10092	5228	0	%SysWOW64%\cscript.exe	C:\programdata\asc.txt:script1.vbs	
12	Pr Exit	20:57:10:385	589717	5480	10092	0	%MSOffice%\EXCEL.EXE		
13	Pr Create	20:57:42:131	666687	916	7660	0	%MSOfficeCommon%\eqnedt32.exe	-Embedding	
14	Pr Create	20:57:42:305	668655	7660	2128	0	%SysWOW64%\cmd.exe	/c REN mp\q v& WSCripT mp\?.wsf C	
15	Pr Create	20:57:42:358	671600	2128	5368	0	%SysWOW64%\wscript.exe	%Temp%\?.wsf C	
16	Pr Create	20:57:42:399	675850	7660	10328	0	%SysWOW64%\WerFault.exe	-u -p 7660 -s 1172	
17	Pr Create	20:57:42:474	678590	5368	12032	0	%SysWOW64%\cmd.exe	/c csript %Temp%\xx.vbs	
18	Pr Create	20:57:42:516	679047	12032	10464	0	%SysWOW64%\cscript.exe	%Temp%\xx.vbs	
19	Ld Image	20:57:42:527	679237	12032	10464	0	%SysWOW64%\cscript.exe		%SysWOW64%\bcryptprimitives.dll
20	Ld Image	20:57:43:645	705589	12032	10464	0	%SysWOW64%\cscript.exe		%SysWOW64%\winhttp.dll
21	Ld Image	20:57:43:652	705594	12032	10464	0	%SysWOW64%\cscript.exe		%SysWOW64%\mswsock.dll
22	Tr Create	20:57:44:210	707985	12032	10464	2844	%SysWOW64%\cscript.exe		
23	Ld Image	20:57:44:226	708195	12032	10464	0	%SysWOW64%\cscript.exe		%SysWOW64%\urlmon.dll
24	Ld Image	20:57:44:232	708207	12032	10464	0	%SysWOW64%\cscript.exe		%SysWOW64%\virtdisk.dll
25	Tr Exit	20:57:44:249	708455	12032	10464	2844	%SysWOW64%\cscript.exe		

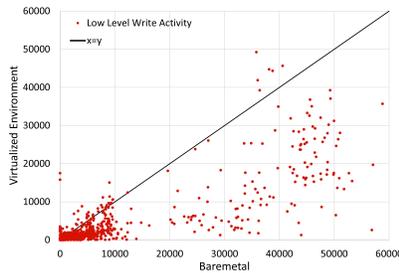
(a) LASE artifacts on process activities.

#	Operation	Time	Duration	Global #	PID	TID	Image path	File Path
1	IRP_Create	20:51:48:636	69	183856	10092	1504	%MSOffice%\EXCEL.EXE	C:\Users\grace\Downloads\ORDER SHEET & SPEC.xlsm
2	IRP_Write	20:51:48:590	1757	211615	10092	3620	%MSOffice%\EXCEL.EXE	C:\Users\grace\AppData\Local\Temp\DED9E0FE.xlsm
3	IRP_Write	20:51:48:755	448	213650	1612	12240	%MSOffice%\EXCEL.EXE	C:\Users\grace\AppData\Local\Temp\560D285B.emf
4	IRP_Write	20:52:13:024	501	296839	10092	1504	%MSOffice%\EXCEL.EXE	C:\Users\grace\AppData\Local\Microsoft\...\1983A0E7.png
5	IRP_Write	20:52:16:156	468	345357	10092	1504	%MSOffice%\EXCEL.EXE	C:\Users\grace\AppData\Local\Microsoft\Windows\...\1959A28D.emf
6	IRP_Write	20:52:16:230	886	345950	10092	1504	%MSOffice%\EXCEL.EXE	C:\Users\grace\AppData\Local\Temp\q
7	IRP_Write	20:52:16:270	551	346113	10092	1504	%MSOffice%\EXCEL.EXE	C:\Users\grace\AppData\Local\Temp\xx
8	IRP_Set_Information	20:52:16:932	748	355451	12148	8056	%SysWOW64%\wscript.exe	C:\Users\grace\AppData\Local\Temp\xx
9	IRP_Close	20:52:16:933	17	355455	12148	8056	%SysWOW64%\wscript.exe	C:\Users\grace\AppData\Local\Temp\xx.vbs
10	IRP_Write	20:53:05:692	385	407107	10092	1504	%MSOffice%\EXCEL.EXE	C:\ProgramData\asc.txt:script1.vbs
11	IRP_Read	20:57:42:541	73	679583	10464	2584	%SysWOW64%\cscript.exe	C:\Users\grace\AppData\Local\Temp\xx.vbs
12	IRP_Write	20:57:44:237	3432	708409	10464	2844	%SysWOW64%\cscript.exe	C:\ProgramData\Podaliri4.exe
13	IRP_Close	20:57:44:248	20	708591	10464	2844	%SysWOW64%\cscript.exe	C:\ProgramData\Podaliri4.exe

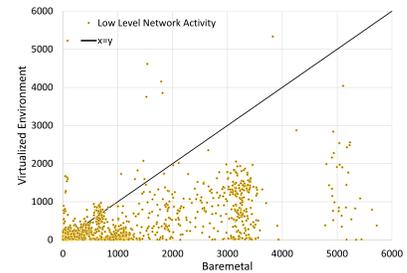
(b) LASE artifacts on filesystem activities.



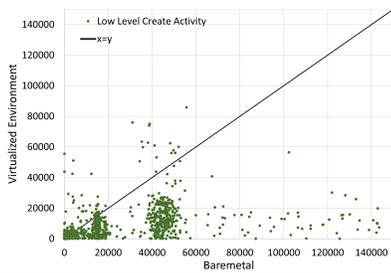
(a) Read Activity



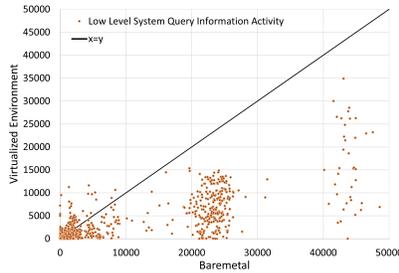
(b) Write Activity



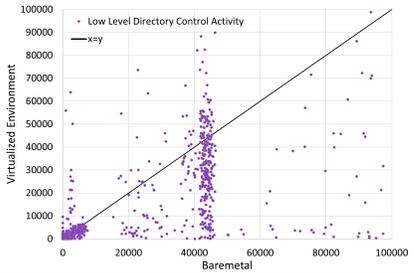
(c) Cached-File Open Activity



(d) File Create Activity



(e) System Query Information Activity



(f) Directory Control Activity

Figure 3: File system activities in LASE-enabled machines in baremetal and virtual environments. The results show that malware samples manifest more activities in the baremetal environment, resulting in more dropped files and execution payloads.

the download of exe files, as a result of launching the malware, had a significant decrease (93.9%) in the virtualized environment. JavaScript files (JS) are the second most common type

of malware payload downloaded, accounting for over 12.68% of all payloads. This is most likely due to the growing popularity of web-based attacks, as JS is the most often used scripting

Table 4: A subset of malware samples with the corresponding labels analyzed by LASE-enabled threat analysis platform. The dataset contains four major malware types, 941 malware families, and 3,596 malware variants.

Malware	Families-Occurrences	Variants
Ransomware	64 - 472 (1.18%)	
StopCrypt	253 (0.63%)	48
Gandcrab	23 (0.06%)	13
Lockbit	20 (0.05%)	10
Hive	19 (0.05%)	6
MedusaLocker	15 (0.04%)	1
Filecoder	14 (0.03%)	8
Others	58 - 128 (0.32%)	71
RAT	602 - 31,158 (77.79%)	
Sabsik	3,474 (8.67%)	8
Berbew	3,116 (7.78%)	5
AgentTesla	2,362 (5.90%)	457
Wacatac	2,009 (5.01%)	3
Vindor	1,175 (2.93%)	3
RpcDcom based	1,069 (2.67%)	1
Others	596 - 17,953 (44.83%)	2,495
PUP	163 - 1,322 (3.30%)	
Zbot	92 (0.23%)	8
Infostealer	83 (0.21%)	1
KuaiZip	80 (0.20%)	1
AutoKMS	75 (0.19%)	3
Ymacco	72 (0.18%)	16
DarkStealer	67 (0.17%)	2
Others	157 - 853 (2.12%)	186
Self Replicating Malware	112 - 7,098 (17.72%)	
Sfone	982 (2.45%)	2
Ganelp	759 (1.89%)	7
Viking	752 (1.87%)	16
Xolxo	707 (1.76%)	1
Autorun	595 (1.48%)	13
Vobfus	492 (1.23%)	50
Others	106 - 2,811 (7.02%)	161
Total	941 - 40,050	3,596

language on websites [27]. PDF documents are the most common type of non-executable payload downloaded, accounting for over 30% of all non-executable document payloads. The prevalence of PDFs as a document-sharing format is a probable cause, as they can serve as a vehicle for embedding and executing malicious code upon opening [50, 78]. Table 7 in the appendix shows a more comprehensive version of the dropped files in the experiments.

We performed an analysis of the fingerprinting checks used by samples in the dataset that cause the differences in the execution traces of LASE-enabled machines in baremetal and virtual environments. Our analysis shows that 1,187 samples from at least five malware families call Windows Management instrumentation API to get the actual hardware-level driver information before loading their actual payloads. Direct access to CPU clocks, bios, and timing checks have also been seen across various malware families in the test, as shown in table 5.

An immediate conclusion from this study is that a low-artifact analysis environment that can satisfy the hardware-based fingerprinting is critical to achieving more visibility in today’s attack landscape. That said, a baremetal-assisted solution should not be considered as an alternative solution to the current

Table 5: A subset of environment fingerprinting checks satisfied by LASE, compared to other core technologies. ✓=Satisfied, ✗=Failed

FP Check	Samples	Baremetal-Based	VM-Based	Top Families
calls-wmi	1,187	✓	✗	AgentTesla, Sabsik, Redline, Leone, Znyon
direct-cpu-clock-access	9,254	✓	✗	AgentTesla, Sabsik, Wacatac, Woreflint, FormBook
checks-bios	6,859	✓	✗	AgentTesla, Leone, Znyon, Formbook, Wacatac
GetTickCount	1,288	✓	✗	AgentTesla, Sabsik, Woreflint, Wacatac, FormBook

hypervisor-based solution that can offer significant reverse engineering flexibility. Instead, LASE-enabled malware analysis can serve as the first line of defense in malicious code analysis by satisfying fingerprinting checks that are impossible or very costly in other defense systems. Furthermore, the results also show the need for a more effective analysis environment for the research community that forces a trade-off between achieving more fine-grained visibility while leaving minimally detectable artifacts on the analysis machines.

4.2 Case Study 2: LASE for Distributed Threat Intelligence

Adversaries are continuously becoming more effective in launching low-profile attacks on critical systems. The question that arises is how to collect relevant evidence about these attacks and their strategies at the early stages of their operations. Lack of proper evidence on attack strategies can potentially put defenders in a highly disadvantaged position because understanding attack tactics and lateral movements becomes complex. While tools and services such as Event Tracing Windows (ETW) offer critical insights on systems, performing root cause analysis and formulating a proper response often requires more fine-grained data. In this case study, we developed a distributed deception-based threat intelligence infrastructure, using LASE to gather evidence on real-world attacks on an intentionally exposed service.

To run this experiment, we deployed LASE-enabled cloud instances on Windows 11 x64 with 2vCPUs. Run-time traces were collected via a high-privilege triage engine to manage collection, packing, and subsequent posting to our remote servers. By design, we made our vulnerable systems detectable by automated scanners that actively scan network-based services for weak Remote Desktop Protocol (RDP) credentials. Intrusions were detected by monitoring a subset of high-valued files; once touched, we received an immediate notification of a successful intrusion. In an effort to lessen the likelihood of encountering the same attack campaigns repeatedly, we reset the servers and assign a random IP address after each successful compromise.

We executed the experiments for a duration of 46 days, from December 19, 2023, to February 3, 2024. During this time, there were interactions with the environment from 398 unique IP addresses originating within 175 geographical locations. Figure 4 shows the distribution of successful attack origins during the experiment timeline. Our data shows that the exposed, vulnerable services were discovered on average 6 hours after publishing the instances, and successful exploitation occurred on average 13 hours after publishing the decoy server. We noticed that the high-valued files were opened in 32 of the experiments. Privilege escalation was inferred in three cases where we observed that files were written in the system32 folder (an operation requiring elevated permissions) using the logs generated by the forensic engine. In 51 experiments, at least one

file was retrieved from remote servers and dropped on the machines. In particular, LASE collected 196,774 dropped files, including 1,221 executable files, 4,586 DLL files, 401 shell scripts, 14,706 program source code files, 235 installers, and 438 digital certificates.

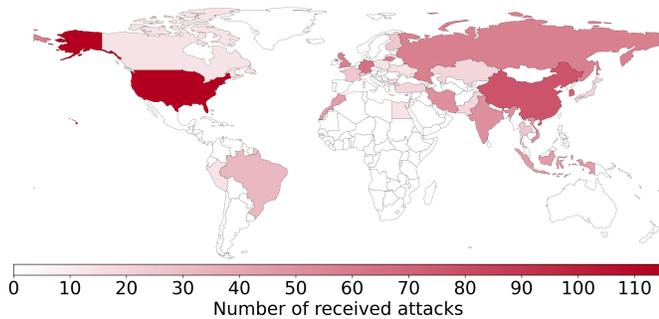


Figure 4: Distribution of the origin of successful attacks. While US-based attacks are the largest by volume, we observed that most of them are being abused as proxy nodes.

We identified attack strategies that allowed adversaries to monetize the compromised machine or run code, which facilitated further exploitation or persistence. In the following, we describe some of the attack strategies we observed.

Using the compromised machine as a proxy. Traffic jacking is the process of converting a compromised machine into a proxy server and renting the server’s bandwidth for adversarial purposes [80]. To initiate traffic jacking, we observed that the adversary dropped an executable that loaded 183 DLLs into the memory, including several cryptographic and socket communication libraries. Once loaded, the process established a connection with a remote server to schedule tasks. Forensics trace data from LASE shows that the parent process initiated hundreds of threads on the infected machine. The machine established a large number of TCP connections with hundreds of remote IP addresses to relay traffic. The adversary ran the malicious process in user-mode, and the forensic data captured by LASE did not indicate any traces of establishing virtual interfaces for relaying packets. Our analysis suggests that the compromised machine was indeed converted to an exit node in this experiment.

Account Creation and Modification. In 15 of the exploitations, we observed indications of account creation and privilege escalation attempts using `net user` and `net localgroup` to gain persistent access. Another popular operation was group enumeration (e.g., `WMIC Group where "SID = 'X' Get Name /Value — Find ""`) to identify group memberships, specifically for admins and remote desktop users to potentially manipulate user privileges.

Backup Erasure We observed a pattern across 12 incidents where Powershell scripts attempted to delete the Volume Shadow Copies and backup catalog. We observed that adversaries were using commands such as `'vssadmin delete shadows /all /quiet'`, `'wbadmin delete catalog -quiet'`, or `'wmic shadowcopy delete'` to prevent the recovery of data, making it more

difficult to restore from backup after an attack. This practice is common among adversaries to either force the victim to pay a ransom fee or destroy the system logging evidence to complicate incident analysis.

Persistence Threats. We observed several attempts among adversaries to stay persistent on the exploited machines. We also observed 5 cases where the adversaries tried to modify the password policies to set passwords to never expire (e.g., `net accounts /maxpwge:unlimited`), making it easier for adversaries to maintain access. In 10 incidents, we observed listing and manipulation queries where adversaries attempted to add or modify registry entries to hide user accounts from the login screen, making the accounts less noticeable to the casual user. In 5 cases, we observed no significant operation during the initial compromise. However, we observed setting scheduled tasks for future logins. In particular, we observed the use of `schtasks /create` to create tasks that execute at specific times or system events.

5 DISCUSSION

In this paper, we aimed to show that modern threat analysis highly depends on defining robust forensics engines that force a reliable trade-off between providing fine-grained behavioral insights and minimizing detectable artifacts. We posit that a solution that can achieve these two goals has several use cases on the defense side. As a first step in this direction, we defined two main application scenarios for LASE and showed how it can add a new lens to the analysis and detection of security incidents. We summarize our main findings along the following four points:

More Visibility over Malware Behavior: While prior work shows modern malware samples are getting less sensitive to virtualized environment [66], our data suggests that such environmental checks are still prevalent across different RAT, droppers, and PUP samples. We observed that 64% (22,021) of the samples with file write activities had at least 70% more dropped or modified files in LASE compared to cases in virtualized environments. We acknowledge that code coverage can have various definitions in the context of program analysis. However, the fact that more filesystem activities were observed in head-to-head experiments and a larger number of payloads were delivered on average suggests that the malware sample was designed to take a different path in environments close to real-world settings. In a broader context, this procedure can be augmented with static analysis approaches to systemically analyze malicious code more deeply. We believe systems similar to LASE should be more accessible to threat researchers and malware analysts for further investigations in malicious code analysis due to the scarcity of real-world run-time behavior datasets on baremetal systems.

LASE in a Broader Context. In this work, we aimed to answer how to improve behavioral visibility in modern evasive malicious code attacks while making them more robust to evasion. That is a critical question in malicious code analysis and the generalizability of the results to other analysis engines and sandboxes is important. For instance, in case study 1, we showed that our baremetal solution works more effectively in comparison to the virtualized environment in satisfying almost

all debugging and VM-based checks. We expect similar results if the experiments are conducted in other virtualization technologies (e.g., hypervisor-based). In particular, we observed that it is quite common across modern attacks to call Windows Management Instrumentation API in order to gain information on system hardware, applications, networks, devices, and other managed components. While manipulating the return values in WMI API to report fake responses is possible, it often leads to serious system crashes if the system has to operate for a longer period (case study 2) because the healthy operation of many system drivers depends on the responses received from the WMI API, and a fake response can easily cause a system crash. Unfortunately, hypervisor-based methods are still vulnerable to this fingerprinting method, although they offer significant freedom for other forms of analysis.

Unique Artifact Catalogs. The dataset we created in this experiment is the largest catalog of threat artifacts collected from a baremetal environment. While there are some available artifacts [6, 40, 57] that helped other work to have basic benchmarks, we are not aware of any large-scale run-time baremetal artifacts from prior work or commercial sandboxes on different forms of malware samples.

The high cost of building robust solutions and the scarcity of high-quality data can potentially impact the development of new data-driven defenses in this dynamic ecosystem where collecting evidence on emerging trends is critical. Lack of access to malicious data and the inherent complexity of defining a robust defense mechanism make it extremely difficult to establish a common ground for comparison. Given the ability to create a usable, timely, and longitudinal database of threat artifacts, we propose a set of research applications that would enable lower-cost threat intelligence. For instance, having longitudinal access to the artifacts of real-world attacks can be useful in testing new defense tools, studying attack evolution, and evaluating defense models with reference datasets.

Towards More Robust Cyber Deception Technologies. The ability to deploy LASE in different computing environments, including baremetal systems, end-points, and cloud environments, allows the deployment of a low-artifact deception-based threat intelligence infrastructure on the Internet. That is, deploying LASE-enabled machines will serve as distributed threat intelligence sensors, offering a unique opportunity to collect evidence on new attacks. Consider the recent attack on the MOVEit transfer service as an example [21]. A large-scale honeypot enabled by LASE could offer significant visibility on who first started the attack, how the attack took place, and a chronological order of the steps taken to perform the attack. As a next step, we plan to systematize the development of a vulnerability insertion module when a zero-day vulnerability is announced to measure the scope and scale, the evolution of the payloads and attacks, as well as the adversarial campaigns behind the attacks.

5.1 Limitations

Section 4 demonstrates that LASE achieves practical and useful detection results on a large, real-world dataset. Unfortunately, adversaries continuously observe defensive advances and adapt their attacks accordingly. In the following, we discuss the limitations of LASE and potential evasion strategies.

Delay Injection. In case study 1, injecting a significant delay before loading the malicious payload is another method to bypass detection. The experiments were designed to record the behavior for five minutes, so if the malicious code stays dormant for a long period of time, LASE cannot record actions associated with the malicious activity. Note that these limitations are not specific to LASE, and almost any dynamic analysis system may be impacted in some way by these evasion methods. Prior work has discussed this evasion mechanism. Note that incorporating these techniques can also potentially make detection easier in static analysis since these approaches require calling specific functions in the operating system. The presence of these mechanisms in the initial binary is being used currently by malware defense solutions to identify suspicious binaries in the wild.

Trusted Computing Model. LASE operates at the kernel level. If the target malicious code is a kernel-level attack, it can potentially thwart some of the hooks LASE uses to monitor runtime behavior. Analyzing high-privilege attacks is out of the scope of this paper. These attacks can be better analyzed in a hypervisor-based environment where debugging kernel-level code is possible. We assume that the trusted computing base includes the display module, OS kernel, and underlying software and hardware stack. Therefore, we can safely assume that the components of the system are free of malicious code and that normal user-based access control prevents attackers from running malicious code with superuser privileges. This is a fair assumption considering the fact that most malicious operations (e.g., malware attacks and software vulnerability checking) are often initiated in the user mode.

6 RELATED WORK

Security research has explored various ways to understand the behavior of evasive code, contextualize the behavior, and predict possible adversarial action. In this section, we discuss prior work, mainly focusing on building tools and systems to collect and analyze threat artifacts.

Forensics Services: Many of the prior works utilized the built-in Windows OS, Event Tracing for Windows (ETW), for their security analysis. For instance, Lee et al. [76] used the cursory data provided by ETW to detect APTs, thus lowering the overhead for log storage. Other approaches by Hassan et al. [49] look at the problem using static analysis to identify structures from built-in operating system audits. Rapsheet’s [46] proposed approach involves the detection of APTs by condensing the acquired low-level data into graphical representations and subsequently comparing the observed steps with the publicly-accessible MITRE ATT&CK knowledge base [122]. This knowledge base is curated by domain experts who analyze real-world APT attacks. In fact, ETW is designed to primarily serve as a telemetry collection tool that gathers high-level event data. Although this is helpful, its use in forensic analysis is constrained because of the level and granularity of reports. Additionally, ETW is susceptible to fingerprinting and can be disabled by malware [20, 99]. Contributions from Bakshi et al. [9] built upon ETW to augment its ability to collect event activity logs. However, its ability to collect data at the granularity needed for a detailed low-level analysis still falls short [97].

Lower-level artifacts such as monitoring file systems, networks, and processes are more useful in this context [97].

At the kernel level, data can be collected using global operating system-level abstractions such as processes, files, and network interactions [47]. There have been several works in which the implementation of lower-level artifact collection is centered around the file system alone [3, 23, 57, 58, 68, 108, 109]. This is advantageous for detecting changes to the filesystem, especially in cases of ransomware in which many files are modified during a relatively short window [57, 108, 118]. A further extension incorporating data collection at a process level is included in RwGuard [81] whenever it was determined that the interactions were suspicious. It approaches the problem by deploying a logger in the kernel space and adding a process monitoring aspect within the user space. Although the methodology is adept at collecting process information, it is still tightly coupled with their logger. Still in the kernel at a memory level, Shah et al. [110] and Kara [54] proposed systems for detecting malware that resides in computer memory. After executing malware, they extract a memory dump that is further converted into an image for processing using computer vision and machine learning.

One gap in this domain is that the core technology used in prior work to collect threat intelligence data either offered relevant but high-level insights about incidents or was primarily focused on one class of attacks (e.g., ransomware). To our knowledge, LASE is the only open-source solution that collects kernel-level data points (processes and threads, filesystem, registry) and generates GBs of data on the collected artifacts.

Sandbox and behavioral Analysis. A drawback for any system collecting data in this adversarial space is its susceptibility to being fingerprinted [29, 35, 100]. Advances in malware construction have seen the advent of anti-debugging and anti-sandboxing techniques to evade analysis and detection by security researchers trying to analyze malware behavior [30, 35]. These techniques are principally designed to make analysis time-consuming and tedious for researchers. It has been observed that malware behaves differently when it perceives that the environment is sandboxed for analysis or executed in a virtualized environment [8, 61, 66, 69, 71, 72, 117]. In some cases, the malware behaves benignly or does not execute if it detects a sandboxed environment [43, 92]. Proposals for anti-sandboxing techniques Liu et al. [72] build upon the works of Miramirkhani et al. [92] and Hu et al. [114] in creating believable system artifacts to mitigate system fingerprinting.

As a technique for sidestepping the issue of virtualized systems being detected, several works have proposed the use of physical machines for the collection of forensic data [61, 62, 114]. BareBox [61] uses a restoration method in which snapshots of memory and disks are used simultaneously with parallel operating systems. BareCloud [62] uses a distributed model of disks, restoring them after each execution. LO-PHI [114], like BareCloud uses machines that do not contain instrumentation for fingerprinting. LASE is designed to complement all these methods by being portable, allowing deployment of the LASE as an OS service in any computing environment.

Fingerprinting and Evasive Techniques. Resistance to profiling in malware is presented in different forms where malicious

authors develop a multitude of techniques to hide the true behavior of their applications whenever an analysis environment is detected. To this end, several studies [35, 38, 77, 98, 127] have focused on understanding the techniques used to evade analysis. Maffia et al. [77] investigate each technique, utilizing publicly available data to create an evasive program profiler to detect and circumvent evasive measures and collect statistical data on evasion methods. Similar research was carried out by Garollo et al. [35] to produce statistically significant results on the link between malware families, the commonality of evasion techniques, and modifications to evasion techniques within families. They investigated evasive techniques in legitimate software and found that they are employed less frequently. Classification of malware by their evasive behaviors is studied by Yin and Nunes [98, 127]. Nunes et al. [98] note that applications are classified as malicious simply based on evasive methods taken.

7 CONCLUSIONS

In this paper, we propose LASE, a low-artifact in-kernel forensics analysis system that aims to achieve improved visibility over the dynamics of the threat landscape. LASE is designed to optimize a balance between offering fine-grained visibility while minimizing detectable artifacts. We show these properties are critical to collecting all the relevant forensics events and conducting evidence-based threat characterization. We evaluated LASE by running two case studies. In the first case study, we integrated LASE in a baremetal analysis environment and analyzed more than 79K malware samples. We collected GBs of artifacts on modern malware threats. In the second case study, we deployed the system as a deception-based threat intelligence, aiming to collect real-world threat artifacts. Our analysis showed that LASE was successful in collecting artifacts about hundreds of binaries and shell scripts developed and executed by adversaries on the threat infrastructure.

REFERENCES

- [1] Impeding automated malware analysis with environment-sensitive malware. In *7th USENIX Workshop on Hot Topics in Security (HotSec 12)*, Bellevue, WA, August 2012. USENIX Association.
- [2] Iozone filesystem benchmark. www.iozone.org, 2024.
- [3] Muhammad Shabbir Abbasi, Harith Al-Sahaf, Masood Mansoori, and Ian Welch. Behavior-based ransomware classification: A particle swarm optimization wrapper-based approach for feature selection. *Applied Soft Computing*, 121:108744, 2022.
- [4] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *ACM Comput. Surv.*, 52(6), nov 2019.
- [5] Saleh Alzahrani, Yang Xiao, and Wei Sun. An analysis of conti ransomware leaked source codes. *IEEE Access*, 10:100178–100193, 2022.

- [6] Andrea Continella. ShieldFS: a self-healing, ransomware-aware filesystem. <http://shieldfs.necst.it/>, Accessed: 05-06-2024.
- [7] Giovanni Apruzzese, Pavel Laskov, and Johannes Schneider. Sok: Pragmatic assessment of machine learning for network intrusion detection. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 592–614. IEEE, 2023.
- [8] Erin Avllazagaj, Ziyun Zhu, Leyla Bilge, Davide Balzarotti, and Tudor Dumitras. When malware changed its mind: An empirical study of variable program behaviors in the real world. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3487–3504. USENIX Association, August 2021.
- [9] Akshay Bakshi, Tanish Sawant, Prasad Thakare, Azeed Dandawala, Manjesh K. Hanawal, and Atul Kabra. Improving threat detection capabilities in windows endpoints with osquery. In *2023 15th International Conference on COMMunication Systems & NETWORKS (COM-SNETS)*, pages 432–435, 2023.
- [10] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. Quincy: Detecting host-based code injection attacks in memory dumps. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pages 209–229. Springer, 2017.
- [11] Thomas Barabosch and Elmar Gerhards-Padilla. Host-based code injection attacks: A popular technique used by malware. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 8–17. IEEE, 2014.
- [12] RR Branco, GN Barbosa, and PD Neto. Scientific but not academical overview of malware anti-debugging. *Anti-Disassembly and Anti-VM Technologies, Black Hat USA*, 2012, 2012.
- [13] Doug Brubacher. Detours: Binary interception of win32 functions. In *Windows NT 3rd symposium (windows NT 3rd symposium)*, 1999.
- [14] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium, ROOTS*, New York, NY, USA, 2017. Association for Computing Machinery.
- [15] Gregory T Byrd and Michael J Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3):456–466, 1999.
- [16] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. Advanced or not? a comparative study of the use of anti-debugging and anti-vm techniques in generic and targeted malware. In *ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30-June 1, 2016, Proceedings 31*, pages 323–336. Springer, 2016.
- [17] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN)*, pages 177–186. IEEE, 2008.
- [18] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–411, 2018.
- [19] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. {Obfuscation-Resilient} executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3451–3468, 2021.
- [20] Adam Chester. Hiding your .net - etw. <https://blog.xpnsec.com/hiding-your-dotnet-etw/>, 2020.
- [21] Cisco Talos. Active exploitation of the MOVEit Transfer vulnerability by Clop ransomware group. <https://blog.talosintelligence.com/active-exploitation-of-moveit/>, Accessed: 06-30-2023.
- [22] Claudiu Teodorescu, Igor Korkin, Andrey Golchikov. Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors. <https://i.blackhat.com/EU-21/Wednesday/EU-21-Teodorescu-Veni-No-Vidi-No-Vici-Attacks-On-ETW-Blind-EDRs.pdf>, Accessed:01-08-2025.
- [23] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd annual conference on computer security applications*, pages 336–347, 2016.
- [24] Léo Cosserson, Louis Rilling, Matthieu Simonin, and Martin Quinson. Simulating the network environment of sandboxes to hide virtual machine introspection pauses. In *Proceedings of the 17th European Workshop on Systems Security, EuroSec '24*, page 1–7, New York, NY, USA, 2024. Association for Computing Machinery.
- [25] Crispin Cowan. Turing around the security problem. In *15th USENIX Security Symposium (USENIX Security 06)*. USENIX Association, 2006.
- [26] Cybrary. Windows Commands Most Used by Attackers. <https://www.cybrary.it/blog/windows-commands-used-attackers>, Accessed: 05-06-2024.
- [27] Kyle Daigle. Octoverse: The state of open source and rise of ai in 2023. <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/>, 2023.
- [28] Daniele Cono D’Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security*, 15:2750–2765, 2020.
- [29] Daniele Cono D’Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security*, 15:2750–2765, 2020.

- [30] Daniele Cono D’Elia, Lorenzo Invidia, Federico Palmaro, and Leonardo Querzoni. Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. *Digital Threats*, 3(2), feb 2022.
- [31] Waeland Elder. Automatic extraction of vulnerability information for security operators using gpt models. 2024.
- [32] Jason Franklin, Mark Luk, Jonathan M McCune, Arvind Seshadri, Adrian Perrig, and Leendert Van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review*, 42(3):83–92, 2008.
- [33] Zhuoqun Fu, Mingxuan Liu, Yue Qin, Jia Zhang, Yuan Zou, Qilei Yin, Qi Li, and Haixin Duan. Encrypted malware traffic detection via graph-based network analysis. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 495–509, 2022.
- [34] Hisham Shehata Galal, Yousef Bassyouni Mahdy, and Mohammed Ali Atiea. Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, 12:59–67, 2016.
- [35] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. A systematical and longitudinal study of evasive behaviors in windows malware. *Computers & Security*, 113:102550, 2022.
- [36] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 391–408, 2023.
- [37] Tal Garfinkel, Keith Adams, Andrew Warfield, Jason Franklin, et al. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS*, 2007.
- [38] Jiakuan Geng, Junfeng Wang, Zhiyang Fang, Yingjie Zhou, Di Wu, and Wenhan Ge. A survey of strategy-driven evasion methods for pe malware: Transformation, concealment, and attack. *Computers & Security*, 137:103595, 2024.
- [39] Shahram Ghandeharizadeh and Hieu Nguyen. Design, implementation, and evaluation of write-back policy with cache augmented data stores. *Proceedings of the VLDB Endowment*, 12(8):836–849, 2019.
- [40] Giorgio Severi. MALREC: Compact Full-Trace Malware Recording for Retrospective Deep Analysis. <https://giantpanda.gtisc.gatech.edu/malrec/dataset/>, Accessed: 05-06-2024.
- [41] Akul Goyal, Xueyuan Han, Gang Wang, and Adam Bates. Sometimes, you aren’t what you do: Mimicry attacks against provenance graph host intrusion detection systems. In *30th ISOC Network and Distributed System Security Symposium (NDSS’23), San Diego, CA, USA*, 2023.
- [42] Yeming Gu, Hui Shu, Pan Yang, and Rongkuan Ma. Minsib: Minimized static instrumentation for fuzzing binaries. *Computers & Security*, 122:102894, 2022.
- [43] Francis Guibernau and Ayelen Torello. ‘catch me if you can!’—detecting sandbox evasion techniques. *Proc. USENIX Assoc*, 2020.
- [44] Rajat Gupta, Lukas Patrick Dresel, Noah Spahn, Giovanni Vigna, Christopher Kruegel, and Taesoo Kim. Popcorn: Popping windows kernel drivers at scale. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 854–868, 2022.
- [45] Tran Hoang Hai, Vu Van Thieu, Tran Thai Duong, Hong Hoa Nguyen, and Eui-Nam Huh. A proposed new endpoint detection and response with image-based malware detection system. *IEEE Access*, 11:122859–122875, 2023.
- [46] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1172–1189. IEEE, 2020.
- [47] Lori Whippler Hollasch, Andrew Kim, Sameer Saiya, and Matt. File systems driver design guide. <https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/>, 2022.
- [48] Ashkan Hosseini. Ten process injection techniques: A technical survey of common and trending process injection techniques, endgame. <https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process>, 2018.
- [49] Hassaan Irshad, Gabriela Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Kyu Hyung Lee, Jignesh Patel, Somesh Jha, Yonghwi Kwon, Dongyan Xu, and Xiangyu Zhang. Trace: Enterprise-wide provenance tracking for real-time apt detection. *IEEE Transactions on Information Forensics and Security*, 16:4363–4376, 2021.
- [50] Maryam Issakhani, Princy Victor, Ali Tekeoglu, and Arash Habibi Lashkari. Pdf malware detection based on stacking learning. In *ICISSP*, pages 562–570, 2022.
- [51] Jacob Smith. BlueSpawn. <https://github.com/ION28/BLUESPAWN>, Accessed: 26-05-2024.
- [52] Justin Jones and Narasimha Shashidhar. Ransomware analysis and defense. *Journal of Colloid and Interface Science*, 374(1):45–53, 2012.
- [53] Rémi Jullian. In-depth formbook malware analysis – obfuscation and process injection. <https://www.stormshield.com/news/in-depth-formbook-malware-analysis-obfuscation-and-process-injection/>, 2023.
- [54] Ilker Kara. Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges. *Expert Systems with Applications*, 214:119133, 2023.
- [55] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. Api chaser: Anti-analysis resistant malware analyzer. In *Research in Attacks, Intrusions, and Defenses: 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings 16*, pages 123–143. Springer, 2013.

- [56] Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura, and Jun Miyoshi. Api chaser: Taint-assisted sandbox for evasive malware analysis. *Journal of Information Processing*, 27:297–314, 2019.
- [57] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. {UNVEIL}: A {Large-Scale}, automated approach to detecting ransomware. In *25th USENIX security symposium (USENIX Security 16)*, pages 757–772, 2016.
- [58] Amin Kharraz and Engin Kirda. Redemption: Real-time protection against ransomware at end-hosts. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*, pages 98–119. Springer, 2017.
- [59] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9–10, 2015, Proceedings 12*, pages 3–24. Springer, 2015.
- [60] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled ElWazeer, Daniel Buettner, and Rajeev Barua. Dynodet: Detecting dynamic obfuscation in malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 97–118, Cham, 2017. Springer International Publishing.
- [61] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412, 2011.
- [62] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, 2014.
- [63] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296, 2011.
- [64] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1691–1708, 2017.
- [65] Igor Korkin. Protected process light is not protected: Memoryranger fills the gap again. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 298–308. IEEE, 2021.
- [66] Alexander Kuechler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *Network and Distributed System Security (NDSS) Symposium, NDSS 21, February 2021*.
- [67] Lawrence Abrams. US govt offers \$10 million bounty for info on Clop ransomware. <https://www.bleepingcomputer.com/news/security/us-govt-offers-10-million-bounty-for-info-on-clop-ransomware/>, Accessed: 06-30-2023.
- [68] Seungkwang Lee, Nam su Jho, Doyoung Chung, Yousung Kang, and Myungchul Kim. Rcryptect: Real-time detection of cryptographic function in the user-space filesystem. *Computers & Security*, 112:102512, 2022.
- [69] Young Bi Lee, Jae Hyuk Suk, and Dong Hoon Lee. Bypassing anti-analysis of commercial protector methods using dbi tools. *IEEE Access*, 9:7655–7673, 2021.
- [70] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20–21, 2011. Proceedings 14*, pages 338–357. Springer, 2011.
- [71] Songsong Liu, Pengbin Feng, Shu Wang, Kun Sun, and Jiahao Cao. Enhancing malware analysis sandboxes with emulated user behavior. *Computers & Security*, 115:102613, 2022.
- [72] Songsong Liu, Pengbin Feng, Shu Wang, Kun Sun, and Jiahao Cao. Enhancing malware analysis sandboxes with emulated user behavior. *Computers & Security*, 115:102613, 2022.
- [73] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18–21, 2018*. The Internet Society, 2018.
- [74] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security*, 1:114–136, 2017.
- [75] R. Ma. Anomaly detection for linux system log, August 2020.
- [76] Shiqing Ma, Kyu Hyung Lee, Chung Hwan Kim, Junghwan Rhee, Xiangyu Zhang, and Dongyan Xu. Accurate, low cost and instrumentation-free security audit logging for windows. In *Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC '15*, page 401–410, New York, NY, USA, 2015. Association for Computing Machinery.
- [77] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. Longitudinal study of the prevalence of malware evasive techniques. *arXiv preprint arXiv:2112.11289*, 2021.
- [78] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Surveys (CSUR)*, 52(4):1–36, 2019.
- [79] Mohd Fadzli Marhusin, Henry Larkin, Chris Lokan, and David Cornforth. An evaluation of api calls hooking performance. In *2008 International Conference on Computational Intelligence and Security*, volume 1, pages 315–319. IEEE, 2008.

- [80] Naif Mehanna, Walter Rudametkin, Pierre Laperdrix, and Antoine Vastel. Free proxies unmasked: A vulnerability and longitudinal analysis of free proxy services. *arXiv preprint arXiv:2403.02445*, 2024.
- [81] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. Rvguard: A real-time detection system against cryptographic ransomware. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 114–136, Cham, 2018. Springer International Publishing.
- [82] Microsoft. Pssetcreateprocessnotifyroutineex function (ntddk.h). <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreateprocessnotifyroutineex>, 2022.
- [83] Microsoft. Pssetcreatethreadnotifyroutineex function (ntddk.h). <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetcreatethreadnotifyroutineex>, 2022.
- [84] Microsoft. Pssetloadimagenotifyroutineex function (ntddk.h). <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetloadimagenotifyroutineex>, 2022.
- [85] Microsoft. ntddk.h header. <https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/>, 2023.
- [86] Microsoft. Overview of Early Launch Anti-Malware. <https://learn.microsoft.com/en-us/windows-hardware/drivers/install/early-launch-antimalware>, Accessed: 10-15-2023.
- [87] Microsoft. Plug and Play Minor IRPs. <https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/plugin-and-play-minor-irps>, Accessed: 10-15-2023.
- [88] Microsoft Learn Challenge. Operations That Can Be IRP-Based or Fast I/O. <https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/operations-that-can-be-irp-based-or-fast-i-o>, Accessed:01-08-2025.
- [89] Microsoft Learn Challenge. Synchronous and Asynchronous I/O. <https://learn.microsoft.com/en-us/windows/win32/fileio/synchronous-and-asynchronous-i-o>, Accessed:01-08-2025.
- [90] Microsoft Learn Challenge. Windows Management Instrumentation API. <https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page>, Accessed:01-08-2025.
- [91] Aleksandar Milenkoski. *ELAM: The Windows Defender ELAM Driver*. PhD thesis, ERNW Enno Rey Netzwerke GmbH, 2019.
- [92] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024. IEEE, 2017.
- [93] Dimitris Mitropoulos and Diomidis Spinellis. Fatal injection: A survey of modern code injection attack countermeasures. *PeerJ Computer Science*, 3:e136, 2017.
- [94] Abhijit Mohanta, Anoop Saldanha, Abhijit Mohanta, and Anoop Saldanha. Code injection, process hollowing, and api hooking. *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*, pages 267–329, 2020.
- [95] KA Monnappa. Detecting deceptive process hollowing techniques usind hollowfind volatility plugin. <https://cysinfo.com/detecting-deceptive-hollowing-techniques/>, 2017.
- [96] Nedim Sabic. Fibratus. <https://github.com/rabbitstack/fibratus>, Accessed: 26-05-2024.
- [97] Matthew Nunes, Pete Burnap, Omer Rana, Philipp Reinecke, and Kaelon Lloyd. Getting to the root of the problem: A detailed comparison of kernel and user level data for dynamic malware analysis. *Journal of Information Security and Applications*, 48:102365, 2019.
- [98] Matthew Nunes, Pete Burnap, Philipp Reinecke, and Kaelon Lloyd. Bane or boon: Measuring the effect of evasive malware on system call classifiers. *Journal of Information Security and Applications*, 67:103202, 2022.
- [99] Odzhan. Another method of bypassing etw and process injection via etw registration entries. <https://modexp.wordpress.com/2020/04/08/red-teams-etw/>, 2020.
- [100] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48, 2019.
- [101] osquery. osquery. <https://github.com/osquery/osquery>, Accessed: 26-05-2024.
- [102] François Plumerault and Baptiste David. Dbi, debuggers, vm: gotta catch them all: How to escape or fool debuggers with internal architecture cpu flaws? *Journal of Computer Virology and Hacking Techniques*, 17:105–117, 2021.
- [103] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D’Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. Measuring and defeating anti-instrumentation-equipped malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pages 73–96. Springer, 2017.
- [104] David B Probert. Windows kernel internals cache manager. *Microsoft Corporation*, page 48, 2010.
- [105] RawSec. Whids. <https://github.com/0xrawsec/whids>, Accessed: 26-05-2024.
- [106] Donald Ray and Jay Ligatti. Defining code-injection attacks. *Acm Sigplan Notices*, 47(1):179–190, 2012.

- [107] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows internals, part 2*. Pearson Education, 2012.
- [108] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. Cryptolock (and drop it): stopping ransomware attacks on user data. In *2016 IEEE 36th international conference on distributed computing systems (ICDCS)*, pages 303–312. IEEE, 2016.
- [109] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C Lupu. Automated Dynamic Analysis of Ransomware: Benefits, Limitations and use for Detection. *arXiv preprint arXiv:1609.03020*, 2016.
- [110] Syed Shakir Hameed Shah, Abd Rahim Ahmad, Norziana Jamil, and Atta ur Rehman Khan. Memory forensics-based malware detection using computer vision and machine learning. *Electronics*, 11(16):2579, 2022.
- [111] Syed Zainudeen Mohd Shaid and Mohd Aizaini Maarof. In memory detection of windows api call hooking technique. In *2015 International conference on computer, communications, and control technology (I4CT)*, pages 294–298. IEEE, 2015.
- [112] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.
- [113] David A Solomon, Mark E Russinovich, and Alex Ionescu. *Windows internals*. Microsoft Press, 2009.
- [114] Chad Spensky, Hongyi Hu, and Kevin Leach. Lo-phi: Low-observable physical host instrumentation for malware analysis. In *NDSS*, 2016.
- [115] Jerre Starink, Marieke Huisman, Andreas Peter, and Andrea Continella. Understanding and measuring inter-process code injection in windows malware. In *19th International Conference on Security and Privacy in Communication Networks, SecureComm 2023*, 2023.
- [116] Steven White, Kent Sharkey, David Coulter, Dan Mabee, Drew Batchelor, Mike Jacobs, Michael Satran. About WMI. <https://learn.microsoft.com/en-us/windows/win32/wmisdk/about-wmi>, Accessed: 10-15-2023.
- [117] Ming-Kung Sun, Mao-Jie Lin, Michael Chang, Chi-Sung Lai, and Hui-Tang Lin. Malware virtualization-resistant behavior detection. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 912–917, 2011.
- [118] Ruimin Sun, Marcus Botacin, Nikolaos Sapountzis, Xiaoyong Yuan, Matt Bishop, Donald E. Porter, Xiaolin Li, Andre Gregio, and Daniela Oliveira. A praise for defensive programming: Leveraging uncertainty for effective malware mitigation. *IEEE Transactions on Dependable and Secure Computing*, 19(1):353–369, 2022.
- [119] Yixin Sun, Kangkook Jee, Suphannee Sivakorn, Zhichun Li, Cristian Lumezanu, Lauri Korts-Parn, Zhenyu Wu, Junghwan Rhee, Chung Hwan Kim, Mung Chiang, and Prateek Mittal. Detecting Malware Injection with Program-DNS Behavior. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 552–568, Los Alamitos, CA, USA, September 2020. IEEE Computer Society.
- [120] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. Mining on someone else’s dime: Mitigating covert mining operations in clouds and enterprises. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 287–310. Springer, 2017.
- [121] Dmitry Tanana. Behavior-based detection of cryptojacking malware. In *2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT)*, pages 0543–0545. IEEE, 2020.
- [122] The MITRE Corporation. MITRE ATTACK. <https://attack.mitre.org/>, Accessed: 09-20-2023.
- [123] wazuh. wazuh. <https://github.com/wazuh>, Accessed: 26-05-2024.
- [124] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.
- [125] Yarden Shafir. ETW internals for security research and forensics. <https://blog.trailofbits.com/2023/11/22/etw-internals-for-security-research-and-forensics/>, Accessed:01-08-2025.
- [126] Yazid. BEOTM. <https://github.com/Xacone/BestEdrOfTheMarket>, Accessed: 26-05-2024.
- [127] Haikuo Yin, Brandon Lou, and Peter Reiher. A method for summarizing and classifying evasive malware. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 455–470, 2023.
- [128] Pavel Yosifovich, David A Solomon, Mark E Russinovich, and Alex Ionescu. *Windows Internals: System architecture, processes, threads, memory management, and more, Part 1*. Microsoft Press, 2017.
- [129] Michael Zipperle, Florian Gottwalt, Elizabeth Chang, and Tharam Dillon. Provenance-based intrusion detection systems: A survey. *ACM Computing Surveys*, 55(7):1–36, 2022.

Table 6: Distribution of a subset of 2.6 TBs of dataset submitted as artifacts in this submission. This represents 2% of the artifacts. Each log contains traces by LASE from baremetal and the virtual environment.

Malware	Occurrences
Ransomware	1 (0.13%)
StopCrypt	1 (0.13%)
RAT	669 (83.83%)
Banload	1 (0.13%)
CoinMiner	35 (4.39%)
CryptInject	401 (50.25%)
Emotet	3 (0.38%)
Farlii	1 (0.13%)
Glupteba	1 (0.13%)
Killav	2 (0.25%)
Musecador	4 (0.50%)
Phonzy	1 (0.13%)
Plyromt	1 (0.13%)
Pykspa	1 (0.13%)
Sabsik	69 (8.65%)
Trickbot	1 (0.13%)
Wabot	18 (2.26%)
Wacatac	2 (0.25%)
Ymacco	128 (16.04%)
PUP	2 (0.25%)
GameBox	2 (0.25%)
Self Replicating Malware	126 (15.79%)
Andriod	5 (0.63%)
Cambot	6 (0.75%)
Canbis	5 (0.63%)
Floxfif	1 (0.13%)
Gogo	10 (1.25%)
Morefi	5 (0.63%)
Neshta	5 (0.63%)
Nuqel	1 (0.13%)
Pidgeon	36 (4.51%)
Ramnit	1 (0.13%)
Sality	1 (0.13%)
Sfone	22 (2.76%)
Shodi	11 (1.38%)
Sivis	4 (0.50%)
Small	2 (0.25%)
Viking	1 (0.13%)
Virus	1 (0.13%)
Vobfus	9 (1.13%)
Total	798

Table 7: Top dropped extensions in the two experiments. A malware sample tends to drop more files in LASE when compared to a virtualized environment.

Extension	LASE	Virtualized Environment	Difference (# — %)	Description
Archive	12,309	9,195		
cab	3,504	1,989	1,515 — 76.2%	Microsoft cabinet file - compressed archive
pak	8,805	7,206	1,599 — 22.2%	Game Archive/Skype Language Pack
Binary	3,132,3425	1,402,507		
api	22,594	7,480	15,114 — 202.1%	Adobe Acrobat Plug-in/WebObjects API File
appx	2,553	557	1,996 — 358.3%	Microsoft Windows 8 app package
bin	4,891	2,053	2,838 — 138.2%	Binary archive
cur	6,772	2,070	4,702 — 227.1%	Windows custom cursor
dat	5,600	3,156	2,444 — 77.4%	General data file
dll	333,946	71,361	262,585 — 368.0%	Windows Dynamic Linked Library
exe	2,160,744	1,114,617	1,046,127 — 93.9%	Executable file
js	586,494	197,952	388,542 — 196.3%	JavaScript file
pmp	2,674	917	1,757 — 191.6%	AutoCAD plot model parameter file
pyd	1,278	789	489 — 62.0%	Windows binary that contains compiled Python code
sequ	4,879	1,555	3,324 — 213.8%	Adobe Acrobat Batch sequence
Document	74,340	34,706		
html	32,586	13380	19206 — 143.5%	Hypertext Markup Language Document
mpp	2,888	924	1,964 — 212.6%	Microsoft Project document
pdf	19,595	13,321	6,274 — 47.1%	Portable Document Format Document
rtf	1,355	910	445 — 48.9%	Rich Text Format document
x3d	4,662	1,391	3,271 — 235.2%	X3D (XML) scene to represent 3D graphics
xml	13,254	4,780	8,474 — 177.3%	Extensible Markup Language File
Font	30,719	10,164		
eot	2,697	871	1,826 — 209.6%	Microsoft Embedded OpenType font
otf	25,360	7,586	17,774 — 234.3%	Font file
woff	2,662	1,707	955 — 55.9%	Web Open Font Format
Image	735,210	306,284		
bmp	1,383	169	1,214 — 718.3%	Bitmap image
gif	75,298	24,718	50,580 — 204.6%	Bitmap image
ico	28,806	18,479	10,327 — 55.9%	Icon file
jpg	16,865	5,462	11,403 — 208.8%	JPEG Bitmap image
png	266,327	136,027	130,300 — 95.8%	Portable Network Graphics image
svg	346,531	121,429	225,102 — 185.4%	Scalable Vector Graphic
Raw	328,714	106,443		
aapp	35,181	10,773	24,408 — 226.6%	Adobe Acrobat AcroApp script
css	46,271	15,218	31,053 — 204.1%	Cascading style sheets
dic	1,873	627	1,246 — 198.7%	Microsoft Office custom dictionary
ini	13,668	4,937	8,731 — 176.8%	Initialization file
json	21,681	1,430	20,251 — 1416.2%	JavaScript Object Notation
lnk	1,160	912	248 — 27.2%	Windows shortcut
log	15,285	7,987	7,298 — 91.4%	General log
tmp	84,564	34,789	49,775 — 143.1%	General temporary file
txt	109,031	29,770	79,261 — 266.2%	Plain text document
Audio	2,924	2,843		
wav	2,924	81	2,843 — 3509.9%	Windows waveform sound

Table 8: I/O Request Packets (IRPs) Types supported by LASE

Major	Minor	Description	Minor	Description
Major				
Standard				
IRP_MJ_CREATE	IRP_MN_REGINFO	Open object handle		Driver registry path
IRP_MJ_CREATE_NAMED_PIPE	IRP_MN_QUERY_DIRECTORY	Create or open named pipe		Get Files in directory
IRP_MJ_CLOSE	IRP_MN_NOTIFY_CHANGE_DIRECTORY	Close open handle		Notify directory changes
IRP_MJ_READ	IRP_MN_USER_FS_REQUEST	Data read		Filesystem requests
IRP_MJ_WRITE	IRP_MN_MOUNT_VOLUME	Data write		Request volume mount
IRP_MJ_QUERY_INFORMATION	IRP_MN_VERIFY_VOLUME	Query object information		Mounted volume integrity check
IRP_MJ_SET_INFORMATION	IRP_MN_LOAD_FILE_SYSTEM	Modify object information		Load driver from filesystem
IRP_MJ_SET_EA	IRP_MN_TRACK_LINK	Query object extended attributes		Watch for device events
IRP_MJ_FLUSH_BUFFERS	IRP_MN_LOCK	Set object extended attributes		Acquire file lock
IRP_MJ_QUERY_VOLUME_INFORMATION	IRP_MN_UNLOCK_SINGLE	Flush cached data to disk		Release file lock
IRP_MJ_SET_VOLUME_INFORMATION	IRP_MN_UNLOCK_ALL	Query volume information		Release all file locks
IRP_MJ_DIRECTORY_CONTROL	IRP_MN_UNLOCK_ALL_BY_KEY	Modify volume property		Release locks associated with key
IRP_MJ_FILE_SYSTEM_CONTROL	IRP_MN_NORMAL	Directory management operations		Perform device management tasks via messages
IRP_MJ_DEVICE_CONTROL	IRP_MN_DPC	Advance file system operations		Deferred procedure calls (DPC)
IRP_MJ_INTERNAL_DEVICE_CONTROL	IRP_MN_MDL	Midware between device and kernel		Manage memory descriptors lists (MDL)
IRP_MJ_SHUTDOWN	IRP_MN_COMPLETE	System shutdown		Mark filesystem operation complete
IRP_MJ_CLEANUP	IRP_MN_COMPRESSED	Managing byte-range locks		Read and transfer compressed data
IRP_MJ_CREATE_MAILSLOT	IRP_MN_MDL_DPC	Release process file resources		Merge DPC with completion alert
IRP_MJ_QUERY_SECURITY	IRP_MN_COMPLETE_MDL_DPC	Manage mailslots for communication		Merge DPC, MDL with completion alert
IRP_MJ_SET_SECURITY	IRP_MN_SCSI_CLASS	Query security info about objects		Manage SCSI devices
IRP_MJ_POWER	IRP_MN_START_DEVICE	Set object security information		Starting and initializing devices
IRP_MJ_SYSTEM_CONTROL	IRP_MN_QUERY_REMOVE_DEVICE	Manage device power state		Pending device removal
IRP_MJ_DEVICE_CHANGE	IRP_MN_REMOVE_DEVICE	Request system-wide operations from OS		Pending device removal tasks
IRP_MJ_QUERY_QUOTA	IRP_MN_CANCEL_REMOVE_DEVICE	Manage device events		Cancel device removal
IRP_MJ_SET_QUOTA	IRP_MN_STOP_DEVICE	Volume quota information		Stop device
IRP_MJ_PNP	IRP_MN_QUERY_STOP_DEVICE	Set volume quota		Query stop device possibility
IRP_MJ_TRANSACTION_NOTIFY	IRP_MN_CANCEL_STOP_DEVICE	Manage Plug-and-play devices		Cancel stop device
Fast IO	IRP_MN_QUERY_INTERFACE	Manage operations notifications		Discover device relationships
IRP_MJ_FAST_IO_CHECK_IF_POSSIBLE	IRP_MN_SET_LOCK	Check object fast I/O capability		Supported device interfaces
IRP_MJ_DETACH_DEVICE	IRP_MN_QUERY_CAPABILITIES	Detach device and free resources		Manage object lock requests
IRP_MJ_NETWORK_QUERY_OPEN	IRP_MN_QUERY_RESOURCES	Query network connection information		Query device capabilities
IRP_MJ_MDL_READ	IRP_MN_QUERY_RESOURCE_REQUIREMENTS	Read from file or device via MDL		Query device resource requirements
IRP_MJ_MDL_READ_COMPLETE	IRP_MN_QUERY_DEVICE_TEXT	Notify MDL object read completion		Query device description and location
IRP_MJ_PREPARE_MDL_WRITE	IRP_MN_FILTER_RESOURCE_REQUIREMENTS	Prepare file for MDL write		Modify device resource requirement
IRP_MJ_MDL_WRITE_COMPLETE	IRP_MN_READ_CONFIG	Complete and notify of MDL write		Read connected device configuration
IRP_MJ_VOLUME_MOUNT	IRP_MN_WRITE_CONFIG	Mount a volume		Write connected device configuration
IRP_MJ_VOLUME_DISMOUNT	IRP_MN_EJECT	Dismount a volume		Remove device from system
FsFilter	IRP_MN_DISABLE_COLLECTION			Prevent automatic resource collection for device
IRP_MJ_ACQUIRE_FOR_SECTION_SYNCHRONIZATION	IRP_MN_EXECUTE_METHOD	Acquire memory block for synchronization		Invoke device functions via its driver
IRP_MJ_RELEASE_FOR_SECTION_SYNCHRONIZATION	IRP_MN_QUERY_ID	Release sync memory block		Uncover the identity of a device
IRP_MJ_ACQUIRE_FOR_MOD_WRITE	IRP_MN_QUERY_PNP_DEVICE_STATE	Acquire memory section for modifier write		Query PnP device state
IRP_MJ_RELEASE_FOR_MOD_WRITE	IRP_MN_QUERY_BUS_INFORMATION	Release modifier write memory		Query connected bus information
IRP_MJ_ACQUIRE_FOR_CC_FLUSH	IRP_MN_DEVICE_USAGE_NOTIFICATION	Acquire memory for cache coherency flush		Notify of device power usage
IRP_MJ_RELEASE_FOR_CC_FLUSH	IRP_MN_SURPRISE_REMOVAL	Release memory for cache coherency flush		Notify of unexpected device removal
IRP_MJ_NOTIFY_STREAM_FO_CREATION	IRP_MN_QUERY_LEGACY_BUS_INFORMATION	Notify driver of file create		Query legacy bus hardware information
	IRP_MN_WAIT_WAKE			Enable wake from low-power state
	IRP_MN_POWER_SEQUENCE			Manage device power during transitions
	IRP_MN_SET_POWER			Change power state of device
	IRP_MN_QUERY_POWER			Query ability to move to a named power state
	IRP_MN_CHANGE_SINGLE_INSTANCE			Get info about named instance of device
	IRP_MN_CHANGE_SINGLE_INSTANCE			Modify named instance of device
	IRP_MN_ENABLE_EVENTS			Modify named device property
	IRP_MN_DISABLE_EVENTS			Enable specific event generation from device
	IRP_MN_ENABLE_COLLECTION			Disable device from events generation
				Enable automatic resource collection for device