File System Shield (FSS): A Pass-Through Strategy Against Unwanted Encryption in Network File Systems

Arash Mahboubi[1], Seyit Camtepe[2], Keyvan Ansari[3], Marcin Pawłowski[4], Paweł Morawiecki[5], Jarek Duda[4], and Josef Pieprzyk[2]

[1] Charles Sturt University and Artificial Intelligence and Cyber Futures Institute, Australia
amahboubi@csu.edu.au
[2] CSIRO, Australia
{seyit.camtepe,josef.pieprzyk}@data61.csiro.au
[3] Murdoch University, Australia
Keyvan.Ansari@murdoch.edu.au
[4] Jagiellonian University, Poland
{pawlowski.mp,dudajar}@gmail.com
[5] Polish Academy of Sciences, Poland
pawel.morawiecki@gmail.com

Abstract. This study introduces the Digital Immunity Module (DIM), a novel pass-through file system gateway, positioned strategically between storage and endpoints to enhance the security of files accessed via network protocols such as NFS and SMB on SharePoint. DIM serves as a protective layer against ransomware, designed with dual objectives: (1) detecting statistical anomalies that may indicate potential encryption within the network file system, and (2) proactively expanding under-attack files using a reverse source-coding algorithm to deprive ransomware of the resources it needs to operate. For practical deployment, we have developed a proxy gateway that connects endpoints to Azure storage using the SMB protocol. This setup effectively differentiates between benign and malicious activities without needing to identify specific processes at the endpoints, i.e., a critical advantage in combating fileless ransomware, which often eludes conventional security mechanisms such as behavioral analysis. Upon detecting malicious encryption, DIM reacts by expanding the size of buffer blocks, preventing ransomware from accessing subsequent files and frequently causing the ransomware to self-terminate. Our comprehensive evaluation, involving a benign dataset of 11,928 files against 75 ransomware families, including fileless types, demonstrates that DIM significantly impedes and often terminates ransomware operations early in the attack life cycle. This confirms the practicality and effectiveness of this pass-through defence strategy.

Keywords: Ransomware · Information Theory · Filesystem in Userspace · Arithmetic encoding · Fileless

1 Introduction

Ransomware, often classified as cryptoviruses, poses a significant threat to cybersecurity through its use of unauthorised encryption processes to render data inaccessible. Complications like ransomware attacks can also emerge when organisational files stored on shared systems are inadvertently encrypted by employees using legitimate cryptographic tools, creating situations that resemble ransomware activities, particularly when decryption keys are mismanaged or lost. Presently, machine learning (ML) techniques are at the vanguard of ransomware detection, employing predictive models that discern behaviour characteristics of ransomware. These models analyse patterns such as distinctive API calls, system library interactions, and file system dynamics, including the frequency of file read and write operations. Although these methodologies demonstrate efficacy against known ransomware types, they encounter limitations in identifying novel, characterised strains of ransomware. The evolving nature of ransomware enables it to mimic the behaviour of benign software, thereby challenging the sustained accuracy and effectiveness of ML detection models.

The relentless evolution of ransomware introduces substantial challenges in its detection, as adversaries refine their tactics to engineer ransomware that closely imitates the behaviour of legitimate software. This deceptive similarity significantly complicates the task of ML systems in distinguishing between malicious and benign programs. Emerging methodologies, such as unsupervised learning and anomaly detection, offer potential advancements in identifying a typical behaviour pattern without relying exclusively on predefined threat databases. These approaches indicate a prospective paradigm shift in the methodologies employed for ransomware detection, enhancing the ability to detect novel threats that evade traditional detection frameworks. Despite the promising advances in ML techniques, the complexity and adaptability of ransomware necessitate further innovations in detection and prevention strategies. This need has lead us to developing the Digital Immunity Module (DIM), a targeted solution designed to address the unique challenges posed by modern ransomware.

This paper presents the Digital Immunity Module (DIM), an innovative framework designed to safeguard data at the file-system level from unauthorised encryption, with applicability to networked and cloud-based environments. DIM proactively intervenes by scrutinising and manipulating file write operations, thereby markedly impeding the encryption process through the expansion of data blocks, which in turn disrupts ransomware activities and enhances attack traceability. This approach underscores the enhancement of resilience against ransomware, especially targeting novel and adaptive variants, by concentrating on the protection of remotely stored critical data. Utilising a custom FUSE (Filesystem in Userspace) file system, DIM thoroughly monitors and controls data transfers between endpoints and network storage, actively identifying and mitigating malicious encryption attempts. This strategy not only fortifies the security of files on network and cloud storage but also supports organisational cybersecurity policies that emphasise the protection of critical data assets over endpoint device data. As a result, the implementation of DIM represents a significant shift in how cybersecurity defences are conceptualised and applied within organisational frameworks.

The structure of this paper is organised as follows: In Section 2, we review related work and provide the necessary background. Section 3 is dedicated to a detailed description of the DIM concept. In Section 4, we elaborate on the experimental setup, evaluation methods, and assess the efficacy and performance of DIM. Section 5 engages in a comparative analysis of DIM with state-of-the-art solutions. Finally, Section 6 summaries our findings and outlines future research directions.

2 Related Works and Background

File System-Oriented Solutions: This section compares our purpose-built model with related storage-level works that use file system features to detect and prevent ransomware operations. We highlight the differences between our approach and most existing results, focusing on the I/O patterns analysis. Kharraz et al. in [19] focuses on analyzing the behavior of ransomware attacks from a file system perspective, describing common characteristics of ransomware attacks by investigating their file system activity. The authors developed a minifilter driver to monitor I/O requests generated by the I/O manager on behalf of user-mode processes to access the file system. They analyzed the file system activity of multiple ransomware samples, identified their attack strategies, and categorized the malicious activities. One such category involves encryption mechanisms, mainly composed of customized and standard cryptosystems. The paper provides insights into the encryption process, other malicious activities, and how a malicious process interacts with the file system during a ransomware attack.

ShieldFS [9] builds upon the work of [19], introducing a forward-looking file system that can prevent the harmful effects of ransomware attacks on data in a transparent manner. ShieldFS proposes to achieve this through automatic detection and transparent file recovery capabilities at the file system level. The detection system designed by the authors is based on entropy analysis involving write operations, frequency of read, write, and folder-listing operations, dispersion of per-file writes, the fraction of files renamed, and file-type usage statistics. ShieldFS then looks for indicators of cryptographic primitives and scans the memory of any process considered "potentially malicious," searching for traces of typical block cipher key schedules.

UNVEIL [18] is a system that employs a file system monitor with direct access to data buffers involved in I/O requests, providing complete visibility into all file system modifications. UNVEIL's monitor establishes callbacks on all I/O requests to the file system generated by any user-mode process. For performance reasons, the system aims to set only one callback per I/O request while maintaining comprehensive visibility into I/O operations. UNVEIL analyzes access patterns in I/O traces, which include sequences of user-mode processes, available files, I/O operations, and the entropy of read or write data buffers.

It identifies distinctive I/O fingerprints for file locker ransomware samples by detecting repetitive I/O access patterns indicative of a ransomware strategy to deny access to user files. Additionally, UNVEIL computes the entropy of read and write requests to and from the same file offset, which is a common indicator of crypto-ransomware behavior. This pattern arises when ransomware reads the original file data, encrypts it, and then overwrites it with the encrypted version. However, this approach is not universally adopted across modern ransomware families and variants. Shannon entropy is used for this computation, assuming a uniform random distribution of bytes in a data block.

Paik et al. [27] proposed a method for detecting ransomware in flash-based storage, involving an access-pattern-based detector coupled with a buffer management policy designed for solid-state drives (SSDs). This method monitors read and write operations at the same location to identify instances where ransomware encrypts and then overwrites original files. However, its effectiveness may be limited if the read-write buffers on the SSD are small, and it may not always detect ransomware if the malicious software doesn't store files at the same location.

Baek et al. [2] introduced the SSD-Insider method, designed to detect and protect NAND flash-based SSDs from ransomware attacks. This approach scrutinizes unique attributes of I/O requests, such as block address, size, and type, to detect the presence of ransomware. It employs an Iterative Dichotomiser 3 (ID3) based binary decision tree for detailed analysis of these characteristics. The authors also propose an innovative Flash Translation Layer (FTL) design capable of restoring infected files by leveraging the delayed deletion feature inherent to NAND flash. Despite its ingenuity, this method incurs significant additional overhead on the SSD system.

Non-File System Oriented Solutions: This section reviews various ransomware detection and mitigation strategies, ranging from rule-based frameworks to those that use machine learning (ML) techniques. The focus of these strategies is on vigilant monitoring of process behaviors, meticulous analysis of network traffic, and scrupulous examination of system calls to detect any signs of suspicious or anomalous activities. These activities encompass various aspects such as OS log entries [8], changes to the Windows Registry [28], arbitrary file modifications [29], and the entropy of files [17]; [20]; [28].

The literature also discusses various machine learning approaches [4], including detection based on API/System Calls [30], file I/O operations [9], and network traffic features. These features include the average packet size, the number of packets exchanged between the host and other machines, and the source or destination IP addresses contained within packet headers [10].

Several research studies have explored process actions, referring to the event sequences occurring while a program or application runs. Researchers have also used process mining techniques to identify file system metrics and conducted Ransom Note Analysis to study instances of ransomware attacks [3]; [14]; [18]. Despite the advancement in detection techniques, recent ransomware variants such as DoppelPaymer, Sodinokibi, Hades, Ryuk, and Conti continue to pose significant challenges, including business disruption, data loss, reputational damage, remediation costs, and legal liabilities. These challenges primarily arise due to frequent changes in ransomware behavior, complicating the detection process. We have summarized related work in Table 1.

Table 1: Summary of ransomware detection techniques.

Detection techniques	**Reference**
Behaviour Based	[29], [18], [30], [12], [1], [15], [17]
I/O Request Packer Monitoring	[9], [2]
Network Traffic Monitoring	[10], [24], [5], [6], [26]
Storage Level	[9], [27], [2], [2]
Opcode-Bytecode Sequences	[3]
Process mining identifying file system metrics	[22]
Ransom Note Analysis	[18]
File and Binary Entropy checking	[28], [17], [20]
API Call	[13]
File types extension analysis	[29]
Others (Autonomous Backup and Recovery SSD)	[23]
Others (Windows Registry, log files, sdhash)	[8], [28], [14], [29], [3], [11]

3 Digital Immunity Module

This section introduces the concept and approach of our proposed Digital Immunity Module (DIM). The primary role of DIM is to actively prevent ransomware from encrypting files under its protection. DIM focuses on file security with two main objectives: (1) Identify DIM-protected files undergoing encryption, and (2) Prevent malicious encryption of DIM-protected files.

Encryption Detection – Shannon Entropy Approach: to achieve its objectives, DIM needs to effectively distinguish between encrypted and unencrypted files. Unencrypted files can be text files (such as PDFs, Microsoft Word documents, and LaTex files), multimedia files (like images, music, and videos), and other commonly used formats. We propose using Shannon's entropy concept as a natural method to differentiate between these file types. Consider a random variable X representing n events $x_1, \ldots, x_n$, with each event x_i occurring with a probability $P(x_i)$. In this context, entropy is defined as follows:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i). \tag{1}$$

We consider the events of our random variable X to be characters of the Unicode standard, including non-printable ones. Entropy is a useful measure to gauge the randomness of a file's content. High entropy is indicative of randomness, whereas redundant (correlated) content results in low entropy. To calculate the entropy $H(f)$ of a file $f = \{x\}_1^N$ with N characters from X, we count the occurrences ℓ_x of each character x in f, as follows:

$$H(f) = -\sum_{x \in f} \frac{\ell_x}{N} \log_2 \frac{\ell_x}{N},$$

where $\frac{\ell_x}{N}$ denotes the probability of character x occurring in file f. For instance, consider a file $f = \{aaaaaaaaa\}$. Here, $\ell_a = 9$, $N = 9$, and thus $\frac{\ell_a}{N} = 1$, leading to $H(f) = 0$. Another file $f' = \{aeaieou!\}$, with varied occurrences, will have $H(f') = 2.5$.

Each type of unencrypted file, treated as a source of symbols/characters, is assumed to have a specific probability distribution characterizing its source program. For sufficiently long files f generated by a PDF source, for example, the probability of a symbol x can be approximated by its occurrence ℓ_x in f:

$$P(x) \approx \frac{\ell_x}{N},$$

where N is the total number of characters in file f. Thus, $H(f)_{PDF} = H_{PDF}$, where $H(f)_{PDF}$ is the entropy of f generated by a PDF source and H_{PDF} is the entropy of the PDF source. However, this approximation is ineffective for "short" files. Since sources of unencrypted files typically generate byte characters, the maximum entropy H_{max} of a file equals 8 bits, occurring when source characters are uniformly distributed.

Entropy of Encrypted Files: We expect sufficiently long encrypted files to have entropy levels approaching $H_{\max}$. Cryptographically robust encryption typically produces cryptograms that are virtually indistinguishable from truly random sequences. To test this hypothesis, we conducted experiments on various file types. Initially, we measured the entropy of a representative sample for each specific file type. Then, we encrypted these samples using three different open-source encryption tools: AESCrypt, GnuPG, and Challenger, and repeated the entropy measurement. The results, depicted in Table 2, confirm with high confidence that a file f is encrypted if its entropy $H(f)$ satisfies $7.99 < H(f) \leq 8$.

However, it is important to remember that compressed files can also exhibit entropy values near $H_{\max}$, which necessitates careful interpretation in such cases.

Table 2: Entropy of encrypted and un-encrypted files

File Format	Unencrypted	Challenger	GnuPG	AESCrypt
.pptx	7.90786434941	7.99993409023	7.99992547868	7.99992691156
.bmp	6.38231379202	7.99994411725	7.99969271027	7.99994075204
.jpg	7.96841466702	7.99969969991	7.99972370710	7.99964407683
.docx	7.82881687843	7.99564129666	7.99531941975	7.99383985158
.pdf	7.85009879424	7.99998526860	7.99997917863	7.99998576992
.xlsx	7.93289273016	7.99846117766	7.99844636333	7.99869412812
.png	7.98653176951	7.99983209472	7.99978322342	7.99982432301
.tex	4.81546551317	7.99832884790	7.99416858853	7.99793348066
.sql	5.14310355930	7.99991774345	7.99949416462	7.99992064505

The above observations lead to the following conclusions: (a) Entropy can be used to characterize redundancy in files generated by specific applications, such as PDF creators, Microsoft Word, and PowerPoint. The entropy of plain or unencrypted files typically falls below 7.99. (b) Encrypted files generally exhibit a uniform probability distribution of characters, resulting in an entropy close to $H_{\max}$. However, a notable caveat is that compressed files may also display high entropy values, necessitating distinct consideration.

Measuring Entropy in Practice: Calculating file entropy typically requires fetching the entire file from the HDD into RAM, which is both computationally expensive and time-consuming, especially for large files. Detection processes that are time- and computation-intensive have proven inefficient in dealing with ransomware attacks. This inefficiency arises from the rapid pace at which ransomware can encrypt target files once an attack is initiated.

A practical solution involves determining a *'minimum threshold length of bytes'* whereby a portion of the target file can be used to approximate its entropy. This approximation aims to yield a value that closely mirrors the file's actual entropy, enabling DIM to make rapid decisions about potential encryption. To test this approach, we used 436 PDF documents containing historical Federal Reserve projections of the U.S. economy, commonly referred to as *Greenbook projections* (or Board of Governors Datasets). These documents consist of text, tables, and graphs. Our experiments suggest that setting a minimum threshold length of 25,000 bytes achieves an acceptable confidence level for approximating a file's entropy. The entropy approximation for 15,000 bytes is presented in Table 3, and for 25,000 bytes in Table 4, using the Greenbook PDF Dataset encrypted by specified ransomware. The bold values in Table 4 indicate the threshold value of 7.99. Note that only a subset of the files is displayed in this table.

Table 3: The first 15000 bytes of the file are read and the entropy is computed.

File Name	Total File Entropy	GandCrab	locked	OFFWHITE	az3zg	avdn	eswasted	WNCRY
GS-1966-01-11.pdf	7.7787168	7.98654117	7.9871808	7.98838185	7.98779854	7.98884194	7.9877807	7.98771588
GS-1966-02-08.pdf	7.7957867	7.98679319	7.98813517	7.98663561	7.98738209	7.98744374	7.98915027	7.98577155
GS-1966-03-01.pdf	7.71154378	7.98849562	7.98780239	7.98756999	7.98761761	7.98670168	7.9885083	7.98640685
GS-1966-03-22.pdf	7.70743395	7.98853245	7.98775498	7.98745479	7.98805904	7.98589657	7.98739649	7.98965494
GS-1966-04-12.pdf	7.70909013	7.9864649	7.98792679	7.98629035	7.98944624	7.98654917	7.98642588	7.9882894
GS-1966-05-10.pdf	7.71872743	7.9873719	7.98942611	7.9889985	7.98769273	7.98756768	7.98744401	7.98777511
GS-1966-06-07.pdf	7.71313681	7.98778132	7.98508746	7.9872488	7.98805607	7.9876873	7.98932451	7.98676996
GS-1966-06-28.pdf	7.7869504	7.98654693	7.98843465	7.98801858	7.98777954	7.98756831	7.98874163	7.98857874
GS-1966-07-26.pdf	7.78627432	7.98702218	7.9866608	7.98770975	7.98996563	7.98626002	7.98707602	7.98900321
GS-1966-08-23.pdf	7.72829092	7.98679314	7.98909114	7.98884781	7.98636617	7.9893225	7.98484725	7.9857893

Entropy of Compressed Files: ***JPEG files:*** Compressed files like JPEGs, which employ lossy compression, inherently have high entropy due to the reduction of character redundancy in image files. For our experiments, we utilised the *INRIA Holidays JPG dataset* [16], comprising 812 JPG images occupying 1.1GB of storage. The entropy of each file in this dataset was calculated, revealing that the highest entropy value, 7.95245506, was derived from the initial $25,000$ bytes of a file. Notably, although JPG files exhibit high entropy values, they are not as high as those of encrypted files. This difference is attributed to non-uniform character distribution probabilities in JPG files. For example, the *Null* (or 00) character is the most frequent in most JPG files, while in others, the *nbsp* (or 255) character predominates.

TIFF Files: TIFF file compression uses adaptive dictionary algorithms, like LZW compression. Our experiments aimed to explore the statistical properties of TIFF files. We used a sample of 87 TIFF images from a medical database, specifically the OME-TIFF dataset. The results showed that the highest file entropy was 5.20967171, significantly lower than the $H_{\max}$ value of 8. This outcome suggests a non-uniform probability distribution of characters in TIFF images.

Table 4: The first 25000 bytes of the file are read and the entropy is computed.

File Name	**GandCrab**	**locked**	**OFFWHITE**	**az3zg**	**avdn**	**eswasted**	**WNCRY**
GS-1966-01-11.pdf	**7.99**338252	**7.99**245514	**7.99**215868	**7.99**127136	**7.99**254762	**7.99**140233	**7.99**28472
GS-1966-02-08.pdf	**7.99**100436	**7.99**313812	**7.99**253868	**7.99**211495	**7.99**210098	**7.99**252393	**7.99**172881
GS-1966-03-01.pdf	**7.99**22206	**7.99**283364	**7.99**205808	**7.99**274639	**7.99**110681	**7.99**350538	**7.99**23769
GS-1966-03-22.pdf	**7.99**364135	**7.99**293043	**7.99**278268	**7.99**234747	**7.99**238042	**7.99**19224	**7.99**362411
GS-1966-04-12.pdf	**7.99**20972	**7.99**251146	**7.99**17135	**7.99**382699	**7.99**163157	**7.99**097479	**7.99**184339
GS-1966-05-10.pdf	**7.99**399683	**7.99**291601	**7.99**391016	**7.99**315747	**7.99**326597	**7.99**235577	**7.99**343543
GS-1966-06-07.pdf	**7.99**248154	**7.99**199274	**7.99**239283	**7.99**317553	**7.99**241421	**7.99**336579	**7.99**261136
GS-1966-06-28.pdf	**7.99**300174	**7.99**353394	**7.99**285691	**7.99**322543	**7.99**293433	**7.99**270476	**7.99**301798
GS-1966-07-26.pdf	**7.99**31377	**7.99**344906	**7.99**250657	**7.99**272543	**7.99**232356	**7.99**183047	**7.99**371094
GS-1966-08-23.pdf	**7.99**218428	**7.99**356795	**7.99**311067	**7.99**240853	**7.99**295244	**7.99**21548	**7.99**167112

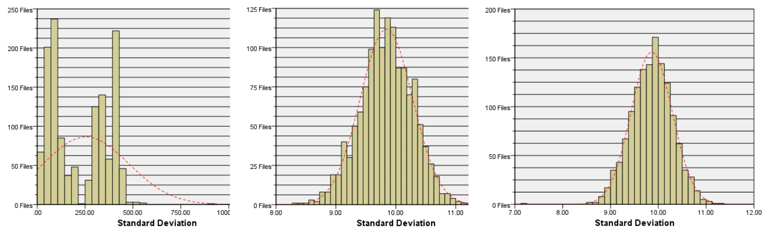
For example, the *Null* character frequently appears with the highest probability in these files.

High-performance compression algorithms, such as those used in TAR and ZIP applications, effectively reduce character redundancy to the extent that the file entropy approaches $H_{\max}$. This resemblance creates a significant challenge in distinguishing compressed files from those encrypted by ransomware, particularly if entropy is the sole characteristic under consideration. To address this issue, user applications and DIM can negotiate an acceptable level of compression quality. This can be achieved by slightly skewing the byte probability distribution, thereby introducing a small redundancy ε. Consequently, the file entropy is adjusted to $H(f) - \varepsilon$, where $H(f)$ represents the entropy of the compressed file. The introduced redundancy ε should be minimal to avoid compromising compression quality, yet substantial enough to enable DIM to differentiate between compressed and encrypted files.

Distribution of Symbols in Encrypted Files: In encrypted files, characters behave as if they are random, making it impossible to extract any sensitive information from them. Essentially, any two adjacent characters should be regarded as independent random variables X and Y, where X does not reveal any information about Y and vice versa. Consequently, characters in encrypted files tend to follow a uniform probability distribution. To test this theory in the context of ransomware encryption, we selected six prevalent ransomware variants and used them to encrypt a collection of three datasets: Greenbook projections, INRIA Holidays, and OME-TIFF. We then computed the standard deviations of characters for the first 25 kilobytes (KB) of each encrypted file.

Figure 1 shows the distribution of symbols in both unencrypted and ransomware-encrypted files, confirming that characters in encrypted files are indeed uniformly distributed. Additionally, Table 5 presents the confidence intervals for both encrypted and unencrypted files, calculated for the first 25 KB of the 8010 encrypted and 1335 unencrypted files, corresponding to confidence levels ranging from 99% to 80%.

Ransomware with Low Entropy: Modern ransomware variants achieve higher throughput and lower latency by employing parallel threads, allowing them to fetch data from the HDD and encrypt it concurrently without waiting for other threads to complete. A notable example of such ransomware is Dharma (also known as CrySIS), which has been active since 2016. With over a hundred different versions, our analysis reveals that this multi-threaded ran-



(a) Unencrypted files (b) AVDN (c) Eswasted

Fig. 1: The Standard Deviation (σ) of PDF, JPG, and TIFF files was computed from the first 25 kilobytes (KB) of each file. Sub-Figure 1a illustrates that symbols in unencrypted files appear loosely distributed, whereas in encrypted files, symbols are more uniformly distributed 1b and 1c

Table 5: Average Confidence Interval of the first 25000 Bytes of 8010 encrypted and 1335 unencrypted files. The data set is encrypted by six different ransomware families.

	Encrypted files			**Unencrypted files**		
	Lower bound	**Mean**	**Upper bound**	**Lower bound**	**Mean**	**Upper bound**
Confidence Interval 99%	96.02748789	97.65625	99.31896542	35.4255017	97.65625	159.8869983
Confidence Interval 95%	96.38530503	97.65625	98.92719497	50.43465809	97.65625	144.8778419
Confidence Interval 90%	96.590831	97.65625	98.721669	58.07091603	97.65625	137.241584
Confidence Interval 85%	96.72440466	97.65625	98.58809534	63.03380651	97.65625	132.2786935
Confidence Interval 80%	96.8330564	97.65625	98.4794436	66.84643556	97.65625	128.4660644

somware encrypts eight files simultaneously. Additionally, it writes 256 KB of the *null* character at the start of image files (like JPG and TIFF), likely to circumvent encryption detection through entropy analysis. This insertion of null bytes effectively lowers the entropy values of files. To counteract this, we employ a sliding window technique while reading the initial 25 KB of a file to detect encryption. The specifics of this sliding window technique are explained in the following section.

Context Triggered Piecewise Hashing: Similarity-preserving hashing, also known as fuzzy hashing, has been used in malware analysis [21]. In this study, we propose employing fuzzy hashing to detect buffers containing malicious content. Consider a collection of files $F = \{f_1, f_2, \ldots, f_m\}$, with each file f_m requiring a finite set of buffers $\{b_1, b_2, \ldots, b_n\}$, where each buffer b_n is 4096 bytes in size. Let $f_{\text{hash}}(f_m(b_n))$ represent the hashing signature (digest) of the b_n buffer contents of f_m. Algorithm 1 computes similarity-preserving hash signatures for all files in the collection F stored on the HDD. This approach allows us to: (1) Detect files with correct similarity-preserving hash signatures before writing them to the HDD. (2) Identify padding data added by ransomware to reduce file entropy. (3) Make it difficult for ransomware to predict which buffers (hash signatures) DIM uses to identify encrypted contents. (4) Determine if ransomware has deleted and rewritten all buffers of a file, indicating that the original file is irreversibly damaged and no ransom should be paid. (5) Update buffers' fuzzy signatures after each legitimate file write operation. (6) Distinguish existing files from new ones, even if the new files exhibit high entropy (such as ZIP or encrypted files).

4 Experimental Setup for DIM Efficacy and Performance Evaluation

To validate the DIM concept, a proof-of-concept setup was created to assess its efficacy and performance. The system consists of a Dell desktop with an Intel Core i7-4770 CPU @ 3.40 GHz, 16 GB RAM, and 4 cores with 8 logical processors. It hosts two virtual machines on a VMware hypervisor, each configured with 4 GB RAM and 4 processor cores. The VMs run on Ubuntu 20.04 Desktop and Windows 10 Education N. The first VM, representing a typical ransomware target, runs Windows 10, chosen for its prevalence in ransomware attacks due to its widespread use. The second VM uses Ubuntu with Samba (SMB protocol) to simulate network storage, protected by DIM, and contains a shared folder with 1335 files of various types, including PDF, JPG, and TIFF. This setup is designed to evaluate DIM's potential in a controlled environment that mimics real-world conditions.

Algorithm 1 collects buffers at random for files and computes their fuzzy hashing signatures. There is a collection of fuzzy signatures for each file that exists on storage.

```
1: Input: $F = \{f_1, f_2, \ldots, f_m\}$ {#All files on HD}
2: Output: Collection of all files fuzzy signatures
3: $FuzzyCollection[]$ {#Global}
4: $files[] \leftarrow f_m$
5: $b_n := 4096$
6: $n := 1$
7: for $f_m$ in $files$ do
8:    {#7 buffers need for entropy}
9:    $Random \leftarrow range(1, 7)$
10:    $Read\ f_m$ :
11:    while $Not\ EOF$ do
12:       $temp \leftarrow read\ f_m(b_n)$ {# Read file 4096 }
13:       if $n = Random$ then
14:          $FuzzyCollection := f_{hash}(temp)$
15:          $break$
16:       end if
17:       $n := +1$
18:    end while
19:    $Close f_m$
20:    $n := 1$
21: end for
```

A key component of our setup is the FUSE (Filesystem in Userspace) framework. FUSE is widely used, with at least 100 different FUSE-based file systems available online. It has two main components: a kernel module and a user-level daemon. Once the kernel module is loaded, it registers with the Linux Virtual

File System (VFS) as a FUSE file system driver, enabling the creation of file systems in user space rather than kernel space. FUSE is popular for custom file system development.

Several FUSE-based file systems have been developed by various vendors, such as Google Cloud Storage FUSE and s3fs-fuse. The FUSE kernel module, integrated into the Linux kernel, serves as an intermediary for specific file systems implemented by different user-level daemons. When a user application accesses a FUSE file system, the Linux VFS directs the request to the FUSE kernel driver. This driver creates a FUSE request, adds it to a queue, and typically pauses the requesting process while the request is processed. The user-level FUSE daemon retrieves the request from the kernel queue via /dev/fuse and handles it [31]. FUSE is also available for Windows systems under the name WinFsb.

DIM Pass-through Module for Encryption Detection: *(Question: Where can DIM be embedded in a resilient system?)* To address this question, we developed a pass-through module that interfaces with the FUSE file system. This module is designed to intercept and examine the read/write buffers within the network share file that is under DIM protection. It has all necessary permissions to manipulate files, including access, chmod, open, create, read, and write, making it a fully functional file system.

Three principal directories **A** $\Leftrightarrow$ **B** $\Leftrightarrow$ **C** interact with the pass-through module. Directory **A** contains all files but is not directly accessible to users. Directory **B**, configured as a shared mounting point (user-level daemon) with the Samba protocol, is accessible to users over the network. Directory **C** is located in the user environment. Note that while both **A** and **B** are on the Ubuntu VM, **C** resides on the Windows VM, representing the user's machine. When a process, initiated by a user or ransomware, opens or writes to a file, the module inspects the file's write buffers.

(Next question: How can DIM detect and identify encrypted write buffers, regardless of the encryption algorithm used?) The pass-through module inspects all write buffers of files originating from directory **C**. It determines whether the contents of a buffer are encrypted before these contents are written to the HDD. Our analysis shows that at least 25 KB of a file is needed to compute reliable file entropy. To achieve this, the module reads seven buffers (each 4096 bytes) to identify encrypted content and calculates the file's entropy, standard deviation, and confidence interval.

Consider a scenario where ransomware reads a 100 KB file, encrypts its contents, and stores it in memory, i.e., the buffer cache. The system then writes these encrypted bytes to the HDD using a default block size, which in this case is 4 KB. Let $\Omega = \{b_1, b_2, \ldots, b_n\}$ represent a collection of n buffers. The process of encryption detection employed by the pass-through module is outlined in Algorithm 2.

DIM Encoding Expansion Module: Compression is a process that assigns shorter bit encodings to more probable symbols, while less probable symbols receive longer bit encodings [25]. There's extensive literature on coding schemes for lossless compression, such as Huffman, arithmetic, the Lempel–Ziv

Algorithm 2 identifies encrypted buffers. It reads seven buffers (chosen at random) to determine whether their signatures belong to the fuzzy hashing collection. It also measures the entropy of the seven buffers.

```
1: **Input**: An encrypted buffer $b_i \in \{b_1, b_2, ..., b_n\}$
2: **Input**: Cumulative buffer $byte[]\ Buffers$
3: **Input**: $n$ the number of encrypted buffers arrived
4: $i := 0$
5: **for** $n := 1$ *to* 7 **do** { # Count for 7 buffer arrivals}
6:    $temp := FuzzyHash(b_n)$
7:    { # From Fuzzy }
8:    **for** $item \leftarrow FuzzyCollection[i]$ **do**
9:       $Compare := FuzzyHash.compare(temp, item)$
10:      { # No match at FuzzyHash}
11:      **if** $Compare = 0$ **then**
12:         $i := i + 1$
13:      **end if**
14:      **if** $i = length\ of\ FuzzyCollection[]$ **then**
15:         {#the file is new}
16:         i := 0
17:         $break$
18:      **end if**
19:   **end for**
20:   Insert $b_n$ to $Buffers[]$ {#*Accumulate buffers*}
21: **end for**
22: $X := H(Buffers[b_1, .., b_7])$
23: **if** $(X \geq 7.99)$ & (*no match in FuzzyCollection*) **then**
24:    $buffers\ are\ encrypted...$
25:    $start\ expanding\ buffers...$
26:    $break$
27: **else**
28:    write buffer to HDD if conditions are not met
29: **end if**
```

family (refer to [25]), and the asymmetric numeral system (ANS) [7], which has been found to be up to 30 times faster than earlier methods.

How can DIM obstruct the functionality of ransomware? Compression is not only crucial for shortening transmission times, saving storage space, and accelerating encryption, but it also presents an interesting defensive mechanism against ransomware. When symbol occurrences in a file do not adhere to expected statistical distributions, compression can inadvertently expand the file. This expansion, typically undesirable, can be leveraged as a defense against ransomware. For instance, if DIM is requested by ransomware to store an encrypted file on the HDD, DIM can apply a compression algorithm using non-uniform symbol statistics, potentially derived from or similar to the file before encryption. This DIM-induced compression expands the file, thereby occupying the ransomware encryption engine. The extent of this expansion can be regulated by altering symbol probabilities, growing significantly when more frequent symbols are assigned very low probabilities. In theory, this expansion can reach sizes of gigabytes (GBs) or even terabytes (TBs), effectively slowing down the ransomware encryption engine without needing to locate the elusive ransomware process. This delay facilitates the triggering of an alarm for system or human intervention. Increasing symbol probabilities could potentially disable the ransomware functionality through excessive I/O load or insufficient storage on a DIM-controlled volume.

The primary function of the encoding expansion module is to engage a ransomware encryption engine by supplying it with an ever-expanding bitstream. To demonstrate this, we developed a simple algorithm that transforms a stream of uniformly distributed symbols (as in an encrypted file) into an exceedingly long bitstream. Suppose ransomware requests to write an encrypted file to a victim's HDD. Before proceeding, let's define some notations. Let $C = \Delta = \{c_1, c_2, \ldots, c_m\}$ represent a collection of characters (Unicode). For a sequence $S = (s_1, \ldots, s_n)$, where $s_i \in \Delta$, $|S|_{c_i}$ denotes the number of occurrences of c_i in S. The probability of occurrence of c_i in S is given by $P(c_i) = \frac{|S|_{c_i}}{n}$.

Algorithm 3 Buffer encryption expansion encoding module.

```
Input: $C = \{c_1, c_2, ..., c_m\}$ be a sequence of $n$ random encrypted symbols
1: Initialize: precision := 1000000
2: Initialize: $S[] := null$
3: Initialize: $S_{c_i}[] := 0$
4: for ($n := 0$ to $C.length$) do
5:     $key := C[n]$
6:     $S_{c_i}[key] += 1$
7:     if ($key\ not\ in\ S[]$) then
8:         $S[] += C[n]$
9:     end if
10: end for
11: $low := 0$
12: $high := 1/S_{c_i}[key]$
13: for ($key\ in\ sorted\ S[]$) do
14:     $cdf\_range[key] := [low, high)$
15:     $low := high$
16:     { # increasing bitstream exponentially.}
17:     $high := high + (\frac{1}{S_{c_i}[key]^{precision}})$
18: end for
19: {# Compute probability distribution function (pdf)}
20: for ($key\ in\ sorted\ S[]$) do
21:     $pdf[key] := \frac{1}{S_{c_i}[key]}$
22: end for
23: $LowerBound := 0$
24: $UpperBound := 1$
25: while (there are still symbols in $S[]$ to encode) do
26:     $CurrentRange := UpperBound - LowerBound$
27:     $UpperBound := LowerBound + (CurrentRange * cdf\_range[key])$
28:     $LowerBound := LowerBound + (CurrentRange * cdf\_range[key-1])$
29:     {#Update the probability tables}
30:     for ($key\ in\ sorted\ pdf$) do
31:         $pdf[key] := \frac{1}{S_{c_i}[key]}$
32:     end for
33:     for ($key\ in\ sorted\ cdf\_range$) do
34:         $pdf[key] := [low, high)$
35:         $low := high;$
36:     end for
37: end while
38: {# write repeat encoded LowerBound.}
39: $\prod_{LowerBound}^{i=1000} = 0$
```

Algorithm 3 operates within the interval $[0, 1)$ of real numbers. As the set C expands, the interval needed to represent it decreases, while the number of bits required to specify the interval increases. Symbols $\{c_1, c_2, \ldots, c_m\}$ from the set of encrypted symbols reduce the interval size in accordance with their probabilities. The algorithm is designed to allow error-free decoding, even when the expansion encoding module has encoded unencrypted symbols. To ensure this, we enhanced the precision of the symbol distribution range table.

Let's consider two examples to illustrate the algorithm. For a set of random unencrypted symbols $\{a, e, i, n, t\}$, arithmetic encoding is demonstrated in Table 6. Each symbol s is allocated a unique interval, the length of which is proportional to the symbol's probability. Suppose our encoding expansion module receives the sequence (e, a, i) of symbols. Upon encountering the first symbol e, the module narrows the initial interval $[0, 1)$ to $[0.2, 0.5)$. For the second symbol a, it proportionally shortens the current interval $[0.2, 0.5)$ to $[0.2, 0.26)$. The lower interval bound is $p_{s_{1(\text{low})}}$, while the upper interval bound is $p_{s_{1(\text{low})}} + p_{s_0} \cdot p_{s_1}$. For the final symbol i, the current interval $[0.2, 0.26)$ is further narrowed to $[0.23, 0.236)$. The lower interval bound becomes $p_{s_{1(\text{low})}} + p_{s_1} \cdot p_{s_2}$, and the upper interval bound is $(p_{s_{1(\text{low})}} + p_{s_1} \cdot p_{s_2}) + p_0 \cdot p_1 \cdot p_2$. Ultimately, the module stores the lower bound 0.23 as the encoding of the sequence (e, a, i).

Table 6: Example of fixed normal symbols distribution range model.

Symbols	**Probability**	**Range**	
S	p_{s_n}	$p_{s_{(low)}}$	$p_{s_{(high)}}$
a	$p_{s_0} \leftarrow$ 0.2	[0.0 ,	0.2)
e	$p_{s_1} \leftarrow$ 0.3	[0.2 ,	0.5)
i	$p_{s_2} \leftarrow$ 0.1	[0.5 ,	0.6)
n	$p_{s_3} \leftarrow$ 0.3	[0.6 ,	0.9)
t	$p_{s_4} \leftarrow$ 0.1	[0.9 ,	1.0)

Now, consider a sequence of symbols $\{x, y, z, k, !, \text{Space}\}$. Let's assume the probability distribution of these symbols is detailed in Table 7. Figure 2-A depicts the probability density function range table with a precision of 15, as outlined in Algorithm 4. We also present the result of expanding encrypted symbols $S = \{x, y, z, k, !, \text{Space}\}$, which are 6 bytes in length. Using a precision of 500, the compression output increases to 672 bytes, as shown in Figure 2-B. A larger precision can inflate the compression output to file sizes on the order of gigabytes (GB). We tested a precision of one million, which expanded a 5-byte file into a 10-GB file before the system crashed — an outcome potentially beneficial during a ransomware attack. In theory, it's possible to expand a short file into a much larger one, potentially on the order of terabytes (TB).

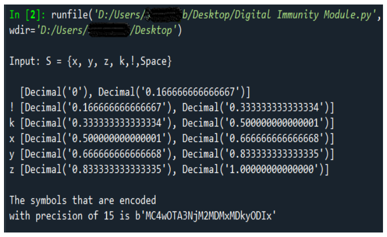
Recovery of Corrupted Files: Modern ransomware variants overwrite original files upon encryption, deleting the originals and saving encrypted copies to storage. DIM addresses this by storing files being accessed in a secure temporary directory, protecting them from immediate encryption. This module reads file buffers and compiles statistics to facilitate file recovery. If the files are unencrypted, DIM deletes the temporary directory; if they are encrypted, it retains the directory to recover the files. This design ensures minimal impact on HDD storage and computational load, providing an effective defense against current

Algorithm 4 computes encoding table with higher precision with the longer bitstream.

```
1: **Precision:** 15
2: **Input:** $S = [x, y, z, k, !, Space]$
3: **Initialize:** $S_{c_i}[]$ {# Algorithm 4 defines it}
4: $low := 0$
5: $high := 1/S_{c_i}[key]$
6: **for** $(key\ in\ sorted\ S[])$ **do**
7:     $cdf\_range[key] := [low, high)$
8:     $low := high$
9:     $high := high + (\frac{1}{S_{c_i}[key]^{precision}})$
10: **end for**
```

Table 7: Probability density function range table for each symbol with equal distribution probability ($p_0, \cdots, p_5 = 0.166667$).

Symbols	***Probability***	***Range*** $\leftarrow 1/\|S\|$	
S	p_s	$p_{s_{low}}$	$p_{s_{high}}$
Space	p_0	[0.000000,	0.166667)
!	p_1	[0.166667,	0.333334)
k	p_2	[0.333334,	0.500001)
x	p_3	[0.500001,	0.666668)
y	p_4	[0.666668,	0.833335)
z	p_5	[0.833335,	1.000000)

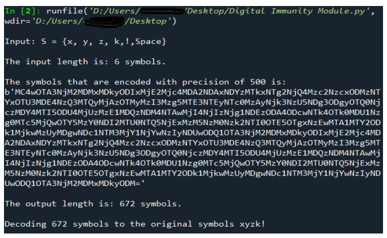


```
In [2]: runfile('D:/Users/[redacted]b/Desktop/Digital Immunity Module.py',
wdir='D:/Users/[redacted]/Desktop')

Input: S = {x, y, z, k,!,Space}

  [Decimal('0'), Decimal('0.166666666666667')]
! [Decimal('0.166666666666667'), Decimal('0.333333333333334')]
k [Decimal('0.333333333333334'), Decimal('0.500000000000001')]
x [Decimal('0.500000000000001'), Decimal('0.666666666666668')]
y [Decimal('0.666666666666668'), Decimal('0.833333333333335')]
z [Decimal('0.833333333333335'), Decimal('1.00000000000000')]

The symbols that are encoded
with precision of 15 is b'MC4wOTA3NjM2MDMxMDkyODIx'
```



```
In [2]: runfile('D:/Users/[redacted]/Desktop/Digital Immunity Module.py',
wdir='D:/Users/[redacted]/Desktop')

Input: S = {x, y, z, k,!,Space}

The input length is: 6 symbols.

The symbols that are encoded with precision of 500 is:
b'MC4wOTA3NjM2MDMxMDkyODIxMjE2Mjc4MDA2NDAxNDYzMTkxNTg2NjQ4Mzc2NzcxODMzNT
YxOTU3MDE4NzQ3MTQyMjAzOTMyMzI3Mzg5MTE3NTEyNTc0MzAyNjk3NzU5NDg3ODgyOTQ0Nj
czMDY4MTI5ODU4MjUzMzE1MDQzNDM4NTAwMjI4NjIzNjg1NDEzODA4ODcwNTk4OTk0MDU1Nz
g0MTc5MjQwOTY5MzY0NDI2MTU0NTQ5NjExMzM5NzM0Nzk2NTI0OTE5OTgxNzEwMTA1MTY2OD
k1MjkwMzUyMDgwNDc1NTM3MjY1NjYwNzIyNDUwODQ1OTA3NjM2MDMxMDkyODIxMjE2Mjc4MD
A2NDAxNDYzMTkxNTg2NjQ4Mzc2NzcxODMzNTYxOTU3MDE4NzQ3MTQyMjAzOTMyMzI3Mzg5MT
E3NTEyNTc0MzAyNjk3NzU5NDg3ODgyOTQ0NjczMDY4MTI5ODU4MjUzMzE1MDQzNDM4NTAwMj
I4NjIzNjg1NDEzODA4ODcwNTk4OTk0MDU1Nzg0MTc5MjQwOTY5MzY0NDI2MTU0NTQ5NjExMz
M5NzM0Nzk2NTI0OTE5OTgxNzEwMTA1MTY2ODk1MjkwMzUyMDgwNDc1NTM3MjY1NjYwNzIyND
UwODQ1OTA3NjM2MDMxMDkyODM='

The output length is: 672 symbols.

Decoding 672 symbols to the original symbols xyzk!
```

Fig. 2: Left: The probability density function range table with precision of 15. Right: Manipulating the PDF range table at a 500 precision level increases input from 6 to 672 symbols.

and future ransomware tactics without focusing on the specific coding of the ransomware.

Evaluation and Results: We gathered a substantial collection of ransomware variants, totaling 1,182 instances across multiple families, from databases like VirusShare, MalwareBazaar, and Kaggle. We selected 100 executable variants from 75 different families for repeated testing against our Defense Mechanism (DIM). The effectiveness of DIM in preventing file encryption was evaluated against a dataset comprising 32.6 GB and 11,928 different file types. Our approach to halt ransomware involves inducing CPU and memory resource starvation by manipulating the probability density function (PDF) to significantly increase resource demands, as shown in Figures 3 and 4, leading to ransomware process termination depicted in Figures 5a and 5b. DIM analyzes file buffers

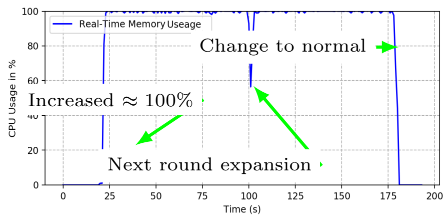


Fig. 3: CPU starvation resulting from the manipulation of the probability density function range table (precision of 100,000) during a ransomware attack, permitting the expansion of buffers without terminating the overwriting process by the ransomware.

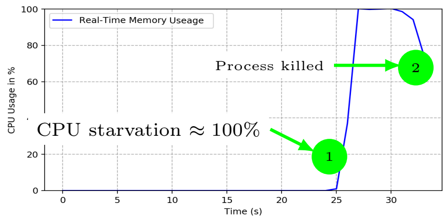


Fig. 4: CPU and memory starvation resulting from the manipulation of the probability density function range table (precision of 10,000,000) during a ransomware attack, killing the malicious process.

using a sliding window technique, effectively detecting encryption, particularly for complex variants like Dharma, by extending its analysis to more windows.

False Positive and False Negative Evaluation: DIM's file expansion strategy effectively combats ransomware by transforming small files into significantly larger ones, stopping encryption and facilitating recovery in cases of false positive detections. Our evaluations reveal no false negatives with the latest ransomware variants. The entropy of encrypted data usually exceeds 7.99, with DIM ensuring effective file protection on shared drives as well. Resource usage analysis shows that DIM maintains minimal CPU usage during idle states, with consistent performance under load, detailed in Figure 7. A video demonstration of DIM's capabilities is available ***here***.

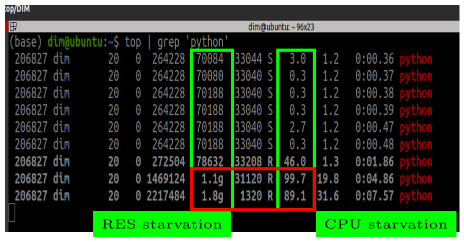
5 Comparison and Discussion

Existing ransomware detection models, focusing on system-level I/O behaviors, face significant challenges, including high false positives due to similarities between ransomware and legitimate applications like file compressors and backup software, leading to unwanted disruptions [2] [9] [19] [27]. These models also suffer from performance overhead, particularly when monitoring high I/O operations, which affects system performance and user experience [9] [27]. Additionally, ransomware can employ evasion techniques like memory mapped I/O or fileless attacks, complicating detection [2] [9]. Traditional approaches are generally reactive, providing a window for ransomware to inflict damage [18] [27], and struggle with scalability in large environments [2] [9] [19]. In contrast, ***DIM*** utilizes a FUSE file system with a statistical analysis approach via sliding windows to identify encryption activities, effectively reducing false positives and eliminating false negatives during ransomware tests. Despite causing some delays in file system operations, DIM does not interrupt benign activities and adapts to detect evasion techniques in real-time. Its scalability and responsiveness make it suitable for large-scale deployments and capable of adapting to new ransomware tactics [27] [19].

6 Conclusion, Limitations and Future Work

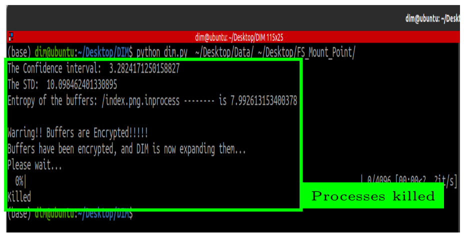
This paper presents the Digital Immunity Module (DIM), which effectively counters encrypted buffers in file systems using real-time analysis and inverse arithmetic coding. Tested against 75 ransomware families, DIM had no false negatives and provided recovery options for false positives. It protected data against Black Basta ransomware's encryption techniques. Future work will enhance anomaly detection in file encryption using online machine learning to minimize entropy and handle Base-64 encoding, with early tests showing a 33.33% increase in file size for lowered entropy. Findings support the efficacy of online learning classifiers in detecting statistical anomalies, leading to further development of a hybrid model.

A Resource Overload and process termination approach



```
(base) dim@ubuntu:~$ top | grep 'python'
 206827 dim      20   0  264228  70084 33044 S   3.0  1.2   0:00.36 python
 206827 dim      20   0  264228  70080 33040 S   0.3  1.2   0:00.37 python
 206827 dim      20   0  264228  70188 33040 S   0.3  1.2   0:00.38 python
 206827 dim      20   0  264228  70188 33040 S   0.3  1.2   0:00.39 python
 206827 dim      20   0  264228  70188 33040 S   2.7  1.2   0:00.47 python
 206827 dim      20   0  264228  70188 33040 S   0.3  1.2   0:00.48 python
 206827 dim      20   0  272504  78632 33208 R  46.0  1.3   0:01.86 python
 206827 dim      20   0 1469124   1.1g 31120 R  99.7 19.8   0:04.86 python
 206827 dim      20   0 2217484   1.8g  1320 R  89.1 31.6   0:07.57 python
```

(a) DIM's response to illegal encryption: Memory usage surged from 79MB to 1.61GB, with CPU utilization peaking at 89.1%, leading to abrupt process termination.



```
(base) dim@ubuntu:~/Desktop/DIM$ python dim.py  ~/Desktop/Data/ ~/Desktop/FS_Mount_Point/
The Confidence interval:  3.2824171250158827
The STD:  10.098462401330895
Entropy of the buffers: /index.png.inprocess -------- is 7.992613153400378

Warring!! Buffers are Encrypted!!!!!
Buffers have been encrypted, and DIM is now expanding them...
Please wait...
  0%|                                                    | 0/4096 [00:00<?, ?it/s]
Killed
(base) dim@ubuntu:~/Desktop/DIM$
```

(b) Killing of a malicious process because of resource starvation - read together with Figure 5a.

Fig. 5: Comparison between two scenarios.

B Sliding approach and CPU overheard analysis

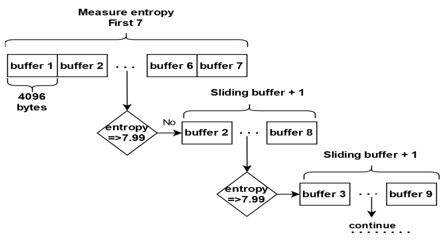


Fig. 6: DIM uses sliding windows to deal with ransomware, which may insert characters to reduce entropy.

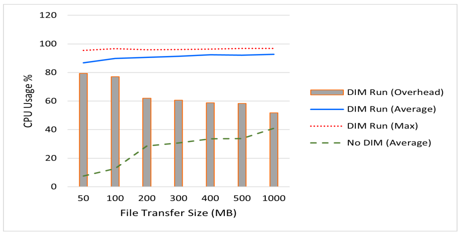


Fig. 7: The results illustrate the server CPU overhead (DIM) during file transfers between a client and a server.

References

1. Al-rimy, B.A.S., Maarof, M.A., Prasetyo, Y.A., Shaid, S.Z.M., Ariffin, A.F.M.: Zero-day aware decision fusion-based model for crypto-ransomware early detection. International Journal of Integrated Engineering **10**(6) (2018)
2. Baek, S., Jung, Y., Mohaisen, A., Lee, S., Nyang, D.: 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). pp. 875–884 (July 2018). `https://doi.org/10.1109/ICDCS.2018.00089`
3. Baldwin, J., Dehghantanha, A.: Leveraging Support Vector Machine for Opcode Density Based Detection of Crypto-Ransomware, pp. 107–136. Springer International Publishing, Cham (2018). `https://doi.org/10.1007/978-3-319-73951-9_6`, `https://doi.org/10.1007/978-3-319-73951-9_6`
4. Bijitha, C.V., Sukumaran, R., Nath, H.V.: A survey on ransomware detection techniques. In: Sahay, S.K., Goel, N., Patil, V., Jadliwala, M. (eds.) Secure Knowledge Management In Artificial Intelligence Era. pp. 55–68. Springer Singapore, Singapore (2020)
5. Cabaj, K., Gregorczyk, M., Mazurczyk, W.: Software-defined networking-based crypto ransomware detection using http traffic characteristics. Computers & Electrical Engineering **66**, 353–368 (2018). `https://doi.org/https://doi.org/10.1016/j.compeleceng.2017.10.012`, `https://www.sciencedirect.com/science/article/pii/S0045790617333542`
6. Cabaj, K., Mazurczyk, W.: Using software-defined networking for ransomware mitigation: The case of cryptowall. IEEE Network **30**(6), 14–20 (2016). `https://doi.org/10.1109/MNET.2016.1600110NM`
7. Camtepe, S., Duda, J., Mahboubi, A., Morawiecki, P., Nepal, S., Pawłowski, M., Pieprzyk, J.: Compcrypt–lightweight ans-based compression and encryption. IEEE Transactions on Information Forensics and Security **16**, 3859–3873 (2021). `https://doi.org/10.1109/TIFS.2021.3096026`
8. Chen, Q., Bridges, R.A.: Automated behavioral analysis of malware: A case study of wannacry ransomware. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 454–460 (2017). `https://doi.org/10.1109/ICMLA.2017.0-119`
9. Continella, A., Guagnelli, A., Zingaro, G., De Pasquale, G., Barenghi, A., Zanero, S., Maggi, F.: Shieldfs: A self-healing, ransomware-aware filesystem. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. p.

336–347. ACSAC '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2991079.2991110, https://doi.org/10.1145/2991079.2991110

10. Cusack, G., Michel, O., Keller, E.: Machine learning-based detection of ransomware using sdn. In: Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks Network Function Virtualization. p. 1–6. SDN-NFV Sec'18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180465.3180467, https://doi.org/10.1145/3180465.3180467
11. Genç, Z.A., Lenzini, G., Ryan, P.Y.A.: No random, no ransom: A key to stop cryptographic ransomware. In: Giuffrida, C., Bardin, S., Blanc, G. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 234–255. Springer International Publishing, Cham (2018)
12. Gómez-Hernández, J., Álvarez González, L., García-Teodoro, P.: R-locker: Thwarting ransomware action through a honeyfile-based approach. Computers & Security **73**, 389–398 (2018). https://doi.org/https://doi.org/10.1016/j.cose.2017.11.019, https://www.sciencedirect.com/science/article/pii/S0167404817302560
13. Harikrishnan, N., Soman, K.: Detecting ransomware using gurls. In: 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC). pp. 1–6 (2018). https://doi.org/10.1109/ICAECC.2018.8479444
14. Homayoun, S., Dehghantanha, A., Ahmadzadeh, M., Hashemi, S., Khayami, R., Choo, K.K.R., Newton, D.E.: Drthis: Deep ransomware threat hunting and intelligence system at the fog layer. Future Generation Computer Systems **90**, 94–104 (2019). https://doi.org/https://doi.org/10.1016/j.future.2018.07.045, https://www.sciencedirect.com/science/article/pii/S0167739X17328467
15. Honda, T., Mukaiyama, K., Shirai, T., Ohki, T., Nishigaki, M.: Ransomware detection considering user's document editing. In: 2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA). pp. 907–914 (2018). https://doi.org/10.1109/AINA.2018.00133
16. Jegou, H., Douze, M., Schmid, C.: Hamming embedding and weak geometric consistency for large scale image search. In: Forsyth, D., Torr, P., Zisserman, A. (eds.) Computer Vision – ECCV 2008. pp. 304–317. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
17. Jung, S., Won, Y.: Ransomware detection method based on context-aware entropy analysis. Soft Computing **22**(20), 6731–6740 (may 2018). https://doi.org/10.1007/s00500-018-3257-z
18. Kharaz, A., Arshad, S., Mulliner, C., Robertson, W., Kirda, E.: UNVEIL: A large-scale, automated approach to detecting ransomware. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 757–772. USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz
19. Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the gordian knot: A look under the hood of ransomware attacks. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–24. Springer International Publishing, Cham (2015)
20. Lee, K., Lee, S.Y., Yim, K.: Machine learning based file entropy analysis for ransomware detection in backup systems. IEEE Access **7**, 110205–110215 (2019). https://doi.org/10.1109/ACCESS.2019.2931136

21. Li, Y., Sundaramurthy, S.C., Bardas, A.G., Ou, X., Caragea, D., Hu, X., Jang, J.: Experimental study of fuzzy hashing in malware clustering analysis. In: 8th Workshop on Cyber Security Experimentation and Test (CSET 15). USENIX Association, Washington, D.C. (Aug 2015), https://www.usenix.org/conference/cset15/workshop-program/presentation/li
22. Mahboubi, A., Ansari, K., Camtepe, S.: Using process mining to identify file system metrics impacted by ransomware execution. In: Mobile, Secure, and Programmable Networking. pp. 57–71. Springer International Publishing, Cham (2021)
23. Min, D., Park, D., Ahn, J., Walker, R., Lee, J., Park, S., Kim, Y.: Amoeba: An autonomous backup and recovery ssd for ransomware attack defense. IEEE Computer Architecture Letters **17**(2), 245–248 (2018). https://doi.org/10.1109/LCA.2018.2883431
24. Morato, D., Berrueta, E., Magaña, E., Izal, M.: Ransomware early detection by the analysis of file sharing traffic. Journal of Network and Computer Applications **124**, 14–32 (2018). https://doi.org/https://doi.org/10.1016/j.jnca.2018.09.013, https://www.sciencedirect.com/science/article/pii/S108480451830300X
25. Nelson, M., Gailly, J.L.: The data compression book, vol. 2. M&t Books New York (1996)
26. Netto, D.F., Shony, K.M., Lalson, E.R.: An integrated approach for detecting ransomware using static and dynamic analysis. In: 2018 International CET Conference on Control, Communication, and Computing (IC4). pp. 410–414 (2018). https://doi.org/10.1109/CETIC4.2018.8531017
27. Paik, J.Y., Choi, J.H., Jin, R., Wang, J., Cho, E.S.: A storage-level detection mechanism against crypto-ransomware. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 2258–2260. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3278491, https://doi.org/10.1145/3243734.3278491
28. Ramesh, G., Menen, A.: Automated dynamic approach for detecting ransomware using finite-state machine. Decision Support Systems **138**, 113400 (2020). https://doi.org/https://doi.org/10.1016/j.dss.2020.113400, https://www.sciencedirect.com/science/article/pii/S016792362030155X
29. Scaife, N., Carter, H., Traynor, P., Butler, K.R.B.: Cryptolock (and drop it): Stopping ransomware attacks on user data. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). pp. 303–312 (2016). https://doi.org/10.1109/ICDCS.2016.46
30. Takeuchi, Y., Sakai, K., Fukumoto, S.: Detecting ransomware using support vector machines. In: Proceedings of the 47th International Conference on Parallel Processing Companion. ICPP '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3229710.3229726, https://doi.org/10.1145/3229710.3229726
31. Vangoor, B.K.R., Tarasov, V., Zadok, E.: To FUSE or not to FUSE: Performance of user-space file systems. In: 15th USENIX Conference on File and Storage Technologies (FAST 17). pp. 59–72. USENIX Association, Santa Clara, CA (Feb 2017), https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor