Artifact Evaluated NDSS
Available
Functional
Reproduced

Detecting Ransomware Despite I/O Overhead: A Practical Multi-Staged Approach

Christian van Sloun[*,✉], Vincent Woeste[*], Konrad Wolsing[*,†], Jan Pennekamp[*], and Klaus Wehrle[*]
[*] Communication and Distributed Systems, RWTH Aachen University • {lastname}@comsys.rwth-aachen.de
[†] Cyber Analysis & Defense, Fraunhofer FKIE • {firstname.lastname}@fkie.fraunhofer.de

Abstract*—Ransomware attacks have become one of the most widely feared cyber attacks for businesses and home users. Since attacks are evolving and use advanced phishing campaigns and zero-day exploits, everyone is at risk, ranging from novice users to experts. As a result, much research has focused on preventing and detecting ransomware attacks, with real-time monitoring of I/O activity being the most prominent approach for detection. These approaches have in common that they inject code into the execution of the operating system's I/O stack, a more and more optimized system. However, they seemingly do not consider the impact the integration of such mechanisms would have on system performance or only consider slow storage mediums, such as rotational hard disk drives. This paper analyzes the impact of monitoring different features of relevant I/O operations for Windows and Linux. We find that even simple features, such as the entropy of a buffer, can increase execution time by 350% and reduce SSD performance by up to 75%. To combat this degradation, we propose adjusting the number of monitored features based on a process's behavior in real-time. To this end, we design and implement a multi-staged IDS that can adjust overhead by moving a process between stages that monitor different numbers of features. By moving seemingly benign processes to stages with fewer features and less overhead while moving suspicious processes to stages with more features to confirm the suspicion, the average time a system requires to perform I/O operations can be reduced drastically. We evaluate the effectiveness of our design by combining actual I/O behavior from a public dataset with the measurements we gathered for each I/O operation and found that a multi-staged design can reduce the overhead to I/O operations by an order of magnitude while maintaining similar detection accuracy of traditional single-staged approaches. As a result, real-time behavior monitoring for ransomware detection becomes feasible despite its inherent overhead impacts.

I. Introduction

Ransomware attacks are currently one of the most prevalent cyber attacks [14], affecting servers, personal computers, laptops, mobile phones, and Internet of Things (IoT) devices [5]. Consequently, several techniques [5], [13], [23] have been proposed to deal with ransomware attacks [5], classified into three phases: prevention, detection, and response. In research, there is a heavy focus on host-based solutions [5], [8] since ransomware activity primarily occurs on the end host, and many strains exhibit limited network activity [23], *e.g.*, just brief interactions with a command and control server to obtain asymmetric encryption keys. Thus, network-based approaches typically focus on preventing damage in the first place by blocking this setup communication and preventing further spreading [23]. Response strategies, *e.g.*, storage-based solutions such as protection offered by cloud providers [30] or the use of additional storage space, transparently back up files and revert the damage done by ransomware [9], [13], [39]. However, cloud storage creates network overhead, and space is often severely limited, with additional storage space requiring recurring payments, and adding additional local storage may not be possible, *e.g.*, for IoT devices or laptops.

Another defense tactic and key to this paper is *dynamic system behavioral analysis*, which presents a detection approach to ongoing attacks. Ransomware targets to quickly access and modify many files because the risk that a user will try to open an encrypted file, thus uncovering the ransomware, increases with time and the number of affected files. As a result, ransomware exhibits peculiar behavior when interacting with the file system, which prompted many state-of-the-art detection mechanisms to exploit these patterns for ransomware detection by actively monitoring I/O activity for traces of malicious behavior [3], [7], [13], [24]. Consequently, these I/O operations must be intercepted and/or modified on the end host to collect statistical information, such as the file type accessed.

Important to consider is, that for dynamic behavioral analysis, the detection mechanism needs to monitor relevant activity at the kernel level, which cannot be offloaded to the network as done in many other security areas, *e.g.*, Wazuh [41], since the information needs to be collected locally in the kernel itself. While some features, *e.g.*, the number of files that are created, are relatively inexpensive and only cause minimal delays, more complex operations, such as calculating a file's entropy [27] or creating backups of a file [13], incur much higher delays. Similar to [36], we found that monitoring file I/O via a Windows File System Filter Driver (minifilter) or eBPF under Linux on systems equipped with a SATA solid-state drive (SSD) introduces an overhead of up to 25% (35%) (*cf.* Sec. III) and the peak performance of the SSD can be reduced to $\approx 25\%$ of its original performance (according to CrystalDiskMark 8.0.5). Consequently, features need to be monitored sparingly to maintain the usability of modern hardware.

To the best of our knowledge, related work focused more on detection performance and storage overhead than on

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240764
www.ndss-symposium.org

realistic practicality. Ahmed *et al.* [3] investigated CPU and memory overhead but did not consider I/O performance. While Continella *et al.* [13] investigated the I/O overhead of ShieldFS and acknowledged overheads of 25% or more, they considered it no noticeable impact in real-world scenarios. We attribute this to the fact that Continella *et al.*'s setup consisted of rotational hard disk drives (HDDs), for which the Input/output operations per second (IOPS) performance typically falls below 100 IOPS. A fixed overhead caused by the detection mechanism can significantly impact real-world performance, especially for modern SSDs (1,500,000 IOPS [35]) and I/O-intensive applications. As a result, we question the typical assumption by related work, which is whether dynamic behavioral analysis for ransomware detection is still usable in modern computing scenarios and how to minimize performance degradation.

To study this question, we develop and evaluate various strategies for dynamic monitoring under the two most common Operating Systems (OSs), Linux and Windows, to analyze the I/O performance impact of real-world monitoring of features. Specifically, we investigate the impact of features like the entropy calculation for read/write calls and discuss its implications for the usability of existing ransomware detection approaches. Our results show that depending on the complexity of the monitored information, the execution time can be increased by up to 350%, which renders naive dynamic monitoring unfeasible for practical use.

However, we also discovered that basic features can be monitored with minimal overhead. Leveraging those, we propose a Multi-Staged Intrusion Detection System (IDS) (MS-IDS) to dynamically adjust the number of features monitored for each process to minimize the performance impact dynamic behavioral monitoring has on benign processes while maintaining comparable detection performance to traditional approaches that continuously monitor all features. Our MS-IDS uses Random Forest Classifiers (RFs) for each stage, which are optimized for different subsets of features to incur minimal overhead for each stage's considered set of features. To evaluate the effectiveness of the designed MS-IDS, we use a publicly available dataset [13]. Using an MS-IDS, we decreased the incurred I/O latency overhead for benign processes from $180.76 \pm 102.86\%$ to below $18.56 \pm 54.5\%$, which allows us to protect modern systems with negligent impact on practical ransomware detection.

In this paper, we make the following main contributions:

- We, to the best of our knowledge, provide the first overview of I/O performance considerations in state-of-the-art approaches (Sec. II).
- We identify performance limitations of behavioral monitoring on modern systems that require fast I/O (Sec. III). Thus, while offering good resilience against ransomware, security gains of existing state-of-the-art approaches may be disregarded from an end-user perspective due to their impact on I/O heavy workloads, especially with consumer devices featuring SSDs.
- We design and implement an MS-IDS that dynamically and individually adjusts the number of monitored features based on the behavior exhibited by each process (Sec. IV) and show that such an MS-IDS can reduce I/O overhead significantly while still achieving its goal of detecting malicious activity (Sec. V), demonstrating the feasibility of real-time behavior monitoring on modern systems.

Our artifacts were evaluated as available, functional, and reproducible by the artifact evaluation; please refer to Sec. A for further details on obtaining and using the artifacts.

II. Background and Related Work

Cryptographic ransomware is a category of malware that encrypts files on an infected host while withholding the used encryption key and promising to provide it against a ransom payment [13]. An attacker exploits the fact that the affected user may not possess data backups, thus risking losing valuable files or starting laborious recovery measures due to inaccessible vital files. While static and dynamic analysis is usable, recent research has primarily focused on dynamic approaches due to the ability to evade static approaches by creating new ransomware samples and/or modifying and obfuscating existing ones [2], [32]. Therefore, dynamic analysis of ransomware has become widely used, with the analysis of I/O activity being the most prevalent technique (*cf.* Tab. I) due to the inherent property of ransomware to modify the victim's file system.

A. I/O-based Ransomware Detection

Ransomware detection techniques that use I/O behavior to identify malicious interactions with the file system typically consider the following I/O characteristics [8], [43]: 1) *Reads/Writes of files:* As ransomware tries to quickly encrypt a large number of files relevant to the user, a large number of read/write operations are performed, 2) *entropy:* encrypted data typically has a high entropy as bytes follow a random distribution as a result of the encryption process, 3) *file type coverage:* ransomware typically targets a specific set of file types (relevant to the victim) and accesses a large number of these files in a short amount of time, 4) *directory listings:* to find the files that are to be encrypted, ransomware needs to traverse a large number of directories and analyze its contents, 5-7) *file system changes (create/rename/delete):* as file size may change after encryption, *e.g.*, due to adding padding or the addition of ransomware-specific meta information, encryption often cannot be performed in place. As a result, additional operations to create temporary files, delete the original, or rename a new file to the name of the original occur for each file processed by the ransomware. Since state-of-the-art approaches report near-perfect accuracy [43], and ransomware cannot change its need to access mass storage due to its inherent nature, dynamic behavior monitoring using file I/O-based characteristics is a promising approach to reduce the danger of ransomware. However, intercepting and recording features for every I/O operation is computationally expensive and adversely affects system performance.

TABLE I: State-of-the-art ransomware detection approaches that utilize I/O behavior monitoring. Many do not consider I/O performance or evaluate it against traditional HDDs. Entropy, accessed file types, and I/O access patterns/frequencies are common features most detectors use.

Paper	Year	ML-based	I/O Performance Evaluation	Disk Type	Features				
					Entropy	File Types	Directory Traversal	File Similarity	I/O Frequency/ I/O Pattern
Continella *et al.* [13]	2016	✓	✓	HDD	✓	✓	✓	x	✓
Kharraz *et al.* [24]	2016	?	x		✓	x	x	x	✓
Scaife *et al.* [37]	2016	x	✓	?	✓	✓	✓	x	✓
Kharraz and Kirda [25]	2017	✓	✓	?	✓	✓	✓	x	✓
Palisse *et al.* [34]	2017	x	✓	HDD	✓	x	x	x	x
Mehnaz *et al.* [29]	2018	✓	?	?	✓	✓	x	✓	x
Shaukat and Ribeiro [38]	2018	✓	x		✓	x	✓	✓	✓
Chew and Kumar [12]	2019	x	x		✓	✓	x	x	x
Hirano and Kobayashi [16]	2019	✓	(✓)	VM_{SSD}	✓	x	x	x	x
Lee *et al.* [27]	2019	✓	x		✓	x	x	x	x
Ayub *et al.* [7]	2020	✓	x		✓	✓	✓	x	(✓)
Jethva *et al.* [23]	2020	✓	x		✓	x	x	✓	x
Ahmed *et al.* [3]	2021	✓	x		x	x	x	x	✓
Sanvito *et al.* [36]	2022	x	∼	SSD	✓	✓	✓	x	✓
Ayub *et al.* [8]	2023	✓	x		x	✓	x	x	x

✓annotates Yes, (✓) annotates Yes (but indirectly), x annotates No, ∼ annotates partially considered, and ? annotates Uncertain

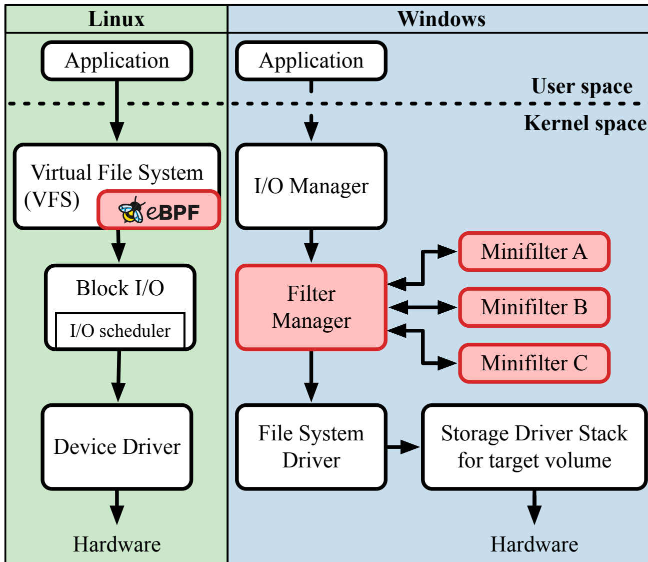

Fig. 1: Overview of I/O stacks under Linux/Windows.

B. File I/O in modern OSs

Before looking at related research w.r.t. to the I/O performance of security mechanisms, we first provide a brief and simplified overview of how Windows and Linux handle file I/O. Both OSs use a set of abstraction layers between the physical hardware, *i.e.*, the HDD or SSD, and the user-space application that initiates an I/O operation (*cf.* Fig. 1).

Windows: On Windows [20], [22], when the application issues a file operation to the kernel, the I/O Manager receives that request, looks up the requested file, checks access rights, and, if the volume is not mounted, suspends the request until the volume is available. Next, the I/O Manager creates an I/O request packet (IRP) and forwards it to the File System Driver. The driver then determines what operations need to be carried out, checks if the file is available in the system's cache, and, if not, forwards the request to the appropriate device driver for the target volume. Once the IRP is completed, the request's result is returned to the I/O Manager. On Windows, a file operation such as read/write may require several IRPs to complete the function in user space, *e.g.*, every read/write requires a handle to open the file and perform the requested operations before finally closing the handle. As not all of these operations can be performed in parallel, several interactions between the I/O Manager and the device driver of the volume may be required. In addition to the presented I/O stack, Windows has the concept of File System Filter Drivers, short *minifilters*, which can perform additional operations on IRPs. As a result, once a minifilter is registered, every IRP needs to be processed by the Filter Manager, which executes the relevant minifilters for the given IRP. Since Microsoft has registered several minifilters [17], one could hypothesize that such filters may process many IRPs. Thus, most IRPs are already processed by the Filter Manager, and minifilters have become the primary method to add I/O functionality to Windows [8], [13], [20]. In addition to the IRP-based I/O method, there is FastI/O [19], an optimized I/O path to access cached files, and BypassI/O [18], which allows even faster storage access, optimized explicitly for NVMEs and targets applications utilizing DirectStorage. Regardless of the I/O used, minifilters are generally supported.

Linux: On Linux [42], when an application invokes an I/O operation, the Virtual File System (VFS) checks whether the data is already present in the system cache (read) or if the cache can buffer the data (write). If the VFS determines that the hardware needs to be accessed, it initiates the descending I/O path for the request and hands it to the Block I/O Layer, which performs I/O scheduling and request merging. As a result, the I/O path may be temporarily suspended (*plugging*) in the request queue until sufficient requests are accumulated or a timeout occurs. Once the request queue is *unplugged*,

the requests are dispatched to the relevant device driver and executed by the storage medium. Once the storage medium finishes the request, an interrupt handles vital tasks before deferring the remaining work to a SoftIRQ that the Block I/O Layer handles. The Block Layer then wakes the user process that initiated the I/O request, which copies the requested data from the kernel to the application and finishes the I/O request.

Regardless of whether one uses a minifilter or eBPF, it is clear that adding code to the execution directly impacts the time an I/O operation needs to be completed.

C. Performance Overhead of Security Mechanisms

The advance of storage mediums, specifically the transition to use faster and faster SSDs, has prompted OS developers to optimize the I/O stack as much as possible [18], [19], [42]. At the same time, technologies like eBPF, or minifilters, allow security researchers to easily add additional functionality to the I/O, which is demonstrated by the fact that minifilters appear to be the preferred way by developers and researchers alike [8], [13], [20], [24], [43]. As a result, adding functionality via such filters may increase the execution time of I/O paths, thus significantly impacting system performance. Thus, if IDSs utilize such functionality, minimizing the overhead added to the I/O stack is vital to forestall the IDS thwarting the whole system's performance.

The vast majority of research on using IDS on end hosts has focused on the IDS's detection performance and is mainly evaluated on prerecorded datasets (*cf.* Tab. I); thus, real-time monitoring and feature processing is not a concern for them. A performance aspect often considered is the requirement for computational resources in Machine-Learning (ML)-based IDS since the resource consumption for training/testing is often a significant concern [31], as deep-learning models often require a large amount of memory and GPU computing power [6]. Nevertheless, some research specifically looked at the performance impact of security mechanisms on end hosts:

An area where computational resources are a more prominent concern is IDSs, specifically targeted at IoT deployments. However, as shown by Mudgerikar *et al.* [33], who use `strace` to monitor system call behavior, it takes two context switches for each intercepted system call, causing significant overhead and slowing benign applications down significantly.

Another area that emphasizes the performance impact of security mechanisms on end hosts is the Anti-Virus software (AV) sector. Even though AV vendors invest considerable effort to minimize the performance impact for end users, real-time monitoring still significantly impacts system performance, with applications requiring more than $2\times$ the number of CPU ticks to perform the same tasks due to the monitoring overhead induced by injected libraries [10].

Continella *et al.* [13] investigated the overhead of their ShieldFS filesystem. While their measurements found that their implementation could cause overheads up to $3.8\times$, the average perceived overhead on five machines of real users was only $0.26\times$ while not being noticeable. This could be explained by using rotational hard disk drives in the machines on which the evaluation was performed. As a result, they did not perform a root-cause analysis to identify the cause for the overhead, nor did they see the need for improvements.

Anecdotal evidence by Sanvito *et al.* [36] indicates that monitoring may cause significant slowdowns to systems with modern hardware. They found that enabling eBPF monitoring of I/O-related calls can cause significant degradation of I/O performance and even hypothesize that reducing the number of monitored features in a multi-staged architecture could solve this problem. However, the authors did not analyze what features would be considered expensive to monitor, nor did they implement their system; instead, they only evaluated the detection performance based on a dataset. Thus, no assertions w.r.t. to performance improvements can be made. Additionally, their approach proposed to start monitoring new processes with minimal features and did not state which features were explicitly considered. We see several problems with this approach: 1) their implementation assumes the identification of "ticks" (similar to [13]), where a tick is triggered every time a process interacts with a predefined number of files. However, if the number of monitored features is minimized, many calls will not be captured; thus, the ticks cannot be calculated in practice; 2) it is unclear what performance improvements could be gained by dynamically adjusting the number of features that are monitored; 3) the authors did not consider how the adjustment of monitored features would be handled in practice, nor did they investigate the effects between how the number of monitored features is changed and its implication on the detection of ransomware and the potential for false positives.

Summarising the above, we conclude that promising state-of-the-art security mechanisms, while providing excellent protection against ransomware, often affect heavily optimized OS mechanisms. At the same time, the effect on a system's performance has not seen widespread attention, leading to several proposed state-of-the-art mechanisms that can only be considered infeasible to deploy in modern systems, leaving potential victims without essential protection mechanisms.

For some users, this overhead may be mostly negligible, *e.g.*, office workers who mainly handle email and text documents where I/O activity is low and infrequent. Still, they are also affected during resource-intensive background tasks such as system/software updates and restarting of applications or the OS. Other users are more significantly affected, *e.g.*, users doing video editing, *i.e.*, copying significant amounts of files to and from their machines and executing encoding tasks. They would see significant slowdowns in their workflow due to their reliance on fast I/O. Similarly affected are gamers, as textures must be loaded, which mainly depends on I/O speed, or users with limited RAM who regularly rely on fast swap.

Research does not appear to be actively tackling the performance issue and analyzing the trade-off between the security gains and the induced overhead. Thus, an analysis of the effect of different features, monitored by the kernel, is required to identify starting points for optimization.

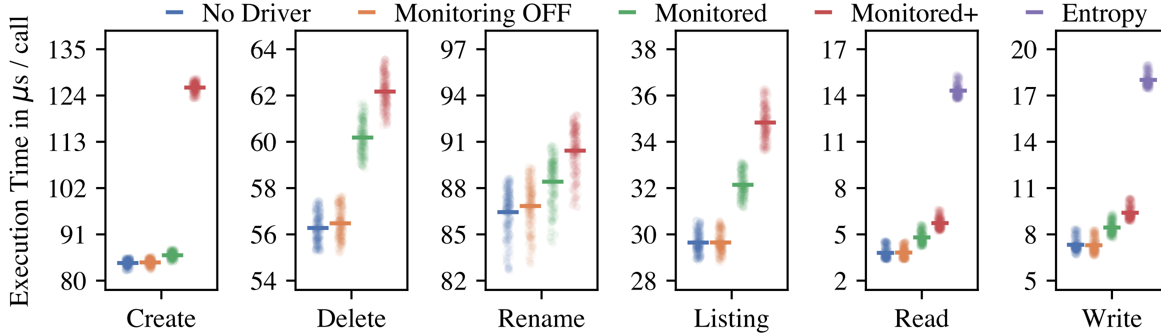

Fig. 2: Monitoring overhead for different system calls on Windows 11. The plots show the distribution of the 10% trimmed execution times across 250 × 10,000 individual measurements for each call and monitoring configuration. The horizontal line denotes the value of the trimmed mean. Monitoring of features quickly becomes computationally expensive, with the calculation of entropy values impacting execution times the most.

III. Initial Measurements

Sec. II shows that modifications to the highly optimized I/O stacks of modern OSs, as done by many security mechanisms, could significantly impact the overall system's performance and might significantly hinder deploying real-time monitoring of system behavior in real-world settings but has not seen significant attention by researchers in the IDS community. To identify where I/O is most heavily impacted, we analyzed the overhead incurred by intercepting the relevant system calls to monitor file I/O of software (*cf.* Fig. 2). We use an eBPF extension (Linux) and a minifilter driver (Windows), as both are highly optimized mechanisms designed to handle this functionality. We considered multiple levels of monitoring for each call:

Baseline: The kernel is unmodified, *i.e.*, kernel functions proceed unaltered to perform the disk I/O. Thus, no overhead exists.

Monitoring OFF: The driver is added to the kernel, but the PID is not included in the hash table. Thus, the execution of the kernel function resumes immediately. This variant shows the best possible performance for each call if it should be possible to monitor the call at all, *i.e.*, omitting the hash-table lookup would result in a call being unable to be monitored.

Monitored: The PID is included in the hash table, and the call is reported to the monitoring application. The reported information includes basic call parameters such as the calling process's PID and parameters provided to the intercepted function.

Monitored+: Includes the basic features of *Monitored* but also additional features that require additional calls to the kernel, *e.g.*, resolving the filename of the accessed file.

Entropy: Identical to *Monitored+*, but for READ and WRITE, the entropy of the read/written data is calculated, which is an essential feature as encrypted files typically have a higher entropy than regular files.

The measurements of I/O execution time were performed under Windows 11 (22621.3447) on a machine with an i7-4770 CPU@3.40GHz, 16GB RAM, and a 500GB SATA SSD. Similarly, Linux measurements were performed on an identical machine using Debian 11. We measured each level for each I/O operation 2,500,000 times. We spread the measurements over several days and different times to minimize the influence of background activity, *e.g.*, Windows performing updates or other OS maintenance tasks. Finally, we discarded the lower/upper 10% of data points to remove outliers due to unusually high/low response times by the SSD and get a clearer picture of real-world performance.

TABLE II: Distribution of I/O operations across ShieldFS.

Call Type	PID_{benign}	$PID_{ransomware}$	Overall
READ	49.31%	8.99%	23.65%
WRITE	33.17%	67.42%	54.97%
LISTING	12.37%	18.97%	16.57%
CREATE	2.24%	2.13%	2.17%
DELETE	1.56%	0.01%	0.57%
RENAME	1.35%	2.48%	2.07%
#calls	117,785,809	206,314,252	324,100,061

Our measurements (*cf.* Fig. 2) show that processing and reporting a call introduces a significant overhead to all call types that increases with the complexity of the gathered information, while the overhead introduced by installing the kernel extension but not recording a specific call is negligible. For eBPF, note that eBPF programs are limited in length/complexity. Thus, the entropy cannot be computed over the whole buffer, only the first 4096 B, to adhere to the limitations set by the verifier. Nevertheless, our analysis under Linux using eBPF shows similar monitoring overhead with up to 46.1%, which would increase for more complex operations once verifier limitations are relaxed further.

Another important factor in monitoring overhead is the frequency of the calls for which overhead is introduced. For example, READ operations account for nearly half of all operations performed by benign processes on the observed machines in the publicly available ShieldFS dataset [13] (*cf.* Tab. II). In contrast, they only account for 9% of calls performed by the ransomware samples. Comparatively, CREATE operations constitute ≈2% of calls overall.

To better assess the real-world impact on SSD performance, we also tested the effects of monitoring in **Entropy** mode using *CrystalDiskMark 8.0.5* [1], a popular open-source benchmarking tool to measure peak and real-world performance of SSDs. Regardless of real-world or peak-performance tests, the read-/write performance dropped from ≈530 MB/s to ≈130 MB/s. The performance can be recovered by enabling multiple threads per I/O queue and writing larger blocks of data (≥512 KiB). As multiple threads generate I/O requests in parallel, even if each request takes longer to generate, once I/O requests reach the queue of the File System Driver, the SSD begins to read/write and does not starve. However, while such settings can be configured in a synthetic benchmark, they do not reflect the implementation of real-world applications. Furthermore, we saw that for smaller block sizes (4 KiB), the number of threads per queue would need to be increased even more, potentially exceeding the number of processing cores of a CPU. We expect even more severe performance degradation for the latest PCIe Gen5 SSDs with data rates above 10 GB/s and IOPS performance above 1 million.

Overall, we see widespread monitoring causing significant delays to syscalls, affecting the performance of modern high-performance storage mediums. At the same time, if fewer features are monitored (**Monitoring OFF** or **Monitored**), the overhead in execution time of an I/O request could be drastically reduced. The significant focus on detection performance by related work and the limitations of existing measurements regarding I/O performance open a research gap. Thus, we investigate the effect of the real-time adjustment of the monitored features for each I/O operation w.r.t. overhead and ransomware detection, as this could result in drastic performance improvements, allowing for improved ransomware detection approaches.

IV. Design

To tackle the issue of the I/O overhead of existing ransomware detection approaches caused by monitoring I/O characteristics at runtime, we design a novel MS-IDS that dynamically adjusts monitored features for each process to minimize computational overhead while maintaining high detection accuracy. It fills the previously outlined research gap (*cf.* Sec. III) and follows the general hypothesis that attacks with previously unknown malware are rare compared to the number of benign processes. Thus, while overhead is acceptable for malicious processes, most processes are benign, and the influence of every security mechanism on regular operation should be minimized. However, since no signatures can be created for unknown attacks [40], signature-based detection cannot be applied universally, and anomaly detection is needed [40]. Therefore, every running process is a potential candidate for malicious behavior, even though most do not exhibit any. Thus, they may not require extensive monitoring but cannot be excluded from monitoring, as vulnerabilities in the software may compromise the process at runtime. Consequently, behavior monitoring results in an inevitable overhead to the runtime of all processes, thus slowing the overall system

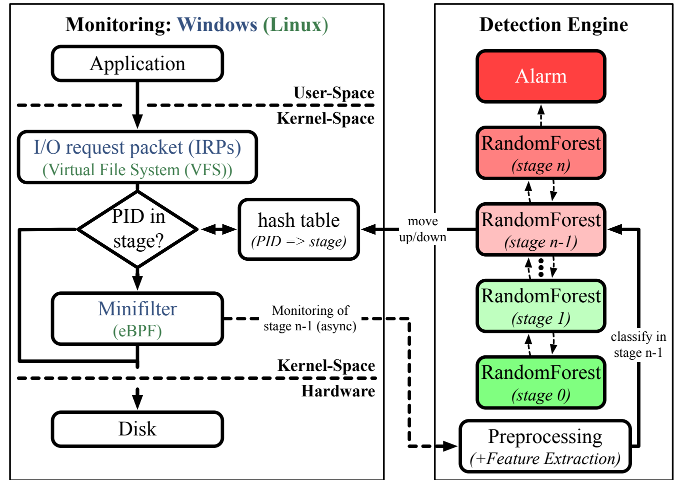

Fig. 3: Design of the MS-IDS. If calls for a given PID are intercepted, relevant features are extracted and forwarded to the MS-IDS while the execution of the PID continues.

significantly. Furthermore, there is a conflict between overhead and the desire to monitor features such as entropy to detect ransomware, as while it is a vital feature to identify ransomware (*cf.* Sec. II), it also creates significant overhead (*cf.* Sec. III).

Our MS-IDS minimizes this overhead by dynamically sorting processes into different levels of suspicion, *i.e.*, stages, where more suspicious processes are moved into higher stages with increased monitoring; thus, also increased overhead. In higher stages, the MS-IDS leverages more features to determine if the process is malicious or benign software that just momentarily exhibited abnormal behavior. At the same time, such a design can only function if these lower stages are sufficiently distrustful, *i.e.*, move processes to higher stages even for slight anomalies, as malicious processes may try to evade the MS-IDS by flying under the radar of a stage's detection threshold.

As a threat model, we use the same model as in [25], [29], and [43], *i.e.*, we focus on the execution phase of ransomware and assume a generally trusted operating system with pre-installed detection tools. An attacker has already gained access to a system and can access files like any other user-level program. However, he cannot interfere with the kernel driver used for monitoring or the classification itself.

In the following, we first provide more detailed insights into our design (Sec. IV-A) before discussing (Sec. IV-C) how the design is realized, how the classification in each stage works (Sec. IV-B), and what design decisions were made based on our measurements in Sec. III.

A. Design Overview

At the center of our prototype are two components: an OS-specific monitoring framework and the detection engine in the form of an MS-IDS (*cf.* Fig. 3):

Monitoring in Kernel: The OS-specific kernel extension monitors a process by intercepting all kernel functions required to extract the necessary features for the process's current

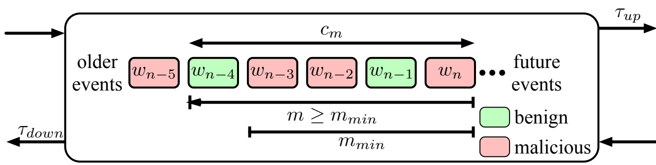

Fig. 4: Stage classification process.

stage, *e.g.*, syscalls or file I/O. Kernel functions currently not monitored for a process are continued, thus minimizing the introduced computational overhead. To monitor which calls need to be intercepted for each process, our MS-IDS uses a hash table that maps the process's PID to its current stage. If a kernel function is monitored, the relevant information is gathered and sent to the detection engine before the execution is returned to the regular operation of the function. An important aspect is deciding which calls are being monitored at which stage, which we analyze in detail in Sec. IV-C.

Multi-Staged IDS: The other component of our MS-IDS is the detection engine, which runs in a separate process to decouple the processing and classification of gathered information from the interception of the kernel function. The advantage of this is two-fold: first, the delay to kernel functions is minimized. Second, it allows the MS-IDS to scale by adjusting how often a process is classified and creating additional instances of the detection engine to handle many processes. For classification, we decided to use RFs as they are robust and scalable and provide the feature importance of each feature [11], which is essential when deciding which feature to put into which stage. Furthermore, using RFs has proven effective in detecting ransomware activity [13] based on I/O characteristics.

B. Classification Procedure

The MS-IDS monitors each process and individually decides if it should be moved to a higher/lower stage. While a process resides in a stage, all monitored events are passed to a process-specific buffer and classified with the stage's RF. The detection engine models the process's behavior by generating a classification window (w_n) every 20 preprocessed calls. Each window consists of the behavior across the last second of the process's execution for which relevant features are extracted, and finally, the window is classified. To allow for sufficient data for classification in a stage, the model requires $m \geq m_{min}$ classification windows to be present, where m_{min} denotes the minimum number of windows and m the currently available number (*cf.* Fig. 4). As each classified window has a certain probability of being malicious, c_m denotes the average probability that the current sequence of m windows is malicious. If c_m exceeds one of the stage's thresholds τ, it is moved to another stage, *i.e.*, if $c_m \geq \tau_{up}$, the process is moved to the next higher stage, while it is moved to the next lower stage if $c_m \leq \tau_{down}$. Once a process is in the last stage but still classified as malicious, an alarm is raised, and appropriate measures can be taken, *e.g.*, terminating the offending process to minimize damage to the system. Otherwise, if the stage changes, the change is communicated to the kernel module by updating the process's stage allocation in the hash table. If a process is supposed to be moved to a lower stage even though it is already in the lowest stage, the stage resets, *i.e.*, classification proceeds as if it has just entered the stage.

One crucial aspect is the classification right after a process moves to a different stage. If a process moves to a lower stage, *i.e.*, a stage with fewer monitored features, all relevant information of the previous second of the process's execution is already present. Thus, classification can proceed unaltered. However, if the process moves to a stage with more features, all monitored calls of the last second do not contain any information on previously unmonitored features. Therefore, we calculate these features only across those calls that include the new information and classify the process according to the outlined procedure.

Besides how a process is classified in a stage, another characteristic of an MS-IDS is how stages are interconnected and which features should be included in each stage.

C. Realization & Design Decisions

Like previous work [3], [13], [24], [26], we consider all I/O actions that READ, WRITE, CREATE, DELETE, or RENAME files or list the contents of directories (LISTING) of particular importance for the detection of ransomware activity. To lay the necessary foundations to realize the MS-IDS, *i.e.*, decide which features to monitor in which stage, we utilize the results of our in-depth analysis of the monitoring overhead that is caused by intercepting relevant system calls (*cf.* Sec. III) and the feature's feature importance in a model that contains all features. A crucial design aspect of an MS-IDS is the design of each stage, *i.e.*, what is monitored and the transitions between stages. Overall, the detection performance of the IDS should remain comparable to that of traditional approaches that monitor everything simultaneously.

During the realization of our MS-IDS, we investigate three criteria that influence the overall performance of an MS-IDS:

Classification Performance: One crucial difference of MS-IDS is that traditional IDSs try to balance precision and recall while raising as few false positives as possible and detecting all attacks. An MS-IDS can individually adjust the balance between precision and recall in each stage, as a false positive in an early stage does not directly result in an alarm, which a human operator would need to investigate. Instead, a process is moved to a stage with increased monitoring, while the performance of the process is merely penalized due to the increased monitoring overhead. As a result, a trade-off between precision in low stages and monitoring overhead is created. When looking at recall, it must be considered that lower stages sacrifice features for performance gain. Thus, even though higher stages could detect ransomware accurately if a lower stage falsely classifies an attack as benign, the process may never reach a stage where the malicious behavior is detected, or the process is only moved at a later point in time, which may cause more harm to the system.

Stage Overhead and Classification Frequency: Another aspect that needs to be considered when designing the stages

for an MS-IDS is the overhead introduced by each stage and how the MS-IDS decides when to classify a process. As the underlying idea of our approach is to reduce the set of monitored features, it means that if a call is not monitored for performance reasons, the MS-IDS cannot know about the file I/O that was performed. Not only is the information contained in the ignored call lost, but the fact that the call took place cannot be used to decide when to query the MS-IDS. Consequently, a tick-based approach, as presented by Continella *et al.* [13] or Sanvito *et al.* [36], cannot be adopted unaltered as our measurements (*cf.* Fig. 2) show that even monitoring basic information, *i.e.*, that a call has happened at all, already causes significant overhead. Similarly, a purely time-based approach would introduce overhead as it requires additional processing during scheduling if monitored per process. Furthermore, such an approach would not consider the I/O activity of the process, *i.e.*, it would require classifications for processes that do not perform any I/O while also waiting for the next "tick" even when a process fully utilizes the disk.

To decide which features to put into which stage, we trained an RF using the ShieldFS dataset [13] as it contains real-world benign behavior from eleven Windows machines. We then calculated the feature importance of each feature and considered, following related work [13], [24], [43], which file I/O operations perform persistent changes to the disk. For the design of our MS-IDS, these are WRITE, CREATE, RENAME, DELETE, and OVERWRITE operations. However, OVERWRITE did not appear to be used by the processes in the dataset and was also not considered by the original authors of ShieldFS. Thus, even though conceptually, it is compatible with our MS-IDS design, we omitted OVERWRITE in our feature space. Furthermore, we also took the frequency distribution of the different operations and the implications on the classification frequency into account. Finally, we settled on the following configuration of our five-stage model to distinguish between benign and ransomware processes while minimizing I/O overhead:

Stage 0: CREATE, RENAME, and WRITE are in *Monitored* mode, *i.e.*, the relative numbers of the respective calls are calculated for each window during preprocessing.

Stage 1: Monitoring WRITE operations is elevated to *Monitored+*, *i.e.*, the average write size and the number of written file types are calculated.

Stage 2: Monitoring WRITE operations is further elevated to *Entropy*, and DELETE and READ are monitored in *Monitored* mode. DELETE and READ are also considered for the relative numbers of calls, and the average written entropy is calculated for each window.

Stage 3: Monitoring READ operations is elevated to *Monitored+*, *i.e.*, the average read size and the number of file types read is calculated.

Stage 4: Monitoring READ operations is further elevated to *Entropy*, and LISTING is monitored in *Monitored* mode. LISTING is additionally considered for calculating the relative number of calls in each window, and the average read entropy across all calls is calculated.

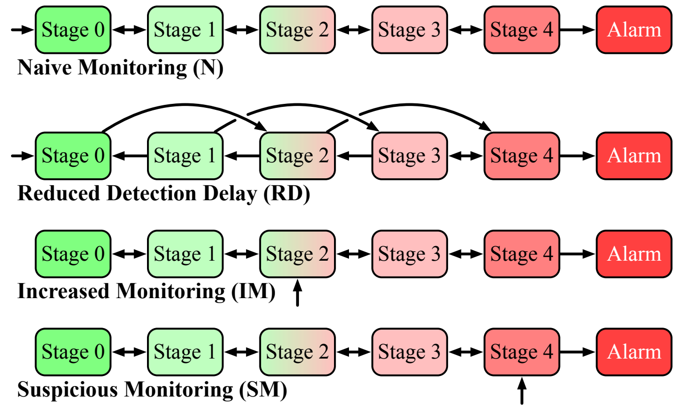

Fig. 5: Different Stage-Design configurations.

Detection Latency: A primary concern when designing an IDS is the latency between the start of an attack and its detection. For MS-IDS, this concern is exacerbated as it may require a positive classification by several stages until an alarm is raised. Therefore, the number of stages, the way the stages are interconnected, and how quickly a process is allowed to move to the next stage are fundamental properties that influence the detection latency.

We propose and evaluate four architectures (*cf.* Fig. 5) that interconnect the stages differently. The first is the naive monitoring approach, "Naive Monitoring (N)", where new processes are sorted to the stage with negligible overhead and moved up if suspected malicious. While this architecture initially leaves ransomware with the most leeway, it also imposes minimal overhead on benign processes. It builds upon the assumption that ransomware will quickly be moved up and detected by choosing ransomware-relevant features for all stages. However, it could also allow for more straightforward evasion, thus false negatives. The second architecture, "Reduced Detection Delay (RD)", moves a potentially malicious process up by two stages while only being moved down a single stage if a stage classifies them benign to reduce the detection latency. With this design, processes exhibiting malicious behavior are moved more quickly to stages that monitor additional features, thus making more qualified decisions about the process's behavior faster. The third architecture, "Increased Monitoring (IM)", is one where processes start in a medium state of observation. While this increases the overhead for all new processes, the assumption is that benign processes would quickly move to lower stages. Thus, their overhead is reduced, while malicious processes are directly analyzed with more features, thus reducing the risk of false negatives and decreasing their detection latency. The final architecture, "Suspicious Monitoring (SM)", uses the opposite assumption of the naive approach in that all processes are initially considered potentially malicious and are directly monitored with all features, but also maximum overhead. If a process exhibits benign behavior, it will gradually be moved to lower stages,

TABLE III: Distribution of session (s) and events (e) of the ShieldFS dataset after preprocessing of IRP logs.

	$\#s_b$	$\#s_m$	$\#e_b$	$\#e_m$
logs$_b$	258	-	75,145,003	-
citroni	-	14	-	1,870,921
crowti	-	125	-	61,619,839
cryptodefense	-	77	-	49,703,299
cryptowall	-	157	-	86,196,590
teslacrypt	-	10	-	6,923,603
logs$_m$	-	383	42,640,806	206,314,252
Overall	258	383	117,785,809	206,314,252

thus lifting overhead penalties. In contrast, malicious processes can be recognized directly and be terminated to minimize the potential damage they cause. However, depending on the tuning of this last stage, there are potentially more false positives, as the misclassification of a benign process would directly result in a false alarm.

Ultimately, designing an MS-IDS based on these considerations can drastically alleviate the cause overhead and, thus, improve system performance without sacrificing detection accuracy in detecting ransomware attacks.

V. Evaluation

Using our four architectures, we now study whether our MS-IDS can significantly reduce monitoring overhead while maintaining a high detection performance. To answer this question, we specifically investigate whether it (i) can detect all attacks, even though malicious processes need to traverse multiple stages, (ii) how the individual stages perform in their classification task under limited information, (iii) how long it takes for ransomware to be detected, and (iv) what overhead reduction can be achieved using multiple stages. Additionally, we compare all four architectures using the proposed five stages and how the results change when the number of stages is reduced to three. The idea is that with fewer stages, processes could be moved to the alarm stage faster, thus reducing the detection delay and, consequently, the damage caused by ransomware. Specifically, we omit the second and second-to-last stages for the three-stage variant of the MS-IDS. To this end, we first provide an overview of our evaluation setup (Sec. V-A) before we analyze how well our approach can detect ransomware from the widely-used dataset ShieldFS (Sec. V-B), and ultimately, the performance gains that can be achieved by dynamically adapting the number of monitored features (Sec. V-C).

A. Evaluation Setup

As the basis of our evaluation, we use the ShieldFS dataset [13], as it contains real-world IRP logs collected from 11 volunteer Windows systems and IRP logs of Virtual Machines (VMs) infected with ransomware samples. We decided against deploying our MS-IDS on real machines because, depending on the configuration, there may still be too much overhead and/or false positives that would cause processes to terminate unexpectedly. Furthermore, a direct comparison with the performance measured by Continella *et al.* [13] is not possible as while the authors did publish the IRP logs they gathered, machine configurations, code for performance measurements and ML-models are not contained in the publicly available data. Thus, we opted to utilize our measurement from Sec. III, as it gives us the precise overhead for each I/O operation and different feature combinations. Using this data, we can use our **Baseline** measurements and apply them to the behavior in the ShieldFS dataset to get a baseline of the system performance without interference by monitoring methods.

Preprocessing: For our MS-IDS, we were interested in file interactions commonly associated with ransomware activity: (i) CREATE, (ii) DELETE, (iii) RENAME, (iv) LISTING, (v) READ, and (vi) WRITE. However, as multiple IRP calls are required for several file interactions,*e.g.*, the creation or deletion of files, and are often only distinguishable by additional parameters in the IRP request, the data required some preprocessing to identify file system interactions such as creating, deleting, or renaming files or listing the content of a directory. In particular, we discarded calls that open/close file handles but do not perform actions that are not realized by the OS and do not change the state of the underlying file system. An example of such an action is a file marked for deletion several times; the actual deletion of the file is only realized once the last file handle is closed. Finally, we labeled the processes of each session according to the code and ground truth provided by the authors of [13]. The final distribution of events across all sessions and labels can be found in Tab. III. After labeling the dataset, we divide all sessions into individual processes and split them into train, validation, and test sets ($70\%/10\%/20\%$). We used the training data exclusively to train the RFs for each stage and the test set for the evaluation. For hyperparameter tuning of the RFs, we used 5-fold cross-validation.

Hyperparameter Tuning: For hyperparameter optimization of the stages, *i.e.*, the thresholds $\tau_{up/down}$ that decide when a process moves up/down a stage and the minimum number of events m_{min} that need to be generated before the process is allowed to move to the next stage, we used the validation set. We optimized hyperparameters using ray tune [28] and the Optuna [4] search algorithm. Ray was configured to search through 300 samples. Optuna was configured to perform a multi-objective optimization with the f1-score, the resulting overhead for benign processes, and the detection delay for ransomware processes as our objectives. Specifically, we considered the detection performance in the form of its f1-score as our main criteria while discarding all results that produced false classifications on the validation set, as a model directly using all features can achieve similar performance [43]. In the following, we explain the calculation of the other objectives:

We first divide the set of processes $\mathfrak{P}$ in the validation set of IRP-logs into two sets:

$$\mathfrak{R} = \{p | p \in \mathfrak{P}, p \text{ is ransomware}\}$$
$$\mathfrak{B} = \{p | p \in \mathfrak{P} \setminus \mathfrak{R}\}.$$

We then define the delay for a ransomware process as:

$$\text{delay}_p = \tanh\left(\frac{\#\text{calls}_p^{Alarm}}{t \cdot \sigma_{\mathfrak{R}}}\right), \qquad p \in \mathfrak{R},$$

where $\#\text{calls}_p^{Alarm}$ is the number of I/O operations the process performed until an alarm was raised, $\sigma_{\mathfrak{R}}$ is the standard deviation of the length of the I/O-logs generated by all ransomware processes, and t is the *acceptable* threshold of calls after which we require the ransomware to be detected. For our evaluation, we used $t = 1\%$. Thus, if ransomware is quickly terminated due to an alarm being raised, the value of delay_p is close to zero. At the same time, the longer it takes for the ransomware to be detected, a diminishing penalty is applied, *i.e.*, the metric asymptotically converges to one since regardless of whether the ransomware was able to encrypt 10 GB of data or 20 GB of data, both can be equally bad if vital files are affected early on.

Finally, we define the overhead a benign process is subjected to, compared to the overhead that would be added if all features were monitored for all calls, as:

$$\text{overhead} = \frac{\sum_{p \in \mathfrak{B}} \sum_{o \in O} \sum_{s \in S} \#\text{calls}_{s,o} \cdot \text{call_overhead}_{s,o}}{\sum_{p \in \mathfrak{B}} \sum_{o \in O} \#\text{calls}_o^{all} \cdot \text{call_overhead}_{t,o}},$$

where S is the set of stages, O is the set of I/O operations, and t refers to a traditional single-stage IDS, i.e., an IDS where all features are monitored at once (thus, the overhead is maximal).

Overhead can be directly used as an Optuna objective. At the same time, for delay, we consider two variants: first, the average delay across all ransomware processes, and second, the maximum delay.

B. Classification Performance

First, we study whether an MS-IDS can compete with traditional IDSs w.r.t. to its ability to detect ransomware, which heavily depends on how stages are designed and how they are interconnected (*cf.* Sec. IV-C). To this end, we use the optimal hyperparameters from our optimization and simulate each of the 12,847 processes in our test set to be classified by our MS-IDS, *i.e.*, the recorded IRP calls are passed to the detection engine for each process. The traditional model operates like an MS-IDS with only a single stage that contains all features.

1) Detection Performance: When comparing all versions of our MS-IDS, we can see (*cf.* Tab. IV) that all models can detect the 91 ransomware samples in our test set, which is not unexpected as the ransomware processes in the dataset created relatively long IRP logs; thus, theoretically, an MS-IDS has multiple tries to detect the ransomware, *i.e.*, for every subsequence of calls that would allow the process to reach the highest stage. However, due to the stage design, we also see that architectures, where processes start in lower stages, only classify around 21% of all processes, as the remaining processes do not perform enough I/O calls that are monitored by that stage, *i.e.*, they do not perform WRITE, CREATE, or RENAME operations (*cf.* Sec. IV-C). Thus, while "Naive Monitoring (N)" and "Reduced Detection Delay (RD)" monitoring appear to be performing the best from an f1-score perspective, both ignored a majority of processes completely. While this may not be problematic for cryptographic ransomware where WRITE operations are guaranteed, it shows that selecting which features to include in which stage can heavily impact an MS-IDS and needs to be carefully considered for other types of malware.

TABLE IV: Detection performance of the MS-IDS for three and five stages and the respective architectures. Some architectures have classified fewer processes due to them not containing the monitored calls of the initial stage of the architecture.

#Stages	Architecture	TN	FP	FN	TP	F1
1	Traditional	12746	10	0	91	**0.95**
3	N	2693	3	0	91	0.98
	RD	2694	2	0	91	**0.99**
	IM	11638	9	0	91	0.95
	SM	12749	7	0	91	0.96
5	N	2696	0	0	91	**1.00**
	RD	2694	2	0	91	0.99
	IM	11644	3	0	91	0.98
	SM	12745	11	0	91	0.94

Besides the overall detection performance of our MS-IDS, we also looked at the individual stages in detail. While every architecture can achieve perfect recall with all stages combined, ransomware processes may initially be erroneously moved to lower stages. Our results show that every architecture is optimized to have high recall in the stages that processes start in and the stages that are on a direct path to the alarm being raised. We also see a stage-by-stage increase in precision for these stages, showing that they are actively used for filtering benign processes. For stages that are not on the direct path, *e.g.*, all stages below the last stage in the case of "Suspicious Monitoring (SM)", we see that after being falsely moved down, some ransomware processes are temporarily bounced between stages before their behavior exceeds the required threshold of these stages and the alarm is raised.

Takeaway: All architectures show comparable detection performance to a single-stage model that uses all features simultaneously. Furthermore, if processes need to traverse multiple stages, an MS-IDS can slightly reduce the number of false positives.

2) Detection Latency: Another key metric for an IDS is its ability to quickly detect attacks and alert personnel or initiate preventative measures, *e.g.*, terminating the malicious process to minimize the harm a malicious program can cause. To study how quickly our MS-IDS can detect ransomware samples, we simulate each operation using the respective I/O logs. Specifically, we look at the number of bytes that ransomware

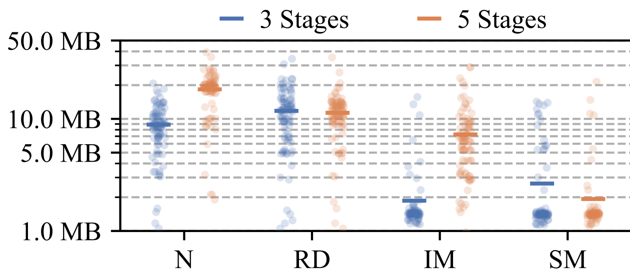

Fig. 6: Detection delay (written bytes) for three vs five stages. For comparison, the original number of bytes written by the 383 ransomware samples was 11.2 ± 6.8 GB.

can write before it is detected in Fig. 6. Comparing "Reduced Detection Delay (RD)" to "Naive Monitoring (N)" we see a slight decrease in the number of bytes written for the version with five stages, which is due to the reduction of a minimum of 25 to 15 events, *i.e.*, a minimum of 5 events per stage, that need to be processed before the ransomware reaches the ALARM stage. This slight decrease cannot be seen for three stages because hyperparameter optimization found a minimum of 15 events optimal for the shortest path through all stages for both architectures. Thus, the three stages' main advantage compared to the five is that additional features are added more quickly, resulting in more ransomware events being generated faster and fewer write calls and written bytes. For "Increased Monitoring (IM)" and "Suspicious Monitoring (SM)", most variants can decrease the number of written bytes by one order of magnitude compared to the other two configurations. Specifically, there is no significant difference between the "Suspicious Monitoring (SM)" versions (both require a minimum of ten events). Also, the three-stage version of "Increased Monitoring (IM)" performs similarly even though LISTING is not monitored for the first five out of ten required events. Only the five-stage version has difficulties detecting some ransomware samples quickly, indicating that those samples were temporarily falsely classified and bounced between stages (*cf.* Sec. V-B1).

All variants of our MS-IDS show that they can quickly detect ransomware and reduce the average damage done by the considered samples below 18.33 ± 6.4 MB (0.17% from the original 11.2 ± 6.8 GB of bytes written by the ransomware samples contained in the dataset). Furthermore, we see that "Suspicious Monitoring (SM)" performs on par with a traditional approach with only one stage (detection delay: 1.9 ± 2.8 MB). In contrast, the most extended detection delay introduced using multiple stages only increases the number of written bytes by a single order of magnitude.

While the detection delay is an inherent characteristic of our design for an MS-IDS, it could be improved by additional optimizations. As ransomware, at the beginning of its execution, typically needs to find the files that are to be encrypted, initial stages could be tailored to identify this behavior, thus detecting ransomware more quickly specifically.

Takeaway: Thus, if the performance gained through multiple stages is significant (*cf.* Sec. V-C), we argue that the accompanying detection latency is acceptable.

3) Detection of Novel Ransomware: Another interesting aspect is MS-IDS's performance on novel ransomware samples. To this end, we evaluated our MS-IDS on recent samples from [21], which contains mostly ransomware from 2020 to 2023. Unmodified, *i.e.*, with hyperparameters and RFs trained on data from 2016, our MS-IDS and also the traditional single-staged model were able to detect 12 out of 47 novel samples with 8 samples being detected by both. After manually adjusting hyperparameters between stages slightly to increase the detection rate, the MS-IDS was able to detect 18 samples in total, 10 of which were also detected by the traditional model.

Takeaway: Even though the detection rate (25 % to 37 %) appears low, it was achieved using supervised ML with models having only seen ransomware up to 2016. Furthermore, the MS-IDS performed on par with the single-staged model, indicating that the low detection rate is not caused by the multi-staged architecture but by the outdated RFs. Thus, if the overhead reduction is significant, MS-IDS remains relevant for modern ransomware.

C. Overhead

To ensure that MS-IDSs can be widely deployed, they do not only require a high detection performance but must have minimal impact on the system they are deployed to; otherwise, users may prioritize system performance over the increased security an anomaly-based an MS-IDS may provide over low-overhead signature-based solutions. Therefore, we look at how calls are distributed across stages to analyze the overhead (Fig. 7). Ideally, processes are quickly separated to both ends of our architectures so that most I/O operations of benign processes are executed while the process is in stages with low overhead. Of secondary importance is the distribution of I/O operations performed by malicious processes, as the overhead to these processes is non-critical but still indicates where ransomware has spent most of the time before detection. Additionally, we measure the relative and absolute overhead induced by monitoring benign processes (Tab. V). Finally, we briefly examine the performance impact w.r.t. CPU utilization and memory consumption.

1) Call Distribution: At first glance, Fig. 7 shows that for "Naive Monitoring (N)" and "Reduced Detection Delay (RD)", over 98.5% of all calls are performed in the lowest stage, which means that for over 62% of all benign I/O operations, the minifilter driver directly returns (*cf.* Tab. II). At the same time, only the minimal possible overhead is applied to the remaining operations. For "Increased Monitoring (IM)" and "Suspicious Monitoring (SM)", a more significant amount of I/O is spent in higher stages with high monitoring overhead. The main reason for this is that all processes start in a stage with high monitoring overhead, and if a process is light on file I/O, it cannot reach a low stage before it is terminated. As a result, most short-running processes are penalized with higher overhead, while longer-running processes reach the low stages where the overhead is minimized. Consequently, for 5.8% of all calls by benign processes, all features are monitored if the processes start in the highest stage.

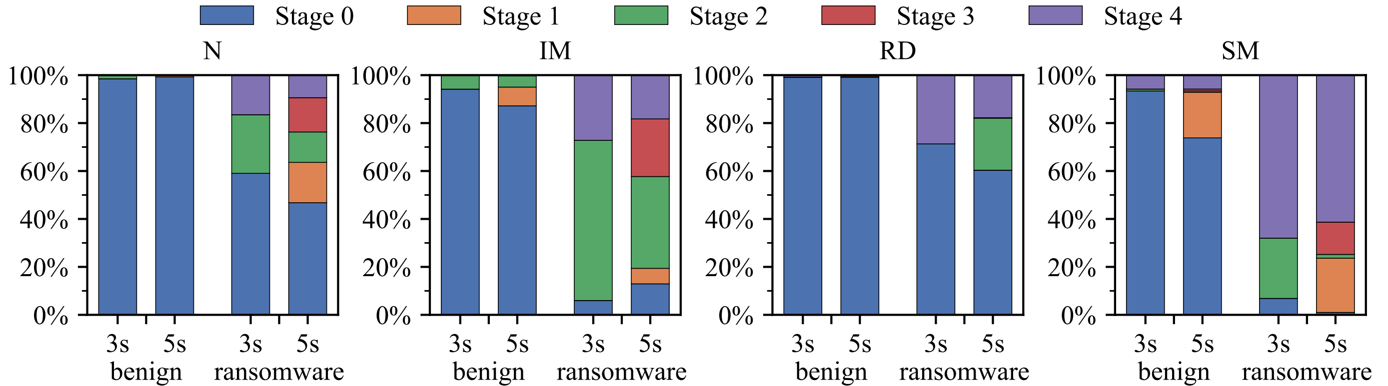

Fig. 7: Average number of calls across stages. For all architectures, IRP operations of benign processes are predominantly processed with minimal overhead (process resides in a low-overhead stage), while ransomware processes are quickly moved to higher stages where more features are monitored and can be leveraged to confirm the suspicion.

TABLE V: Relative and absolute overhead of different architectures compared to a traditional IDS. The average overhead can be reduced by an order of magnitude for all architectures.

#Stages	Architecture	overhead relative [%]			abs [s]
1	-	180.76%	±	102.86%	232.42
	(no entropy)	34.85%	±	16.83%	53.79
3	N	6.76%	±	14.45%	13.53
	RD	6.64%	±	15.44%	13.12
	IM	8.42%	±	17.03%	14.89
	SM	18.16%	±	54.38%	23.55
5	N	5.80%	±	8.34%	11.88
	RD	6.03%	±	10.18%	12.28
	IM	7.66%	±	13.62%	13.62
	SM	18.56%	±	54.50%	24.33

At the same time, ransomware processes perform the majority of file I/O operations in the initial stage of each architecture, the reason for which is found in our hyperparameter-tuning process. For "Naive Monitoring (N)" and "Reduced Detection Delay (RD)", while our MS-IDS is optimized to move suspicious processes quickly to stages with more features, it is also optimized to keep benign processes in a low stage. Furthermore, LISTING is not included in our set of features for the first stages due to the overhead minimization as its relevant IRP operation has multiple uses of which only a subset are relevant[1]. Thus, our MS-IDS inadvertently ignores the enumeration phase of ransomware samples but still reacts quickly once file encryption commences. For "Increased Monitoring (IM)", a similar effect is present, but even more increased since if a ransomware sample had been moved down a stage accidentally, it would need to be moved up by the lower stage and pass through the original stage again. Therefore, it may be cheaper w.r.t. delay and overhead to keep a process longer in the initial stage than wrongfully moving it. Finally, "Suspicious Monitoring (SM)" shows the expected behavior that over 60% of all calls are performed in the highest stage.

[1]only $\approx 10\%$ of the dataset are `IRP_MJ_DIRECTORY_CONTROL` operations, of which only $\approx 40\%$ are relevant for LISTING.

2) Overhead Measurements: While the call distribution shows that the stage design works in principle, *i.e.*, benign processes perform most of their file I/O in stages with low overhead, it does not quantify the actual performance gains. For this, we used the trimmed means of our measurements from Sec. III as our baseline to calculate the relative and absolute overhead for each I/O operation in each stage and calculated the overhead for all benign processes (Tab. V). From the results, it becomes clear that even if the processes start in the highest stage, our MS-IDS can decrease overhead by an order of magnitude compared to an IDS, which monitors everything. Similarly, if processes start in lower stages, this overhead can be reduced by 40%. However, diminishing returns apply as benign processes are moved quickly to lower stages, even if initially sorted to the stage with the highest overhead. As the entropy calculation of READ and WRITE operations cause the most overhead (*cf.* Fig. 2), we also included the overhead of a traditional approach with only a single stage, but without entropy calculations, *i.e.*, the maximum overhead is that of *Monitored+*. Still, multi-staged approaches (with entropy) can reduce the overhead for benign processes by over 50% compared to a single stage without entropy.

3) CPU and Memory Overhead: The impact of our MS-IDS on CPU and memory overhead can be divided into the overhead caused by monitoring and that of the detection engine. The only aspects of MS-IDS that affect memory consumption during monitoring are the hash table managed by the kernel and the buffering of IRP operations during communication with the detection engine. Both are negligible as they only consist of a few bytes per process. CPU overhead of monitoring is directly connected to the slowdown of IRP/s, as other calls are not modified or intercepted. As such, I/O heavy tasks are quickly I/O bound, which reduces overall CPU utilization, while processes with minimal I/O are not affected by monitoring.

Another aspect of potential overhead that hinders the practical deployment of an MS-IDS is the CPU/memory usage of the detection engine. Memory consumption varies by model (300 MB to 700 MB in RSS per RF) and whether models are kept in memory or loaded on demand. Consequently, CPU

and memory overhead depend on the classification method; however, unlike I/O monitoring, it can be offloaded to another machine. Therefore, the CPU and memory overhead of the detection engine is not a major concern for real-time monitoring using our approach.

Takeaway: All variants show that by employing multiple stages with different monitoring overhead, processes can effectively separated by their behavior and monitored accordingly. Even if processes start with the highest suspicion/overhead and must prove their benign nature, the average relative overhead induced on every I/O operation can be improved by 90% (50%) compared to no entropy monitoring).

VI. Discussion, Limitations, and Next Steps

Our analysis in Sec. V shows that the overhead caused by real-time behavior monitoring of file I/O can be mitigated to a large degree by dynamically adjusting which features are monitored on a per-process basis, *i.e.*, prioritizing relevant features, without compromising the IDS's ability to detect ransomware. Instead, only a few false positives are raised, even if processes start at a high stage where a single misclassification could result in countermeasures taking effect, *e.g.*, the (in this case) wrongful termination of the process.

Given these promising results, there are some limitations to our work, as well as future research directions to pursue:

Impact on Other Platforms: While our analysis measured the performance impact under Linux, its main focus was Windows because it is the primary target of most ransomware attacks [15], and public datasets of the I/O behavior of real-world machines and ransomware were available to us. As a result, our conclusions might differ in a comparable approach on Linux or any other OS. Generally, our findings translate because the general architecture of the I/O system is similar (*cf.* Sec. II), and adding functionality, interrupts, and extensions to the optimized execution is necessary. Likewise, we did not investigate the effects of dynamic behavior monitoring of file I/O on data-center SSDs, RAID configurations, *e.g.*, as used in network-attached storage systems, or the latest technologies such as *Microsoft DirectStorage*. Still, ransomware is an issue for consumer devices, where the transition to mainly using SSDs has made I/O performance relevant.

Security of MS-IDSs: The evaluations in Sec. V-B and Sec. V-C demonstrate that an MS-IDS can significantly reduce monitoring overhead while maintaining comparable detection performance to traditional approaches using File I/O for detection. However, the design of an MS-IDS does affect the leeway available to an attacker. A straightforward tactic would be to pause malicious activity and feign benign behavior until a lower alert stage is reached before resuming ransomware activity. The damage from each iteration may vary depending on how many stages a process must traverse before triggering an alarm. This is particularly effective in white-box scenarios, where attackers know the MS-IDS's configuration and can infer the ransomware process's stage – a common issue with many IDSs, including network- and host-based ones. In contrast, such attacks are less likely to succeed in black-box scenarios without prior knowledge of the MS-IDS's architecture. Here, attackers must monitor I/O execution times within the ransomware process to gauge whether they can safely continue their activities. Since response times are affected by the MS-IDS and other system activities, it becomes challenging for attackers to determine the current ransomware stage without knowledge of the classifiers. However, ML could help infer how suspicious the ransomware process appears, but these evasion tactics can also be used against traditional IDSs and are not exclusive to an MS-IDS.

Another approach an attacker could follow to evade an MS-IDS would be to divide the malicious activity across multiple processes so that the individual processes' behavior falls below the threshold of the last stage. This division of work was recently shown to be very effective against file I/O-based detection mechanisms by a prototype ransomware called ANIMAGUS [43]. The authors showed that approaches that consider individual processes, but even ones that incorporated system-wide statistics, *e.g.*, ShielfFS [13], could not detect ANIMAGUS successfully. Our MS-IDS does not intend to improve resilience against such evading tactics but instead addresses performance limitations that prevent the applicability of real-time IDSs in practice. We are confident that countermeasures to novel evasion tactics, such as those done by ANIMAGUS, can also be applied to our multi-staged design.

Finally, another general evasion strategy that could influence file I/O-based detection of ransomware, and, thus our MS-IDS, is partial encryption [43], *i.e.*, instead of encrypting the whole file, it is often sufficient to encrypt a significant portion of each file to prevent the original content from being recovered. Such a tactic primarily influences the usefulness of entropy as a feature for detection but is not an inherent problem of MS-IDSs. Again, this threat is not specific to our work but rather ransomware detection in general. The same applies to threats in which the MS-IDS is compromised itself.

Overall, every MS-IDS inherits the vulnerabilities to evasion of its underlying detection approach and its configuration.

Beyond File I/O: Our analysis solely focused on file I/O, and even here, we discovered many variants to achieve similar things within the kernel or parameters that would influence the design of an MS-IDS. Thus, extending the general idea of dynamically adjusting the monitored features to other types of malware would need additional in-depth analysis in the respective kernel implementation of the considered OS. As a guide for other researchers and our future work, we suggest focusing on operations performed with high frequency, requiring processing resources to handle, and may directly impact perceived system performance.

VII. Conclusion

This paper analyzes the impact of real-time monitoring of I/O behavior for ransomware detection on modern systems utilizing high-performance storage media. Due to the computational overhead that calculating relevant features would impose on every system call in a highly optimized part of the OS, such as its I/O stack, such behavior monitoring quickly becomes

infeasible in modern systems that utilize high-performance storage mediums, such as SSDs. Our analysis of the effect of monitoring different feature sets on the processing time of I/O requests uncovered severe degradation of SSD performance that could render real-time monitoring intolerable for end users. Thus, we proposed the real-time adjustment of which features are monitored as a potential solution to this overhead problem. We implemented an MS-IDS that uses different feature combinations in each stage to minimize the overhead to which benign processes are subjected. The MS-IDS rewards benign processes by lowering the imposed overhead while penalizing processes that behave anomalously by increasing the scrutiny of these processes. Using this approach, we reduced the average overhead inflicted upon IRP calls by an order of magnitude without affecting the MS-IDS's ability to detect ransomware quickly. As a result, real-time behavior monitoring for ransomware detection becomes feasible despite its inherent overhead impacts.

References

[1] "CrystalDiskMark - Crystal Dew World." [Online]. Available: https://crystalmark.info/en/software/crystaldiskmark/

[2] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020*, 2020.

[3] M. E. Ahmed, H. Kim, S. Camtepe, and S. Nepal, "Peeler: Profiling Kernel-Level Events to Detect Ransomware," 2021. [Online]. Available: http://arxiv.org/abs/2101.12434

[4] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," 2019.

[5] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, "Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions," *Computers & Security*, 2018.

[6] J. Arshad, M. Azad, R. Amad, K. Salah, M. Alazab, and R. Iqbal, "A Review of Performance, Energy and Privacy of Intrusion Detection Systems for IoT," *Electronics (Switzerland)*, 2020.

[7] M. A. Ayub, A. Continella, and A. Siraj, "An I/O Request Packet (IRP) Driven Effective Ransomware Detection Scheme using Artificial Neural Network," in *2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI)*, 2020.

[8] M. A. Ayub, A. Siraj, B. Filar, and M. Gupta, "RWArmor: A static-informed dynamic analysis approach for early detection of cryptographic windows ransomware," *Int. J. Inf. Secur.*, 2023.

[9] S. Baek, Y. Jung, A. Mohaisen, S. Lee, and D. Nyang, "SSD-Insider: Internal Defense of Solid-State Drive against Ransomware with Perfect Data Recovery," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018-07-02/2018-07-06.

[10] M. Botacin, F. D. Domingues, F. Ceschin, R. Machnicki, M. A. Zanata Alves, P. L. de Geus, and A. Grégio, "AntiViruses under the microscope: A hands-on perspective," *Computers & Security*, 2022.

[11] L. Breiman, "Random forests," *Machine Learning*, 2001.

[12] Z. Chen, M. Simsek, B. Kantarci, M. Bagheri, and P. Djukic, "Host-Based Network Intrusion Detection via Feature Flattening and Two-stage Collaborative Classifier," 2023. [Online]. Available: http://arxiv.org/abs/2306.09451

[13] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi, "ShieldFS: A self-healing, ransomware-aware filesystem," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016.

[14] European Union Agency for Cybersecurity, *ENISA Threat Landscape 2023: July 2022 to June 2023*, 2023. [Online]. Available: https://data.europa.eu/doi/10.2824/782573

[15] C. Griffiths, "The Latest Ransomware Statistics (updated June 2024): AAG IT Support," 2024. [Online]. Available: https://aag-it.com/the-latest-ransomware-statistics/

[16] M. Hirano and R. Kobayashi, "Machine learning based ransomware detection using storage access patterns obtained from live-forensic hypervisor," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019.

[17] L. Hollasch, M. Hopkins, D. Mabee, N. Schonning, J. Pilov, J. Rajwan, E. Graff, and matt-msft, "Allocated Filter Altitudes - Windows drivers," 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/allocated-altitudes

[18] L. Hollasch and matt-msft, "BypassIO for Filter Drivers - Windows drivers," 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/bypassio

[19] ——, "IRPs Are Different From Fast I/O - Windows drivers," 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/irps-are-different-from-fast-i-o

[20] L. Hollasch, A. Viviano, and matt-msft, "Filter Manager Concepts - Windows drivers," 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts

[21] Y. Hou, L. Guo, C. Zhou, Y. Xu, Z. Yin, S. Li, C. Sun, and Y. Jiang, "An empirical study of data disruption by ransomware attacks," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE'24)*, Lisbon, Portugal, 2024.

[22] T. Hudek, M. Hopkins, A. Viviano, and T. Sherer, "Example I/O Request - an Overview - Windows drivers," 2021. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/example-i-o-request---an-overview

[23] B. Jethva, I. Traoré, A. Ghaleb, K. Ganame, and S. Ahmed, "Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring," *J. Comput. Secur.*, 2020.

[24] A. Kharaz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz

[25] A. Kharraz and E. Kirda, "Redemption: Real-Time Protection Against Ransomware at End-Hosts," in *Research in Attacks, Intrusions, and Defenses*, 2017.

[26] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda, "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[27] K. Lee, S.-Y. Lee, and K. Yim, "Machine Learning Based File Entropy Analysis for Ransomware Detection in Backup Systems," *IEEE Access*, 2019.

[28] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[29] S. Mehnaz, A. Mudgerikar, and E. Bertino, "RWGuard: A Real-Time Detection System Against Cryptographic Ransomware," in *Research in Attacks, Intrusions, and Defenses*, 2018.

[30] Microsoft, "Ransomware detection and recovering your files." [Online]. Available: https://support.microsoft.com/en-us/office/0d90ec50-6bfd-40f4-acc7-b8c12c73637f

[31] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018*, 2018.

[32] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007.

[33] A. Mudgerikar, P. Sharma, and E. Bertino, "E-Spion: A System-Level Intrusion Detection System for IoT Devices," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019.

[34] A. Palisse, A. Durand, H. Le Bouder, C. Le Guernic, and J.-L. Lanet, "Data Aware Defense (DaD): Towards a Generic and Practical Ransomware Countermeasure," in *Secure IT Systems*, 2017.

[35] Samsung, "Samsung 990 PRO PCIe 4.0 SSD | Samsung Semiconductor Global — semiconductor.samsung.com." [Online]. Available: https://semiconductor.samsung.com/consumer-storage/internal-ssd/990-pro/

[36] D. Sanvito, G. Siracusano, R. Gonzalez, and R. Bifulco, "Poster: MUSTARD - Adaptive Behavioral Analysis for Ransomware Detection," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[37] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data," 2016.

[38] S. K. Shaukat and V. J. Ribeiro, “RansomWall: A layered defense system against cryptographic ransomware attacks using machine learning,” in *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*, 0003/2018-01-07.

[39] K. P. Subedi, D. R. Budhathoki, B. Chen, and D. Dasgupta, “RDS3: Ransomware defense strategy by using stealthily spare space,” in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, 27 Nov.-1 Dec. 2017.

[40] M. Tavallaee, N. Stakhanova, and A. A. Ghorbani, “Toward Credible Evaluation of Anomaly-Based Intrusion-Detection Methods,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 2010.

[41] Wazuh, “Wazuh - Open Source XDR. Open Source SIEM. — wazuh.com.” [Online]. Available: https://wazuh.com

[42] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, “Optimizing the Block I/O Subsystem for Fast Storage Devices,” *ACM Trans. Comput. Syst.*, 2014.

[43] C. Zhou, L. Guo, Y. Hou, Z. Ma, Q. Zhang, M. Wang, Z. Liu, and Y. Jiang, “Limits of I/O Based Ransomware Detection: An Imitation Based Attack,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

APPENDIX A
ARTIFACT APPENDIX

A. Artifact Description & Requirements

The artifacts published together with this paper support future research by allowing fellow researchers to use or modify the MS-IDS and compare results. As such, we want to publish our code and RF models and help reproduce results by providing instructions on using these artifacts.

1) How to access: The artifacts are available online and can be downloaded from Zenodo via https://doi.org/10.5281/zenodo.14249681.

2) Hardware dependencies: For I/O operation overhead measurements in Sec. III (C2), a desktop machine with typical hardware (8 cores, 16 GB RAM) and equipped with a SATA SSD, is required. The OS (Windows/Linux) must be installed on bare metal. The evaluation in Sec V. (C3) does not require special hardware; however, more cores (and more RAM) may significantly improve processing times when running preprocessing or training for the full ShieldFS [13]) dataset.

3) Software dependencies: The programming language mainly used throughout the artifacts (especially for the evaluation in Sec. V) is Python and requires a Python environment of version 3.11.7. Required modules (and version) are included in the `requirements.txt`, which is part of the artifacts. On top of that, some way to open and use a Jupyter Notebook using the same Python version as the kernel is required, *e.g.*, using VScode or using the jupyterlab Python package. For I/O operation overhead measurements, the following requirements must be met:

Windows: The OS version used was Windows 11 Pro 22H2 (22621.3447). For driver compilation, Visual Studio 2022, and the WDK for Windows 11, version 10.0.22621.382, is required. Other versions of Windows that allow for driver development should work but were not tested.

4) Benchmarks: The artifacts depend on the usage of the ShieldFS [13]) dataset for which access can be requested via email (http://shieldfs.necst.it).

B. Artifact Installation & Configuration

Python packages can be installed using `pip install -r requirements.txt` using the requirements provided together with these artifacts.

a) Windows Testbed: A bare-metal installation of Windows 11 Pro (22H2) equipped with a SATA SSD is required to perform measurements on Windows directly. Using a virtual machine for measurements of delays (*cf.* Sec. III) is not possible as results would be heavily influenced by the hardware abstraction as I/O operations may be buffered and do not require a write-through to the underlying physical storage.

Configuring the machine for driver deployment involves the following steps and takes approximately 3 h to 5 h:

Disclaimer: Configuring the Windows machine to test unsigned drivers is non-trivial and should not be attempted on a machine that is required to function, *e.g.*, a fresh install of Windows may be the easiest solution to revert these changes. Access to a Windows 11 VM image of the machine used for our evaluation can be granted upon request (see README provided with the artifacts). However, results differ from real-world measurements due to I/O having to traverse virtualization, two I/O stacks, *i.e.*, in the guest OS and the host OS, and the background activity of both OSs.

In the following, we describe the steps needed to install the driver on a Windows machine:

1) Download the `windows_eval` folder to the testbed machine.
2) Install Visual Studio 2022 and the Software Development Kit (SDK) to compile the driver yourself (configure the installation to be ready for Desktop C++ development and select the spectre mitigated version of `MSVC v143 - VS 2022 C++ x64/86 libs` - the SDK should automatically be included) - we also provide a precompiled version in Zenodo, but Windows Version support may be limited (if the precompiled version is used, skip the next step).
3) Download and install the Windows Driver Kit (WDK) for your version of Windows (10.0.22621.382 in our case) - or see Visual Studio's package-manager.
4) If Secure Boot is enabled, disable it (required to disable driver integrity checks or to enable test signing)
5) Disable Driver Signature Enforcement using an elevated command prompt (**note that this disables a security mechanism in Windows**): `bcdedit.exe /set nointegritychecks on`
   Alternatively, Driver Signature Enforcement can also be temporarily disabled using Windows Startup Settings:
   a) Press and hold the `Shift`-Key while using the start menu to reboot the machine/VM
   b) In recovery mode, select *Troubleshoot*
   c) Now select *Advanced options*
   d) Select *Start-up Settings*, and in Start-up Settings, click *Restart* (as prompted) to change a Windows option
   e) *Disable driver signature enforcement* (Option 7)

This variant must be re-performed after each reboot of the machine/VM.

Compiling the driver requires the following steps:

1) Import the Driver Project
2) Build the Driver (Visual Studio) and install the driver by navigating to the `.inf` file in the driver's main directory:
   a) Right-click the file and choose install.
   b) In the window that pops up, browse to the driver's `.sys` file, typically under `filter/Debug/x64/`, choose it and click continue.
   c) Now an error will occur because the driver cannot find the `.exe`. Correct the path (the `.exe` is located under `user/Debug/x64`) and retry.

After installing the driver, make sure to also install Python 3.11 and the necessary required packages for the overhead evaluation (included in a separate `requirements.txt` inside the `windows_eval` folder).

C. Major Claims

- (C1): The delay incurred in monitoring I/O activity depends heavily on the feature set monitored for each call. This is proven by experiment (E1), whose results are reported in [Sec. III, Fig. 2].
- (C2): An MS-IDS achieves comparable detection accuracy for detecting ransomware to an approach utilizing all features at once in the ShieldFS dataset. This is proven by experiment (E2), whose results are reported in [Sec. V.B, Table IV].
- (C3): Even though a ransomware process may need to traverse multiple stages of an MS-IDS, ransomware is detected quickly; thus, the damage is minimized. This is proven by experiment (E2), whose results are reported in [Sec. V.B, Fig. 6].
- (C4): By employing multiple stages and adapting the monitoring overhead dynamically, the average relative overhead that is induced on every I/O operation can be improved by 90% (or 50% compared to a single-staged approach that does not use entropy as a feature). This is proven by experiment (E3), whose results are reported in [Sec. V.C, Fig. 7, Table V].

D. Evaluation

The following experiments require preprocessing of the ShieldFS dataset as well as the performance metrics measured for individual features on actual hardware. Note that preprocessing takes considerable time and resources ($\approx$26 h using 90 cores and $\approx$400 GB RAM) and more than a week for hyperparameter optimization of the multi-stage architectures. Thus, we include information on which processes were included in the training, validation, and test data, the optimal hyperparameters found through Optuna search, and the RF models as part of the artifact. Additionally, the performance measurements require a non-trivial configuration of physical machines (*cf.* Sec. A-A); thus, we included the measured delays in our scripts as they are needed in the experiments.

1) Experiment (E1): [Measurements] [$\approx$ 1 human-hour + 1-2 compute-hours]: This experiment uses the prepared Windows machine to reproduce I/O delays. Can also be executed in a VM.

[Preparation] Copy the `windows_eval` folder to the machine. Install the driver on a Windows 11 `x86/amd64` machine. `arm` is not supported. Reboot the machine using the *Shift* key to disable Driver Signature Enforcement (*cf.* Sec. A-B), and log back in.

[Execution] Start the Python script (`feature_overhead_windows.py`) found inside the `windows_eval` folder and wait for the experiment to finish.

[Results] The script prints the measured IO delays for the relevant calls and feature combinations. Results may differ depending on the used hardware, but the general trend of increased overhead for more monitored features remains. If a VM is used, results differ as I/O requests must traverse two IO stacks, virtualization software, and two OSes that may cause background activity.

2) Experiment (E2): [Detection Performance/Delay] [30 human minutes + 5-7 compute hours]: In this experiment, every process that is contained in the test data is simulated using the multi-staged detector.

The RF models for each stage and the processed test data must be available for this experiment. To simulate the performance of the MS-IDS on the given data, the Python script `simulate_stage_config.py` can be used. The hyperparameters for each architecture are contained in the file and will be applied automatically.

[Preparation] Have the processed version of the test data available (output of `preprocess.py`), as well as the RF models and relevant Python modules. You should also make sure to have at least 5 GB available for the script to store temporary files.

[Execution] Run `simulate_stage_config.py` while providing necessary parameters as described in the `README`. The script only executes the simulation for a single architecture that can be supplied as a parameter. Multiple instances of this script can be run in parallel using multiple terminals to gather results for all architectures. During execution, a progress bar informs about the number of IRP calls being processed and the progress. Once finished, a data frame is created, which stores the simulation results. Using these data frames, the Jupyter Notebook (`plot_experiments.ipynb`) part for (E2) can be executed.

[Results] If using the provided RFs, the notebook should report zero false negatives for all configurations, a low number of false positives, and a high F1 score. Furthermore, the detection delay plot (Fig. 6) is computed. If new RFs are being trained, results may vary slightly as no optimization and or cross-validation is performed.

3) Experiment (E3): [Overhead] [30 human-minutes]: Using the simulation results from (E2) and the measured overhead metrics for a SATA SSD on a Windows 11 machine (also included in the JupyterNotebook for plotting/computing re-

sults - `plot_experiments.ipynb`), the overhead can be computed compared to a single-staged model.

[Preparation] Make sure to have the data frames storing the simulation results available for each architecture.

[Execution] We provide a Jupyter notebook that can be used to process the results of (E2) or manual `simulate_stage_config.py` runs. Just open the Notebook, select the same Python 3.11.7 version for the kernel, and adjust the paths in the second code cell if files were placed in different directories. The notebook should run and produce the same tables/figures from Sec V (minus some postprocessing for formatting and font size scaling).

[Results] The distribution of IRP calls across stages should look similar/identical to Figure 7, while the generated Table should contain comparable values to Table V (We found that the values may differ minimally ($\pm 0.1\%$ relative overhead, $\pm 0.2\,\mathrm{s}$ absolute overhead) when executed on a different machine, which we attribute to floating point precision).