CLEAR: Command Level Annotated Dataset for Ransomware Detection

Barak Bringoltz
Samsung Semiconductor Israel Research and Development Center
Tel Aviv, Israel
barak.b@samsung.com

Elisha Halperin
Samsung Semiconductor Israel Research and Development Center
Tel Aviv, Israel
elisha.h@samsung.com

Ran Feraru
Samsung Semiconductor Israel Research and Development Center
Tel Aviv, Israel
ran.feraru@samsung.com

Evgeny Blaichman
Samsung Semiconductor Israel Research and Development Center
Tel Aviv, Israel
evgeny.bl@samsung.com

Amit Berman
Samsung Semiconductor Israel Research and Development Center
Tel Aviv, Israel
amit.berman@samsung.com

Abstract

Over the last decade, ransomware detection has become a central topic in cybersecurity research. Due to ransomware's direct interaction with storage devices, analyzing I/O streams has become an effective detection method and represents a vital area of focus for research. A major challenge in this field is the lack of publicly accessible data featuring individual command labeling. To address this problem, we introduce the Command LEvel Annotated Ransomware (CLEAR) dataset, a large-scale collection of storage devices' stream data. The dataset comprises 1,045 TiB of I/O traffic data, featuring malicious traffic from 137 ransomware variants. It offers two orders of magnitude more I/O traffic data and one order of magnitude more ransomware variants than any other publicly accessible dataset. Importantly, it is the only dataset that individually labels each I/O command as either ransomware or benign activity. This labeling enables the use of advanced sequential models, which we show to outperform existing state-of-the-art models by up to 82% in data loss prevention. Additionally, this allows us to create new tasks, such as data recovery, by selectively reverting only the commands recognized as ransomware while preserving benign activity. The CLEAR dataset also includes supplementary auxiliary features derived from the data, which we demonstrate to improve performance through feature ablation studies. Lastly, a critical aspect

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

of any ransomware detection model is its robustness to new, unseen ransomware variants, as new strains constantly emerge. Therefore, we offer a benchmark based on our dataset to evaluate performance against unknown ransomware samples and illustrate its application across different models.

1 Introduction

Ransomware is a cyber attack that encrypts a victim's data until a ransom is paid. It is widespread, affecting healthcare, government, law enforcement, and education institutions [1], making it a serious threat to data privacy and national security. Ransomware detection is an active area of research in cybersecurity, with detection techniques falling into one of two categories: 'static' methods, which identify malicious software signatures, and 'behavioral' methods, which detect patterns in the behavior of computing systems infected by ransomware. In recent years, machine learning techniques have become widely used for ransomware detection, with features generated at different levels of the software stack – from network activity traces down to low-level disk I/O commands [2].

While focusing on disk I/O data restricts the available features, it allows the detection process to happen entirely on the disk hardware, which enjoys two major advantages. First, it would be impossible for the ransomware to shut the detection process down as it may do with software-based solutions. Second, the detection suite can lock the disk to avoid further damage and even reverse the damage already done, e.g., by buffering destructive operations, and discarding them if they are identified as malicious [3]. This has led to a growing body of research focusing on ransomware detection from I/O traces; however, most datasets in the field are closed, and publicly available datasets are limited in both their size and the scope of their ransomware coverage. Furthermore, all available datasets label the data only at the stream level – labeling streams containing any amount of ransomware activity as entirely ransomware, potentially mislabeling any benign commands present in the stream.

To address these issues, we introduce our Command LEvel Annotated Ransomware (CLEAR) dataset. It is a large-scale dataset, containing over 100 times more I/O traffic than other publicly available datasets, over 10 times more ransomware variants, and a variety of benign traffic patterns. All data in CLEAR is labeled at the command level, denoting the source of each command as benign or malicious. To demonstrate the effectiveness of our dataset, we train and evaluate different models on it – representing the current state-of-the-art, as well as our own baseline models – and show that models trained using command-based labeling outperform those trained without it. We then further analyze the results to show that the improved performance of the command-based models represents actual learning of individual command classification, and that this classification is not just a product of the command's features, but tied directly to the rich sequential structure of the data.

Finally, CLEAR's large collection of ransomware samples enables us to construct a benchmark for model performance on unseen ransomware. We measure a model's out-of-distribution performance by first clustering our ransomware samples and then training and evaluating on different clusters. Since new ransomware variants emerge regularly, robustness to unseen ransomware is a valuable metric. The CLEAR dataset can be found here [1], and the code and benchmark can be found here [2].

2 Storage I/O data

The literature on ransomware detection uses data acquired from different layers in the SW stack, among them the storage device's I/O data, which we define as the set of attributes associated with the computer commands reading from and writing to the storage device.

2.1 Structure of Data

The basic I/O attributes appearing in the literature and that we primarily consider are the following four: the floating point *timestamp* denotes the time of command initialization, $opcode$ is a categorical attribute denoting the command operation (read: R or write: W), the integer *size* is the number of

[1] https://www.kaggle.com/datasets/johndoenvme/clear-command-level-annotated-ransomware
[2] https://github.com/ravensorioles/rwdetection

bytes the command refers to, and the ordinal $offset$ denotes the byte offset on disk (disk addresses in the range $[offset, offset + size)$ constitute the storage space referred to by the command). In our case, this data comes from NVMe streams, but more generally, it can come from SATA streams as well.

Because the very nature of ransomware is to read data, encrypt it, and then destroy the original data, several works (e.g. [4]) calculate the overwritten byte volume to derive useful features. Inspired by such ideas, we complement the fundamental NVMe attributes with additional derived features. In particular, for each command c with $opcode(c) = W$, we identify the command c' that is the latest of all earlier commands with $opcode(c') = R$ and that has overlap OV_{WAR} with the bytes of c (WAR for 'Write-After-Read'). We also calculate the time-lapse $\Delta t_{WAR}(c) = timestamp(c) - timestamp(c')$ between c and c'. In the same vein, we calculate three additional overlaps and time lapses for $(opcode(c), opcode(c')) = (R, R), (R, W), (W, W)$. Similar derived attributes were defined in [5]. We pictorially demonstrate the way we calculate these derived properties in Table 5 of Appendix A.1. We also calculate the per-command time difference $\delta t(c) \equiv timestamp(c) - timestamp(c - 1)$ for each command c to keep track of the I/O rate.

Finally, we note that it is often useful to have additional, higher level semantic attributes per command, like the file system path that the I/O command's $offset$ maps to, the name and ID of the computer process that initiates the I/O command, the ID of the host initiating the command, and the name space ID of the disk associated with the $offset$. These quantities can be used for feature generation, labeling, pre-processing, and data filtering.

To summarize, we use Table 6 and Table 7 of Appendix A.1 to list the SW stack source and the types of the different per-command attributes in our data.

2.2 Existing Storage I/O Datasets

Almost all ransomware detection research based on storage I/O uses data from different combinations of the attributes mentioned in Section 2.1.[3] For example, the authors of [7] collect data about the number of bytes read or written and the distributions of type and path of accessed files. Derived attributes like the overwrite volume are calculated in works such as [4, 8, 9, 10, 5] (with [4] also reporting the use of file type data, directory information, and written bytes entropy). More recently, in a series of papers by [11], [12], and [13], the authors collect I/O attributes as well as memory access attributes and entropy data.

In Table 1 we present the existing datasets from the last decade, including those mentioned above. As the table shows, we found that out of 15 works that use I/O attributes for ransomware detection, only 3 make their data publicly available ([12, 5, 13]). Unfortunately, none of them label the data at the command level, making it impossible to develop high-granularity models. Inspecting these 3 sets, we see that they contain the four fundamental attributes described in Section 2.1; the RanSAP dataset also contains the write-bytes entropy, and the more recent RanSMAP dataset adds to that memory access patterns (rather than just storage). None of them contain derived attributes of the OV and Δt type mentioned in Section 2.1, which are computationally expensive to generate (see Section 4).

Calculating the number of commands and the total Byte traffic associated with the I/O streams, we find that all 3 open datasets *combined* have $\sim 5 \times 10^9$ commands and a traffic of $\sim$ 27 Tebibytes. These are modest sizes, and in the next section, we show that our dataset has nearly $\times 5$ more commands and a traffic that is nearly $\times 40$ larger. Finally, the ransomware coverage of all the 3 datasets together is also modest (not greater than 20 samples), while the coverage of our set is nearly $\times 7$ more comprehensive.

3 The CLEAR dataset

Our dataset CLEAR – the Command LEvel Annotated Ransomware – is annotated as benign or ransomware at the command level. CLEAR contains 6120 workloads of ransomware activity mixed with up to 3 different benign applications running concurrently, and chosen out of a set of 17 benign user applications like git, 7z, pip, secure deletion, and others that are very much adversarial to ransomware classification (like the encrypting application AESCrypt.exe). The NVMe streams

[3] For a contemporary review we refer to [6], and to a more general review on Machine-Learning for ransomware detection, using features at various levels of the SW stack we refer to [2].

Table 1: A comparison of the CLEAR dataset with existing datasets (published and unpublished).

Dataset (year)	Publicly available	Variants	Total traffic [TiB]	OS	Command labels	Mixed workloads
CLEAR (ours, 2025)	✓	**137**	**1045**	**Win**	✓	✓
RanSMAP [13] (2025)		6	8	Win	✗	✓
DeftPunk [5] (2024)		13	11	Win, Linux	✗	✓
RanSAP [11, 12] (2022)		7	8	Win	✗	✗
Reategui et al. [6] (2024)	✗	16	NA	Win, Linux	✗	✗
Zhu et al.[14] (2024)		10		Linux	✗	✓
Higuchi et al. [15] (2023)		5		Linux	✗	✗
Ma et al. [16] (2022)		32		Win, Linux	✗	✗
MimosaFTL [10] (2019)		14		Win	✗	✗
SSD-insider [9, 17] (2018)		8		Win	✗	✓
Amoeba [18], 2018		1		Linux	✗	✗
Paik et al. [8], 2018		1		Win	✗	✗
Redemption [4], 2017		29		Win	✗	✗
Shukla et al. [7], 2016		3		Win	✗	✗

also unavoidably involve other OS processes that are benign – like disk indexing and Windows updates. The parameters of all these benign applications also vary in the dataset (for example, the git application cloned different repositories, the archiving/encryption applications processed different directories, and we have turned on or off the disk indexing utility). The ransomware activity in these workloads was generated by choosing and detonating a single strand per workload out of 137 ransomware variants that represent 49 ransomware families. An additional variability was introduced by initiating the ransomware processes after 4 different time delays of 0, 25, 50, and 75 seconds post the workload start.

All data was collected on emulated SSDs. The SSD emulations attacked by the ransomware contained different types of victim data comprising up to 44 different file types and chosen from an in-house curation of victim files and from the large-scale NapierOne mixed file benchmark – especially designed for ransomware research [19]. The SSDs were of disk sizes 100GB and 512GB with occupancy disk levels in the range 10% – 97%. In total, we had 10 compute environments, all using Windows 10 Home Edition 22h, with 16GB RAM, and an NTFS file system, emulated with and without a disk indexing process, as it was seen to significantly change the NVMe stream. CLEAR also contains pure benign workloads: 6079 of these are in-house generated using the same infrastructure mentioned in the previous paragraph, only without detonating a ransomware payload. Additionally, CLEAR has 686 workloads downloaded from 3 subsets of the data curated by the Storage Networking Industry Association (SNIA) [20, 21, 22].

All the data in CLEAR contains the four fundamental I/O storage attributes described at the beginning of Section 2.1, and all in-house data also contains the path, process name, and process ID associated with each command. In total, CLEAR contains over 23×10^9 I/O commands in 12,885 workloads, which reflect the processing of more than 1,045 Tebibytes of data traffic. This is a significantly larger

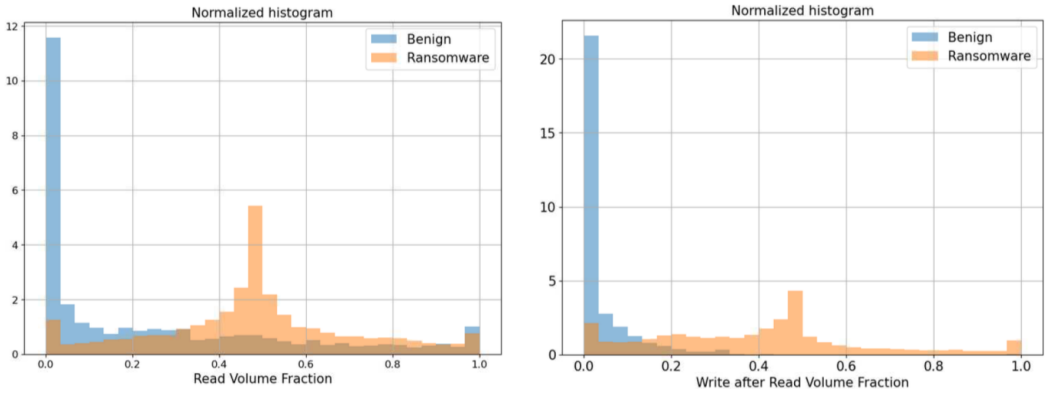


Figure 1: Normalized histograms for the frac_R and frac_{WAR} across 5,809 benign chunks and 21,243 ransomware chunks. All chunks have the same traffic volume equal to 0.5GiB.

dataset than any other in the literature (open or closed), both in size and in malware coverage. For a more detailed description of our data and its variability, see Appendix A.2.

To briefly visualize the data, we aggregate subsets of commands into chunks of traffic volume 0.5 GiB each. We do so for a subset of the data containing 185 workloads with ransomware. Per such chunk we calculate the fraction of traffic volume generated by read commands, frac_R, and the fraction of traffic volume contained in Write-after-Read byte overlaps, frac_{WAR}. We plot the results in Figure 1 for benign chunks (containing no ransomware commands) and for ransomware chunks (those with at least one ransomware command). From the histograms, it is easy to see that these fractional features contain very useful information that can be used to classify the chunks; especially, we see that ransomware activity, which encrypts and writes the same volume it reads, has $\text{frac}_R \simeq 1/2$. We also see that much of this data is overwritten so that $\text{frac}_{WAR} \simeq 1/2$.

4 Collection Method

Our data collection setup is based on emulating the ransomware attacks on a virtual machine (VM). The VM was set up using QEMU version 4.2.1 system emulator running on a host machine with Ubuntu 20.04 and CPUs Intel(R) Core(TM) i9-10900F and i9-11900K. The guest OS is a Windows 10h22 Home edition. The CPUs emulated were either the QEMU default or the host CPUs. The emulation software running on the host was Python-based and initiated the ransomware externally, allowing us to know the identity of the parent ransomware process on the VM. To avoid situations where the ransomware fails to connect to its command and control center and stops operating, the VM was connected to the internet. Ransomware samples were obtained from MalwareBazaar [23].

To capture the I/O stream logs, we use the Xperf component of the Microsoft Windows Performance Toolkit, providing us with an NVMe command log for each emulated workload, and use it to generate the basic I/O attributes. The per-command log also contains two semantic attributes – the command's file path and the name and ID of the process initiating the command. An additional useful Xperf output is the process tree of all processes in the log. This semantic data was critical in our ability to label the data at the command level (see below). We validated the robustness of the Xperf log by comparing it to the low-level output of the QEMU SSD tracer and found that the fundamental I/O attributes were identical for an absolute majority of commands. In certain rare situations, however, we observed that Xperf integrated sets of NVMe commands of the same type (e.g., a sequential set of read commands) into large artificial commands whose volume is the associated sum of the volumes in the set. We identified all such sets as having a command with $size > 2MiB$. This allowed us to deconstruct the artificial integrated commands into individual commands of $size \leq 2MiB$ by aligning them linearly along the $offset$ and $timestamp$ axes.

Researching Xperf logs, we identified two patterns that we used to develop CLEAR's per-command labeling. First, we observed that most data read by the ransomware parent process (or any of its descendant sub-processes) is written to disk post-encryption by one of the following processes:

(1) the ransomware parent process itself or (2) any of that process's descendants or (3) the main Windows OS 'System' process. The second observation involves the way file system paths change upon encryption. In many cases, all post-encrypted files are the same as pre-encrypted, except for a ransomware-generated extension. Other changes include lower-casing, and some paths not changing at all. These observations allowed us to label all commands in a stream using the following algorithm applied per workload.

1. Label all commands whose process is equal to, or that descends from, the parent ransomware process as **ransomware**.
2. Calculate the set of pre-to-post encryption path changes, and normalize all post-encryption paths to their pre-encrypted form.
3. Scan through all unique normalized paths and mark all those accessed in step 1 as 'black-listed'. Then add all commands whose normalized path is blacklisted into a 'potentially ransomware' list.
4. Mark all commands in the 'potentially ransomware' list which were initiated with the 'System' process as **ransomware**.[4]

To make sure we do not contaminate the dataset with workloads where ransomware failed to operate, we discard all workloads whose total traffic volume V encrypted by ransomware obeys $V < 0.5GB$. We also monitor a set of Honeypot files and discard workloads for which these honeypots were not encrypted.

Finally, we calculate the 8 derived OV and Δt quantities mentioned in Section 2.1 by finding, per command c, the latest of all read or write commands that are in the history of c and that have byte overlap with those of c. Since each log may have up to $O(10^7)$ commands, this is by far the most expensive processing of the raw logs. We implement the calculation of these overlaps with two dynamically updated causal databases that rely on cuckoo hash tables, and keep a record of all the read and write intervals in the workload. Because these auxiliary command attributes generation can be computationally expensive we have already precomputed them for the entire dataset and provide them as part of the published dataset for the benefit of the community.

5 Experiments

To demonstrate the usefulness of our dataset, we train and evaluate several models on it. Full implementation details of the models can be found in Appendix B.

5.1 Aggregative Models

The two SotA ransomware detection algorithms we compare to are aggregative tabular models - extracting dozens of statistical features from large chunks (e.g. tens of thousands) of I/O commands, and using those to classify the entire chunk. Due to the lack of published code, we developed a Random Forest (RF) model to represent a typical tabular approach in the literature, inspired by [9]. The second model is DeftPunk [5] - a Decision Tree combined with XGBoost [24]. We also train two more baseline models - a convolutional UNET architecture and a Patch Level Transformer (PLT), representing our best effort with aggregative models. Both partition each chunk into 100 consecutive patches, generate 181 statistical features from each patch, and concatenate them to form the input sequence. The PLT, especially, is already a powerful sequential model that outperforms all other aggregative models. However, it still falls short of our command-based models described in the next section.

5.2 Command Based Models

To fully utilize our dataset's per-command labeling, we evaluate two classes of sequential models, both of which work directly on the command stream without aggregation and predict a label for each command. The first is Command Level Transformer (CLT) architecture, with 3 self-attention encoder

[4]Excluding from these the System commands that access the blacklisted files *after* any commands by processes known to be benign (e.g., processes in Table 10 initiated by the emulation). This exclusion is done since we know a benign process accesses data only if the malicious System process has already 'released' it.

layers, each following the architecture presented in [25], and a final classification layer comprised of a fully connected projection layer followed by a sigmoid function. The output is a vector of label predictions - one for each input, and the training loss is binary cross-entropy, applied to each of the predictions. The second model class is an LSTM architecture, mirroring the Transformer but replacing each self-attention layer with an LSTM layer. We trained two variants, one biLSTM following the architecture presented in [26], with a first bidirectional layer, and one uLSTM where all layers are unidirectional. Hidden dimensions of all models are 128, and all models were trained with an input length of 1000 commands. The CLT and biLSTM were also evaluated on the same input length, while the uLSTM was run continuously on the input stream without resetting its internal state.

5.3 Command Tokenization

Each I/O command contains 7 numerical features of different types, altogether containing up to 40 digits per command. Common tokenization methods (e.g., those presented in [27]) would thus result in dozens of tokens per command and prohibitively long sequences. Instead, we first compress the commands by quantizing each feature into a few bits, and then tokenize the compressed representation by concatenating those bits into integer tokens. The quantization process uses domain knowledge[5] to preserve as much information as possible, and is described in detail in Appendix C. Table 2 specifies the quantization method and the bit output for each feature. To avoid a large vocabulary size, we split the resulting 18-bit tokens into two 9-bit tokens and trade effective context length, which is reduced by a factor of 2, for a smaller vocabulary size. To make sure the learning algorithm treats the two tokens differently we add a 10^{th} "index" bit to each token in the pair – $index_1 = 0$ for the first and $index_2 = 1$ for the second – thus fully separating the tokens in the embedding space, at the cost of increasing the vocabulary size to 1024.

Table 2: Quantization and tokenization of an I/O command. The bits are concatenated into a pair of 9-bit tokens, and an additional 10^{th} bit is added to each token as its index in the pair.

Feature	Operation	Bits	Feature	Operation	Bits
δt	$\log N$	4	$offset$	take MSB	4
$size$	$\log N$	4	$offset$	take LSB	2
$opcode$	None	1	$Auxilliary$	binarize	3
$index_1$	always 0	1	$index_2$	always 1	1

5.4 Results

The common metrics used in the literature such as Precision, Recall, and F1 score (for example see [13]), are all heavily dependent on the aggregation methods used to chunk the I/O stream, and thus difficult to compare between different models, using different aggregation methods (or none at all). We instead use metrics that are agnostic to aggregation and directly measure the rate of mislabeled I/O traffic. Since our data is labeled for each command, we can accurately measure these metrics. We opt to use the following metrics:

1. **Missed Detection Rate (MDR)**: Measures the percent of missed ransomware traffic out of the total ransomware traffic volume, calibrated for a specified False Alarm Rate (FAR) workpoint. We use the FAR work point of 1 false alarm per 50 GiB of data traffic.
2. **MegaBytes to Detection (MBD)**: The volume of traffic corrupted by the ransomware before detection, as a measurement of the severity of the damage caused by the ransomware. In particular, we measure the cumulative distribution function (CDF) of the MBD across all ransomware streams and quote its third upper quantile as $\mathbf{MBD_3}$. This is done under the same work point as the MDR.
3. **Area Under the Curve (AUC)**: Measuring the tradeoff between FAR and MDR, and is agnostic to any specific choice of workpoint.

[5] e.g., typical command *size* and δt distributions, typical file sizes and OS behavior for $offset$ bits, etc.

We optimized all models and Table 3 summarizes the best results for each model. The details of the hyper-parameter optimization appear in Appendix D. As expected, the more powerful aggregative models, especially the sequential PLT, outperform the SotA tabular models on aggregated data and represent our best effort for a fair comparison with the command-based models. However, we see that all command-based models outperform the aggregated models (or for *MDR* - the differences are within the STD range), despite being much smaller in size. Among the command-based models, both the biLSTM and the Transformer show similar results, with the clear winner being the continuous uLSTM, which retains infinite history. This suggests that once we move to a command-based approach, the main difference is not the specific architecture but the context it uses to classify a command. In the next sections, we show that such models do not simply revert to being aggregative or only use the predicted command's features, but rather rely on context for their predictions.

Table 3: Measurements of MDR, MBD_3, and AUC for our models, with standard deviation (STD).

Model	MDR	MBD_3	AUC	Command Level	Parameters
RF	9.07±1.8	286±33	0.995843±0.000502	✗	10K
DeftPunk	6.74±1.2	282±34	0.990172±0.000759	✗	260K
UNET	1.34±0.4	195±41	0.998994±0.000093	✗	153M
PLT	1.16±0.2	157±18	0.998845±0.000116	✗	17M
CLT	1.42±0.4	77±10	0.999276±0.000112	✓	430K
biLSTM	1.24±0.3	76±07	0.999322±0.000106	✓	400K
uLSTM	**0.36±0.1**	**50±03**	**0.999664 ± 0.000065**	✓	530K

5.5 The Importance of Context

We trained a series of biLSTMs with increasing context lengths and plotted the results in Figure 2. We see that performance steadily improves with more context. Since they predict labels per command, this means that the more context each command "sees", the better the model's prediction on it becomes, suggesting a rich sequential structure of the command stream that the command-based models can exploit, rather than simply predicting based on the command's features alone. Furthermore, if we consider the continuous uLSTM model as having "infinite history", we can see that it performs significantly better than models with a limited context window.

5.6 Command Accuracy

Next, we want to show that command-based models do indeed learn low-level patterns in the data, and not simply revert to some global aggregative function applied equally to all of their outputs. To demonstrate this, we slice the data into 1000 command slices and calculate the prediction accuracy per slice.

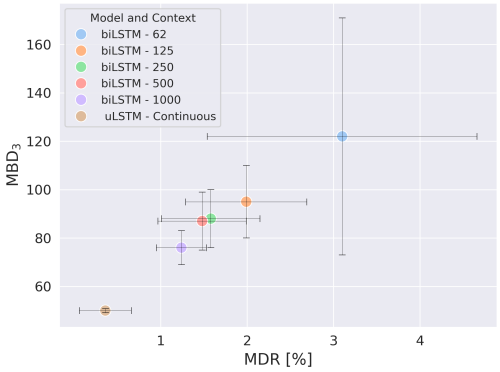


Figure 2: The dependence of MBD_3 and MDR on context length for command-based models.

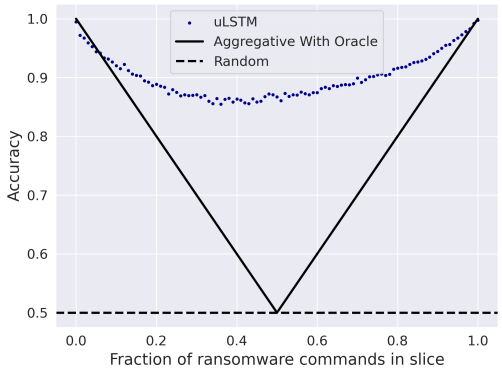


Figure 3: Mean accuracy of the uLSTM on data slices (1000 commands) plotted against the fraction of ransomware commands in each slice.

We group the slices by the fraction of ransomware commands they contain, measure the model's average accuracy on each such group, and plot the results in Figure 3. The solid line represents the

best an aggregative (single output) model can do with an oracle – always predicting the majority in the slice. We see that the model's accuracy is $\geq 85\%$, even on the most challenging slices whose ransomware content is $\sim 50\%$. The model can still separate the mix into ransomware and benign commands with significantly better than the best aggregate accuracy, showing command-level pattern recognition capabilities.

This demonstrates that the data is rich enough to support individual command predictions, as well as the usefulness of our per-command labeling, as it opens possibilities for learning tasks at the individual command level. Such tasks can be, for example, recovering only the data corrupted by the ransomware, or simply faster detection, as it is no longer necessary to aggregate thousands of commands before a prediction can be made.

5.7 Feature Ablation

To measure the contribution of our features and tokenization to the overall performance of our models, we measure it for ablated versions of our models' token space. We do so by training a new set of CLT models while dropping sub-spaces of the embedding space as per their description in Section 5.3. The results of this study are presented in Table 4. In particular, we see that the auxiliary features we provide with our dataset positively contribute to the overall performance of the model.

6 Unseen Ransomware Benchmark

Due to the large number of ransomware variants in the wild and new variants that regularly emerge, an important consideration for any detection model is its ability to generalize to ransomware types it has not seen during training. To evaluate this, we created an 'unseen' ransomware benchmark.

Since differently named ransomware may, in actuality, be similar in their behavior[6], we aim to ensure that we do not have data leaks due to similar ransomware misclassified as unseen. To achieve this, we first perform a clustering analysis of the 137 ransomware variants we have in CLEAR. As a distance metric, we use TLSH [28] with a similarity score of 100 [29] (± 40 scores produced the same clusters). The analysis shows that the number of independent ransomware families in CLEAR is 47, with the full list of clusters in Appendix A.3. We now divide the ransomware variants into 3 roughly equal groups and perform 3 test runs. For each run, we mix two groups and evenly split them into train and in-distribution test sets. The third group is the out-of-distribution test set. The benign streams used for the train and test data remain the same throughout the splits, to provide the same negative samples to the different id and ood tests. Finally, we average the results of all three runs to get an evaluation of the difference in performance between the in- and out-of-distribution sets. The running code, including the group splits, is available in our code repo. Here we provide an example run for several of our models, in Figure 4. We can see that all models degrade to some degree over unseen ransomware, with tabular models degrading more than the sequential models.

Table 4: Feature ablation for the command-based models. Each feature subset was ablated together. The numbers represent the ratio by which the corresponding metric increased.

Feature Subset	Performance Degradation		
	MDR	MBD_3	$1 - AUC$
offset	x2.1	x1.6	x2.1
δt	x1.7	x1.3	x1.5
opcode	x1.4	x1.2	x1.4
size	x1.3	x1.1	x1.7
Auxilliary	x1.3	x1.1	x1.3
index	x1.1	x1.1	x1.0

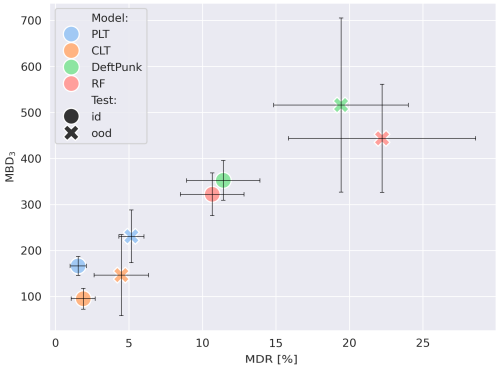


Figure 4: In and out of distribution MBD_3 and *MDR* results. Error bars represent both run variability and individual run uncertainty.

[6]Due to sample mislabeling, code or algorithm copying between ransomware groups, etc.

7 Limitations

The first limitation is the absence of traffic-based features, such as traffic data entropy. These can further enrich our data, and we are currently working on expanding the CLEAR dataset to include recordings with such features as well. A second limitation is the lack of data generated under the Linux OS. While most ransomware attacks target Windows systems, There are also Linux-based attacks, and we intend to expand the CLEAR dataset to include Linux data in the future. The third limitation is that our data was collected under a controlled environment, and not "in the wild". However, due to the damaging effects of ransomware attacks, collecting such data from real systems is infeasible.

8 Conclusion

We introduce the CLEAR dataset, the largest publicly available collection of ransomware activity traces, containing over 100 times more traffic data and over 10 times more ransomware variants. This is also the only publicly available dataset with per-command labeling, allowing for the identification of low-level command-based ransomware patterns lost under aggregation, and thus enabling new possible tasks. We supplement the data with generated auxiliary features and, through feature ablation, show them to improve model performance.

We train several baseline command-based algorithms and demonstrate that they outperform the SotA tabular models as well as "best effort" aggregative models. We further show that their strength comes from recognizing command-level patterns in the data, and not simply from better aggregation or individual command features. Finally, we provide the code for the new models and the pipeline, as well as a benchmark for evaluating a model's robustness to previously unseen ransomware variants.

References

[1] Harun Oz, Ahmet Aris, Albert Levi, and A. Selcuk Uluagac. A survey on ransomware: Evolution, taxonomy, and defense solutions. *ACM Comput. Surv.*, 54(11s), September 2022. ISSN 0360-0300. doi: 10.1145/3514229. URL https://doi.org/10.1145/3514229.

[2] Jamil Ispahany, Md. Rafiqul Islam, Md. Zahidul Islam, and M. Arif Khan. Ransomware detection using machine learning: A review, research limitations and future directions. *IEEE Access*, 12:68785–68813, 2024. doi: 10.1109/ACCESS.2024.3397921.

[3] Junhong Yin and Kyungtae Kang. Defense and recovery strategies for flash-based storage under ransomware attacks: A survey. In *2025 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4, 2025. doi: 10.1109/ICEIC64972.2025.10879685.

[4] Amin Kharraz and Engin Kirda. Redemption: Real-time protection against ransomware at end-hosts. pages 98–119, 10 2017. ISBN 978-3-319-66331-9. doi: 10.1007/978-3-319-66332-6_5.

[5] Zhongyu Wang, Yaheng Song, Erci Xu, Haonan Wu, Guangxun Tong, Shizhuo Sun, Haoran Li, Jincheng Liu, Lijun Ding, Rong Liu, Jiaji Zhu, and Jiesheng Wu. Ransom access memories: Achieving practical ransomware protection in cloud with DeftPunk. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 687–702, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/wang-zhongyu.

[6] Nicolas Reategui, Roman Pletka, and Dionysios Diamantopoulos. On the generalizability of machine learning-based ransomware detection in block storage, 2024. URL https://arxiv.org/abs/2412.21084.

[7] Manish Shukla, Sutapa Mondal, and Sachin Lodha. Poster: Locally virtualized environment for mitigating ransomware threat. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1784–1786, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2989051. URL https://doi.org/10.1145/2976749.2989051.

[8] Joon-Young Paik, Joong-Hyun Choi, Rize Jin, Jianming Wang, and Eun-Sun Cho. A storage-level detection mechanism against crypto-ransomware. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2258–2260, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3278491. URL https://doi.org/10.1145/3243734.3278491.

[9] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun 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)*, pages 875–884, 2018. doi: 10.1109/ICDCS.2018.00089.

[10] Peiying Wang, Shijie Jia, Bo Chen, Luning Xia, and Peng Liu. Mimosaftl: Adding secure and practical ransomware defense strategy to flash translation layer. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 327–338, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360999. doi: 10.1145/3292006.3300041. URL https://doi.org/10.1145/3292006.3300041.

[11] Manabu Hirano and Ryotaro Kobayashi. Machine learning-based ransomware detection using low-level memory access patterns obtained from live-forensic hypervisor. In *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 323–330, 2022. doi: 10.1109/CSR54599.2022.9850340.

[12] Manabu Hirano, Ryo Hodota, and Ryotaro Kobayashi. Ransap: An open dataset of ransomware storage access patterns for training machine learning models. *Forensic Science International: Digital Investigation*, 40:301314, 2022. ISSN 2666-2817. doi: https://doi.org/10.1016/j.fsidi.2021.301314. URL https://www.sciencedirect.com/science/article/pii/S2666281721002390.

[13] Manabu Hirano and Ryotaro Kobayashi. Ransmap: Open dataset of ransomware storage and memory access patterns for creating deep learning based ransomware detectors. *Comput. Secur.*, 150(C), March 2025. ISSN 0167-4048. doi: 10.1016/j.cose.2024.104202. URL https://doi.org/10.1016/j.cose.2024.104202.

[14] Weidong Zhu, Grant Hernandez, Tian Dave (Jing) Garcia, Washington, Sara Rampazzi, and Kevin Butler. Minding the semantic gap for effective storage-based ransomware defense. In *International Conference on Massive Storage Systems and Technology*, 2024.

[15] Kosuke Higuchi and Ryotaro Kobayashi. Real-time defense system using ebpf for machine learning-based ransomware detection method. In *2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*, pages 213–219, 2023. doi: 10.1109/CANDARW60564.2023.00043.

[16] Boyang Ma, Yilin Yang, Jinku Li, Fengwei Zhang, Wenbo Shen, Yajin Zhou, and Jianfeng Ma. Travelling the hypervisor and ssd: A tag-based approach against crypto ransomware with fine-grained data recovery. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 341–355, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. doi: 10.1145/3576915.3616665. URL https://doi.org/10.1145/3576915.3616665.

[17] Sungha Baek, Youngdon Jung, David Mohaisen, Sungjin Lee, and DaeHun Nyang. Ssd-assisted ransomware detection and data recovery techniques. *IEEE Transactions on Computers*, 70(10): 1762–1776, 2021. doi: 10.1109/TC.2020.3011214.

[18] Donghyun Min, Donggyu Park, Jinwoo Ahn, Ryan Walker, Junghee Lee, Sungyong Park, and Youngjae Kim. Amoeba: An autonomous backup and recovery ssd for ransomware attack defense. *IEEE Computer Architecture Letters*, 17(2):245–248, 2018. doi: 10.1109/LCA.2018.2883431.

[19] Simon R. Davies, Richard Macfarlane, and William J. Buchanan. Napierone: A modern mixed file data set alternative to govdocs1. *Forensic Science International: Digital Investigation*, 40:301330, 2022. ISSN 2666-2817. doi: https://doi.org/10.1016/j.fsidi.2021.301330. URL https://www.sciencedirect.com/science/article/pii/S2666281721002560.

[20] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Systor '17 traces (SNIA IOTTA trace set 4928). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2016. URL http://iotta.snia.org/traces/block-io?only=4928.

[21] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007. URL http://iotta.snia.org/traces/block-io?only=388.

[22] Vishal Sharda, Swaroop Kavalanekar, and Bruce Worthington. Microsoft production server traces (SNIA IOTTA trace set 158). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2008. URL http://iotta.snia.org/traces/block-io?only=158.

[23] abuse.ch. MalwareBazaar: A Project to Share Malware Samples with the Security Community. https://bazaar.abuse.ch/, 2025. Accessed: 2025-05-16.

[24] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[26] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[27] Aaditya K Singh and DJ Strouse. Tokenization counts: the impact of tokenization on arithmetic in frontier llms. *arXiv preprint arXiv:2402.14903*, 2024.

[28] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh – a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, 2013. doi: 10.1109/CTC.2013.9.

[29] Haoping Liu, Josiah Hagen, Muqeet Ali, and Jonathan Oliver. An evaluation of malware triage similarity hashes. In *ICEIS (1)*, pages 431–435, 2023.

[30] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020. URL https://arxiv.org/abs/2010.11929.

[31] Esmeralda A. Ramalho, Joaquim J.S. Ramalho, and José M.R. Murteira. Alternative estimating and testing empirical strategies for fractional regression models. *Journal of Economic Surveys*, 25(1):19–68, 2011. doi: https://doi.org/10.1111/j.1467-6419.2009.00602.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-6419.2009.00602.x.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: All claims in the abstract and introduction are substantiated in the paper. In particular sections: Section 2.2, Appendix A.2, Section 5, Section 6

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 7 Clearly deliniates several limitaitons our work has.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: Our paper does not contain any theoretical claims

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We provide full implementation details of our algorithms, full description of our data and collection methods (Appendix B, Section 6, Appendix A.2, Appendix A, Appendix B, Appendix C), and all the code used for processing the data, training and evaluating the models, and producing the result metrics. We include full train-test split specifications in the code for all train and test work we performed, including the benchmark we provide. We, of course, also provide all the data the experiments in this paper were performed on. When relying on external sources (such as sources for the ransomware variants we used to generate data), we clearly specify them.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility.

In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide the full dataset we worked with (linked in the paper to an anonymized repository on Kaggle). We also provide all the code used in the experiments in this paper, including processing the data we provide and generating the result metrics (linked in the paper to an anonymized repository on GitHub). The data collection system is not our code (e.g., relying on external tools such as Microsoft's Xperf toolkit). In this case, we provide full details on what we used and how, allowing anyone else using these tools to produce the same data (up to the inherent randomness of the systems themselves, such as the ransomware variants). The code we provide includes a README guide to run the code from scratch, and it requires no external dependencies beyond publicly available Python packages. Section 1

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide full implementation details in the appendices, and publish the full code used in the experiments in this paper, including train-test splits, etc. (Appendix A, Appendix B, Appendix C)

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: We provide error bars where applicable and suitably describe them (e.g., as standard deviation, etc.) Section 5

Guidelines:

- The answer NA means that the paper does not include experiments.
- The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide details on the hardware our models were trained on Appendix B

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Our experiments and data did not conflict with the NeurIPS code of ethics

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Our work aims to improve ransomware detection and thus reduce the damage to society from this malicious activity. We discuss this briefly in the introduction Section 1

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We do not release any of our ransomware samples, and the data and code are harmless

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

Justification: Our data is provided under the CC BY-NC-SA 4.0 license. Usage of external data (SNIA data) is properly attributed Appendix A

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: We provide a thorough documentation of our dataset with it, and provide full README of our code with our code

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: We did not use human subjects for this paper

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: We did not use human subjects for this paper

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [NA]

Justification: We did not use LLMs for this paper

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.

Table 5: A pictorial representation of the overlaps. Here we denote a read command by an empty circle, ○, and a write command by the full circle, ● . We denote the *timestamp* by t, and note that the horizontal axis in the NVMe stream column represents the $offset$ attribute.

NVMe Stream	Attributes	Derived Properties
t=2: ○○○○	t=2, $offset$=0, $size$=4, $opcode$=R	-
t=3: ●●●●●	t=3, $offset$=1, $size$=5, $opcode$=W	WAR: OV=3, Δt=1
t=5: ○○○	t=5, $offset$=2, $size$=3, $opcode$=R	RAR: OV=2, Δt=3, RAW: OV=3, Δt=2

Table 6: A description of the SW stack levels used to generate the different data attributes, demonstrating the way we generate the derived properties and the labels.

SW stack	Attributes	$\longrightarrow$	Derived Properties	$\Longrightarrow$	Label (see Section 4)
High Level	process, PID, path, ...	$\longrightarrow$	file name, parent PID, ...	$\Longrightarrow$	Ransomware or Benign
Low Level	$opcode$, $offset$, $size$, ...	$\longrightarrow$	OV_{WAR}, Δt_{WAR}, ...		

A Data Specifications

A.1 Details of the I/O Data Structure

We use this section to add more details to the discussion in Section 2.1. Especially, in Table 5 we pictorially show how we calculate the derived overlap features OV and Δt, Table 6 describes the data generation flow (from raw data obtained in various levels of the SW stack to the derived per-command attributes and the per command label), and finally in Table 7 we list the per-command data attributes contained in our data.

A.2 Specification of the CLEAR data set

CLEAR was collected with a data collection, verification, and labeling system, that we developed in-house. Additional information on the data set is brought in Table 8. In particular, we ran 137 ransomware variants list in Table 9. We also ran 17 types of benign software that are presented in Table 10. These ran on virtualization of stand-alone PCs with either a 100GB SSD whose disk usage

Table 7: A list of the per command attributes, their types, source, and ranges. We note in passing that the range noted in the table for the *size* attribute is for the data recorded internally; for the SNIA data that range goes up to 32MiB.

Attribute	Type	Source	Example	Range
offset	ordinal	NVMe	25245132	[0, disk size]
size	integer	NVMe	32	[1, 2MiB]
opcode	categorical	NVMe	R/W	
timestamp	float	NVMe	76.214 sec	-
all four OV's	integer	derived from NVMe	32	-
all four Δt's	float	derived from NVMe	0.15 sec	-
Process Name	string	process manager	TiWorker.exe	-
PID	integer	process manager	4963	-
Path	string	file system	C:\Desktop\001.jpg	-

Table 8: CLEAR Data set Overview

Label	Number of Recordings	Volume [TiB]	Days	No. of Commands
Ransomware	6,120	709.3	71	16,272,162,036
Benign	6,765	335.7	114	6,860,594,885
Total	12,885	1,045	185	23,132,756,921

Table 9: Ransomware families in our data set

Ransomware Families	No. of Variant Streams per Family
Sodinokibi	14
LockBit	9
BlackMatter	8
Hive, Thanos	7
AvosLocker	6
Maze, BlackBasta, TimeTime, Babuk	5
Mespinoza, RagnarLocker, GlobeImposter	4
Play, Karma, Lorenz, Diavol	3
Sugar, WannaCry, MedusaLocker, Royal, ViceSociety, Stop, Cuba, Rook, Conti, BianLian	2
Neshta, CryLock, Zeppelin, Ransomware.Makop, Alkhal, Clop, LIKEAHORSE, Teslarvng, ATOMSILO, Ransomware.Koxic, Phobos, RansomEXX, RanzyLocker, Nefilim, MRAC, Intercobros, HelloXD, DECAF, MountLocker, BlackOut, Cerber, DarkSide	1

was up to 95%, or a 512GB SSD disk with a disk usage of 49%-60%. The OS was a Windows 10h22 Home edition, with two CPUs: Intel(R) Core(TM) i9-10900F and i9-11900K, and with 16GB RAM. In addition, we placed different types of victim files on the virtual disk, including the Napier-Small repository (157GB) and the Napier-tiny repository (17.8GB) [19], some from an in-house generic user files repository, and 57GB stored in $\sim 65K$ png files from the DiffusionBM-2M data set repository. Finally, because we saw that disk indexing can be a CPU-heavy process and can significantly change the NVMe sequence, the data was generated in two configurations: where the indexing processes are either turned on or off. In total, we collected data from 10 such configurations. Each of our traces reflects up to approximately 1000 seconds of operation time, with additional variability introduced via varying the launch time of ransomware. Specific information regarding the data acquired from SNIA can be found on our published data set[7]. The description of the images on which we ran the workloads is brought in Table 11.

[7] `https://www.kaggle.com/datasets/johndoenvme/clear-command-level-annotated-ransomware`

Table 10: Benign SW workload types. Additional variability was introduced by using more than a single parameter for certain SWs. For example, the files read, written, deleted, archived, and encrypted were varied.

Workload Type	Specific Application
Archiving	7z, winRAR
Encryption	AESCrypt
Deletion	SDelete, fsutil
Other disk accesses	download and install apps from the internet
	git clone
	read and write files from disk
	conda install, pip install
	Windows update
	Compilation
	Document editing
	Web surfing
	Application downloading and installing
	Disk populating

Table 11: Disk images

SSD Volume [GB]	Victim File Sources	Disk Occupancy
100	Napier, 65K png files from the DiffusionBM-2M data set repository, a user file collection	10% - 95%
512		50% - 60%

A.3 Details for Clustering and the Unseen Ransomware Benchmark

For each experiment of the leave-one-out cross-validation, the dataset was partitioned into three folds. Each fold represents the data used for an out-of-distribution test set and is presented in Table 12. Each ransomware variant is named in the format `FAMILY_ABCD`, stating the ransomware family name and the first four characters of the SHA-256 hash commonly used as the ransomware signifier.

The TLSH distance chosen for the clustering process was 100 (every distance between 50 to 100 yielded the same clusters). In rows where more than one cluster (of the same size) appears, the different clusters are distinguished and separated from one another with round brackets.

Table 12: Ransomware variant divided into clusters per fold.

Ransomware Variant Clusters	Cluster Size	Number of Clusters	Fold
AvosLocker_43b7, Ransomare.Koxic_7a5e, Mespinoza_44f1, AvosLocker_fb54, Mespinoza_7c77, Mespinoza_f602, Mespinoza_0433, Neshta_9317, Cerber_078d, AvosLocker_f810, AvosLocker_6cc5, AvosLocker_c0a4, AvosLocker_84d9, Conti_24ac, RansomEXX_fa28, BlackBasta_2558, Lorenz_1264, Lorenz_a0cc, Lorenz_edc2, Teslarvng_bb91	20	1	1
TimeTime_5ee8, TimeTime_972e, TimeTime_b599, TimeTime_b722, Royal_44f5, Royal_d9be,	6	1	
Babuk_eb18, Babuk_ca0d, Babuk_575c, Babuk_a522, Babuk_77c7	5	1	
Karma_4dec, Karma_3462, Karma_84d2	3	1	
Cuba_21ac, BlackBasta_723d	2	1	
ViceSociety_HelloKitty_fa72, Alkhal_7a31, ATOMSILO_d9f7, Stop_0d50, Sugar_09ad, Maze_4263, MRAC_768c, Phobos_265d, WannaCry_be22, Play_952f	1	10	
Sodinokibi_fd16, Sodinokibi_b992, Sodinokibi_0441, Sodinokibi_6834, Sodinokibi_20d4, Sodinokibi_9df3, Sodinokibi_de20, Sodinokibi_9b11, Sodinokibi_cb4a, Sodinokibi_2f00, Sodinokibi_9437, Sodinokibi_7c8c, Sodinokibi_db59, Sodinokibi_3b0c	14	1	2
BlackOut_ee13, Diavol_b3da, Diavol_7945, Diavol_2723, Play_006a, ViceSociety_HelloKitty_c249, RanzyLocker_0db6, Play_dd10	8	1	
Thanos_caf8, Thanos_8141, Thanos_66ed, Thanos_4852, Thanos_6e6b, Thanos_cbdb, Thanos_d29a	7	1	
MountLocker_00ed, MedusaLocker_f5fb, MedusaLocker_c2a0	3	1	
(BlackBasta_df5b, GlobeImposter_185f), (Rook_c2d4, LockBit_f2da)	2	2	
WannaCry_d103, Cuba_521c, Zeppelin_824a, Sugar_1d4f, DECAF_a471, Ransomare.Makop_a617, Stop_59d0, Conti_53b1, Intercobros_ade5, HelloXD_903c	1	10	
Hive_9e9f, Hive_0320, Hive_a45c, Nefilim_fb3f, Hive_460b, Hive_0302, Hive_dfa5, BianLian_eaf5, Hive_45fe, BianLian_46d3	10	1	3
(BlackMatter_8ead, BlackMatter_22d7, BlackMatter_e4fd, BlackMatter_c6e2, BlackMatter_730f, BlackMatter_b824, BlackMatter_2aad, BlackMatter_5da8), (LockBit_bdc2, LockBit_e216, LockBit_4bb1, LockBit_0545, LockBit_acad, LockBit_7340, LockBit_dd8f, LockBit_786a)	8	2	
(Maze_e8a0, Maze_3885, Maze_6a22), (GlobeImposter_e6fa, GlobeImposter_39f5, GlobeImposter_70fa)	3	2	
(RagnarLocker_10f9, RagnarLocker_5469), (BlackBasta_e281, BlackBasta_c4c8)	2	2	
RagnarLocker_afab, CryLock_4a47, RagnarLocker_041f, LIKEAHORSE_6d2e, DarkSide_f3f2, Rook_f87b, TimeTime_c535, Clop_bc1f, Maze_4e25	1	9	

B Model Specifications

B.1 Random Forest Model

The first model we use as a baseline comparison for our approach is a Random Forest (RF) with 23 tabular features extending past works on decision trees and random forest approaches like [9]. Especially, to avoid over-fitting and being constrained by the hardware implementation, our model is restricted to 20 trees with a maximum depth of 20. After training (using the Gini criterion) we validated through a feature importance analysis the common wisdom, which suggests the read and overwrite fractional volumes are the most important features (during a ransomware cyber attack, the read, write, and overwrite volumes are nearly identical, making the fractional read and overwrite volume very close to $1/2$). We find that the space on the disk the trained model takes up is roughly $18MB$, corresponding to 260K nodes in the forest.

B.1.1 Features

The featurization process is preceded by slicing the data into slices of identical read and write volume (option (iii) described in Appendix C.1) and that we choose the slice volume to be equal to $V_0 = 0.5GiB$. We extract the features we present in Table 13 from each slice. In that table, we denote by V the total number of logical blocks in the slice. The value of V obeys $V \lesssim V_0$ because there is no guarantee that the cumulative sum of *size* along the trace will be evenly commensurate with V_0. Finally, by $\sigma_{OV_{WAR}}$ and $\mu_{OV_{WAR}}$ we denote the standard deviation and the mean of OV_{WAR} across the commands of the slice. We choose 10 bins for the histograms H_R and H_{WAR} described in Table 13 and so have 23 features per slice.

Table 13: RF Features.

f_R	$\frac{1}{V}\sum_{\substack{c \\ opcode=R}} size(c)$
f_{WAR}	$\frac{1}{V}\sum_{c} OV_{WAR}(c)$
CV_{WAR}	$\sigma_{OV_{WAR}}/\mu_{OV_{WAR}}$
H_R	The histogram of read commands' *size*
H_{WAR}	The histograms of OV_{WAR} values

B.2 DeftPunk Model

The second baseline model we use is the DeftPunk model [5]. It is a two-layer model: the authors choose to apply a Decision Tree (DT) on a first set of features and pass the ransomware-suspicious samples to a second layer chosen to be XGBoost, which is applied on a larger set of features. The partition between features in the first and second layers is done by the computational effort made in feature extraction and we follow the same choice described by the authors. The models we trained have a maximum depth of 6 in both the DT and the XGBoost. We found that the model on disk takes up $0.5MB$, corresponding to its 10K nodes, and 100 trees in the XGBoost.

B.3 CLT and PLT Models

The structure of our problem requires us to identify the presence of malicious commands, often interspersed with differing amounts of benign commands in between. Any single command or small patch of commands in itself is very hard to model, and it is only when taken with other relevant commands that we can attempt to classify it. Attention mechanism, with its ability to identify relevant information at varying, and often long, distances, and ignore large amounts of irrelevant information in between, is a natural choice for this problem. Thus, we need an architecture that is especially

suited to identify relevant information at varying, and often long, distances, and ignore large amounts of irrelevant information in between. An attention mechanism is therefore a natural choice for model architecture and we indeed apply it to our problem The architecture itself follows the encoder parts of the architecture presented in [25]. In the remainder of this section, we introduce two modeling approaches: at the command level, using the Command Level Transformer (CLT), and at the level of small command patches, using the Patch Level Transformer (PLT).

We note in passing that to train both models, we used an ADAM optimizer with a learning rate of $1e^{-4}$. The CLT's optimizer used an LRScheduler of 30 steps and $\gamma = 0.8$. The CLT and PLT were trained for 300 and 400 epochs, respectively. The models were trained on a single H100 GPU.

B.3.1 The Command Level Transformer (CLT)

The CLT works on frames of 250 commands, which the tokenization process turns into 500 tokens (for tokenization details, see the next section). The transformer is made up of three self-attention encoding layers, which follow the architecture presented in [25]. On top of the transformer, we then compose a classification module, comprised of a fully connected projection layer – reducing the output dimension back to 500×1 – and a 2×1 convolution with stride 2, producing 250 outputs. These are fed through a softmax layer to predict a label (ransomware or benign) per input command. Thus, the model predicts the label of each command individually, given its context in the frame. Training loss is binary cross-entropy, applied to each of the 250 predictions. During inference, the per-command predictions are averaged over a slice, and this average is then thresholded to produce a final binary prediction for the entire slice. We describe the hyperparameters chosen for the CLT and the PLT in Table 14.

Table 14: CLT Hyperparameters.

Hyperparameter Name	Chosen Value
Vocabulary Size	1,024
Batch Size	64
Embedding Dimensions	128
Feedforward Dimensions	128
No. of Heads	4
No. of Layers	3
Context Length	1000
Convolution Kernel Size	2
Dropout	0.1

B.3.2 The Patch Level Transformer (PLT)

The PLT follows the idea of [30] and instead of working on individual commands, works on 100 small patches of the command stream – these are the tokens of the PLT described in the next section. The transformer has six transformer encoding layers. It has a final regression and classification module comprised of a linear projection to reduce the output dimension to 100×2 and a sigmoid, predicting two fractions per token representing the read and write ransomware IO volumes in a patch. The model is trained to perform per-token fractional regression [31] with a cross-entropy (CE) loss by summing the CE terms of all 100×2 fractional volumes. To get a final prediction for an entire slice, the fractions are summed per token, and the result is pooled by averaging across tokens to provide a ransomware/benign prediction probability. We threshold this probability to obtain a binary class per slice. We show the hyperparameters of the PLT in Table 15.

Table 15: PLT Hyperparameters

Hyperparameter Name	Chosen Value
Input Size	181
Batch Size	256
Embedding Dimensions	512
Feedforward Dimensions	2,048
No. of Heads	4
No. of Layers	6
Context Length	100
Dropout	0.1

B.4 biLSTM and uLSTM

Like the Transformer architecture, an LSTM is a natural choice for modeling long sequential data such as the command streams in CLEAR. We attempted two variants - a bidirectional LSTM (biLSTM) following the architecture presented in [26] with a first bidirectional layer, and a fully unidirectional (uLSTM) architecture. The first was trained on several different context lengths to measure the impact of context on performance. The second was trained on a context length of 1000, but inference was done continuously, never resetting the LSTM memory. Table 16 summarizes the hyperparameters used in all cases. The models were trained on a single H100 GPU.

Table 16: LSTM Hyperparameters

Hyperparameter Name	Chosen Value	
	biLSTM	uLSTM
Vocabulary Size	1,024	1,024
Batch Size	64	64
Embedding Dimension	128	128
Hidden Dimension	128	128
No. of Layers	3	3
Context Length	62, 125, 250, 500, 1000	1000
Dropout	0.1	0.1

B.5 UNET

To test the importance and fit of the transformer architecture, we tested other architectures like fully connected applied to a concatenation of the PLT features of all tokens, as well as 1D convolutional networks. The best models of these architectures were of the UNET type with the architecture Appendix B.5.1, input with the tokens of the PLT, hence $T(0) = T(f) = 100$ and $F(0) = 181$ and $F(f) = 2$.

B.5.1 UNET Architectures

We pictorially present the specific implementation of the architecture we use in Figure 5. By $T(i)$, $F(i)$ we denote the number of tokens and the embedding size, respectively, of the UNET level i. The blue arrows denote a max pooling with kernel size = 2, which means that $T(i+1) = T(i)//2$. The green arrow denotes transpose convolution with stride = 2 and with kernel size equal to $max(3, k)$, where k is a parameter of the encoder and decoder blocks E and D that we discuss below. This transpose

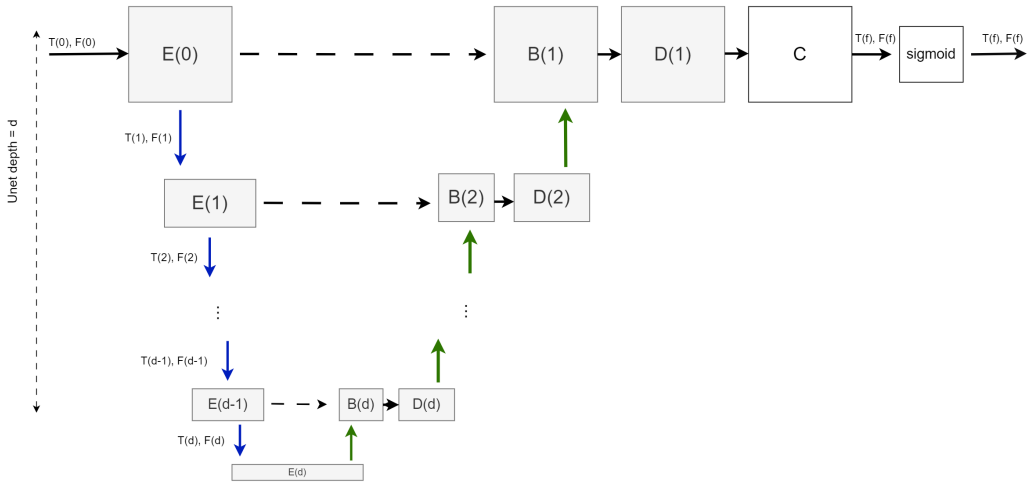
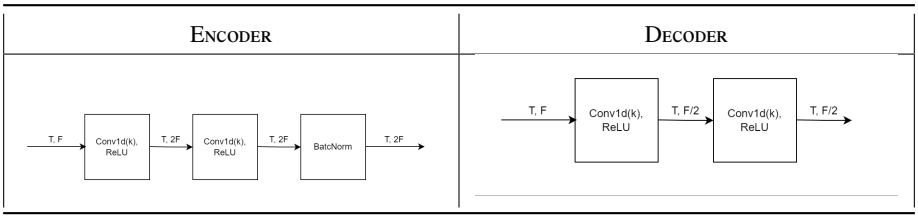


Figure 5: A pictorial representation of the UNET architecture that we use.

convolution also reduces the embedding dimension ×2. The blocks $B(i)$ denote the concatenation of the UNET skip connection (the dashed arrow) and the result of the transposed convolution along the feature dimension. Finally, the block C denotes a 1×1 convolution from an initial dimension (which after $D(1)$ is equal to $2F(0)$) into a final dimension equal to 2, followed by a sigmoid. This provides an output of dimension $T(0) \times 2$ of elements, each bounded in the range $[0, 1]$, that is then compared to the read-and-write ransomware fractions series using a binary cross-entropy loss function.

The *padding* and *output_padding* parameters of the transpose convolutions depend on the value of $T(i)$ (or more precisely on whether $T(i)$, which is determined by applying pooling on the higher-level series with size $T(i-1)$, obeys $T(i-1) = T(i) \times 2$ or $T(i-1) = T(i) \times 2 + 1$). The main UNET parameters are the depth d of the UNET and the kernel size k of the E and D blocks in Table 17 which we grid search for $d = 0, 1, 2, 3$ and $k = 3, 5, 7, 9$ (when the UNET depth is zero, $d = 0$, the architecture represented by the set of gray rectangles in Figure 5 reduces to $E(0)$). The initial dimensions $T(0)$ and $F(0)$ can also be considered as hyper-parameters, but $T(f), F(f)$ are not as we fix them to obey $T(f) = T(0)$ and $F(f) = 2$. We attempted various types of encoder and decoder blocks (the E and D blocks in Figure 5), including ResNet blocks and simple convolutions, and found that the best are those presented in Table 17.

Table 17: The encoder block E and decoder block D.



C Tokenization and Embedding Specifications

C.1 Data Sampling

To generate samples for training and testing, we sample data by slicing each NVMe stream into successive non-overlapping units of data slices. There are three natural options to define the slices: by equal (i) time extent, (ii) number of commands, or (iii) byte volume. A common choice in the literature is to use option (i) with a time extent of $\sim 10-60$ seconds (e.g. [9, 12, 5, 6]). Because the SotA models, as well as some of our models, can perform detection only after at least one data slice is complete, the earliest detection with option (i) at a typical I/O throughput of 130MB/sec can happen after over a gigabyte of data has been read/overwritten by ransomware. To avoid this issue and the potential lack of robustness to throughput variability, we choose to sample with option (ii) and a slice size of 16500 commands or option (iii) with a slice size of 0.5GiB. For brevity, we term the models that are based on slicing options (ii) and (iii) as `ByCommand` and `ByVolume` version models.

C.2 Command Level Tokenization

Each NVMe command, c, passes through the CLT tokenization scheme, as follows:

- The time lapse $\delta t(c) \equiv \min(round(\log((timestamp(c) - timestamp(c-1)) \cdot 10^5 + 1)), 15)$.
- $size \equiv \min(\lfloor \log(\frac{S}{512}) \rfloor, 12)$. Where S is the size of the command in bytes. Since 4 bits are assigned, the remaining 3 bins (13, 14, 15) are assigned to particularly common size values: 512K, 128K and 16K.
- The command's opcode (R/W) is assigned with 1 bit.
- The three auxiliary attributes - WAR, RAR, and RAW - are assigned 1 bit each, denoting whether their value for this command is greater than 0.
- The last (minor) 21 bits of the offset are dropped (this represents a 2MB resolution) and the next 2 bits are taken as $offset_{lsb}$.
- The first (major) 4 bits of the offset are taken as $offset_{msb}$.

In total, these attributes are spread across two tokens, where each token is composed of 9 bits: the first token contains the $\delta t(c)$, size, and opcode attributes, and the second receives the two offset attributes and the three auxiliary attributes. Then, an additional single-bit is added to each token (0 to the first, 1 to the second), denoting its index in the token pair, and creating a $2^{10} = 1024$ vocabulary size.

The finite number of bits that we allocate per each attribute cdetermines the overall quantization of the NVMe command and the above choice is one of several that we explored (see Table 28).

C.3 Patch Level Representation for the PLT and the UNET

To generate tokens, each slice is divided into 100 patches by sliding a window. When we model slices with a uniform number of commands (`ByCommand` slicing), we choose each patch to contain 250 commands with a stride of 165. When we model slices with uniform byte volume (`ByVolume` slicing) we choose each patch to have a total *size* of $50MB$ with stride $\simeq 5MB$.

C.3.1 Embedding

Distinct from the embedding method of [30] and because each command comes with attributes of different value types, we find that a natural way to embed the patch data is by quantizing attributes into bins and histogramming each across commands of certain types (read/write/write-after-read/read-after-read). Further details about this histogram embedding are in Table 18, where we denote by WAR and RAR commands with $OV_{WAR} > 0$ and $OV_{RAR} > 0$ respectively, and by $Rest$ the commands with no such overlaps. Some of the histograms are weighted by *size* or by OV. This weighting is chosen to reflect the fact that most ransomware commands carry a high *size* because they aim to read and write as fast as possible, thereby enhancing the impact of such commands. Finally, to avoid over-fitting to patterns on certain disk locations, we normalize the $offset$ values in a slice to zero mean and unit variance, and to increase robustness to throughput variability, we normalize δt and Δt by factors that represent their exponentially back averages across past slices.

In addition, to easily capture information about the total byte size and amount of commands in each patch, we calculate these for read, write, *WAR*, and *RAR* commands, as well as for all command types together. We normalize these additional 9 features to the range [0, 1], and concatenate them to the rest, forming a $d_{input} = 181$-dimensional embedding space per token.

Table 18: Details of patch tokens embedding. The $offset$ and δt histograms are weighted by *size* and the histograms for Δt_{WAR} and Δt_{RAR} by OV_{WAR} and OV_{RAR} respectively.

NVMe attribute	Histogram Weight	Commands types	Bins
log *size*	-	read/write/*Rest*	12
log *OV*	-	*WAR*, *RAR*	12
$offset$	*size*	read/write	14
		WAR, *RAR*, *Rest*	14
Δt	*OV*	*WAR*, *RAR*	14
δt	*size*	Any	14

C.3.2 Normalization of δt, Δt, and the $offset$

Starting with the *timestamp* related quantities, for each slice I and for a per-command attribute Q (for example $Q = \delta t$ or $\Delta t_{W,R}$) we denote by $\langle Q \rangle_{I,w}$ the average across all commands within a the slice performed with the per-command weight w_c,

$$\langle Q \rangle_{I,w} = \frac{\sum_{c \in I} Q_c w_c}{\sum_{c \in I} w_c}, \tag{1}$$

and by $\langle\langle Q \rangle\rangle_{I,w}$ the exponentially back averaged of $Q_{I,w}$ across slices, defined recursively by $\langle\langle Q \rangle\rangle_{I,w} = \alpha \langle\langle Q \rangle\rangle_{I-1,w} + (1-\alpha)\langle Q \rangle_{I,w}$. In this work, we choose the diminish factor $\alpha = 0.8$, which means the effective memory of the average is $\sim 1/\log \alpha^{-1} \sim 4.48$ slices. For the purpose of featurization, we define the following normalized quantities.

$$\overline{offset}_I \equiv \left(offset - \langle offset \rangle_{I,size^2}\right) / \sqrt{\langle offset^2 \rangle_{I,size^2}}, \tag{2}$$

$$\overline{\delta t}_I \equiv \delta t / \langle\langle \delta t \rangle\rangle_{I,size^2}, \tag{3}$$

$$\overline{\Delta t}_I \equiv \Delta t / \left(10 \langle\langle \delta t \rangle\rangle_{I,OV^2}\right). \tag{4}$$

The purpose of the δt normalization is to create an embedding that is invariant to CPU speed while normalizing the *size* is aimed to achieve a disk size invariant embedding.

C.3.3 Normalization of per-token IO size and IO byte

As mentioned in Appendix C.3.1 we concatenate to the histogram features of Table 18 a set of 9 features associated with the number and volume of commands in a patch token. We describe their normalization in Table 19 and Table 20. Here, the normalization factors v_0 and n_0 are the designed token width in the `ByVolume` and `ByCommand` version models and were chosen as $v_0 = 50MB$ and $n_0 = 250$.

Table 19: The details of the last 9 features participating in the token embedding for the `ByVolume` slicing.

Feature	Normalization factor	Number of features
Patch volume	v_0	1
Patch read and write volume	v_0	2
Sum of OV_{WAR} and OV_{RAR} in a patch	v_0	2
Number of read/write commands in a patch	Number of commands in a patch	2
Number of *WAR* and *RAR* commands in a patch	Number of commands in a patch	2

Table 20: The details of the last 9 features participating in the token embedding for the `ByCommand` slicing.

Feature	Normalization factor	Number of features
Patch number of commands	n_0	1
Patch read and write number of commands	n_0	2
Number of *WAR* and *RAR* commands in a patch	n_0	2
Volume of read/write commands in a patch	Patch volume	2
Sum of OV_{WAR} and OV_{RAR} in a patch	Patch volume	2

D Hyper-parameter optimizations

In the current section we describe the optimization that we performed across the hyper-parameters of the models, the embeddings, and the tokenizations (all described in Appendix B and Appendix C).

D.1 Random Forest (RF)

The RF parameters which we optimized include

- The way one samples the data – for aggregative models like the RF, one performs prediction per data chunk; the latter is defined to have uniform traffic volume ('ByVolume' chunks) or to have a uniform number of commands ('ByCommand' chunks). See results in Table 21.
- The size of the chunks – see results in Table 22.
- The number of trees N, the maximum tree depth D. See results in Table 23.
- The type of feature sets – the nominal set was the 23-dimensional feature set described in Appendix B.1 and the second was the concatenation of all the UNET/PLT features described in Appendix C.3.1 (constituting – for 100 tokens – a 18,100 dimensional feature vector). See Table 24 for the results.

All the results show that the best RF model is the one that we describe in Appendix B.1.

Table 21: Performance of the RF model in the nominal configuration of $D = 20$, $N = 20$ on CLEAR with two chunk types. Since the chunks in both models are difference, we use a sampling invariant property to compare them and here choose the MDB_3. Both models were calibrated to have approximately 1 false alarm per 50GiB.

Configuration	MBD_3
ByVolume chunks of size $0.5GiB$	286 ± 33
ByCommand chunks of size $16,500$	335 ± 53

Table 22: Performance of the RF model in the nominal configuration of $D = 20$, $N = 20$ on a subset of CLEAR that contains 659 recordings, with two chunk sizes of the ByVolume chunk type – the nominal and a chunk smaller by a factor of 10. Both models were calibrated to to have the same false alarm rate (of 1 per $50GiB$). Also, we present the 99%-precentile instead of the 3rd quartile.

Chunk size	$MBD_{99\%}$
$0.5GiB$	749 ± 63
$51.2MiB$	1316 ± 996

D.2 DeftPunk

The DeftPunk model is described in [5] and we have established independently that its two-stage structure is indeed required by benchmarking its performance with and without the second stage (the XGBoost step) – see Table 25.

D.3 UNET

We optimized the UNET architecture across the following parameters: the UNET depth d, the convolutional kernel k, the structure of the encoder and decoder blocks (with their own parameters), the learning rate, the embedding dimension and the number of tokens (in addition to checking the UNET applied to both ByVolume and ByCommand chunks). In Table 26 we present a subset of the tests leading us to choose the UNET configuration with $k = 5$, $d = 3$, a ByVolume chunk of size 0.5GiB, and standard convolutional encoder and decoder blocks.

Table 23: Performance of the RF model with the ByVolume chunk with a 0.5GiB chunk size across a subset of CLEAR containing 280 recordings, all calibrated to an aggressive point of zero false alarms on that data. Since all models were trained on the same type of chunks we show the 95% confidence interval of the the false negative rate as calculated by Clopper-Pearson.

Maximum Tree depth D	Number of trees N	False Negative rate confidence interval
10	10	[0.052,0.107]
10	20	[0.059,0.116]
10	50	[0.05,0.105]
20	10	[0.036,0.084]
20	**20**	**[0.033,0.081]**
20	50	[0.036,0.084]
50	10	[0.036,0.084]
50	20	[0.033,0.081]
50	50	[0.036,0.084]

Table 24: Performance of the RF model in the nominal configuration of $D = 20$, $N = 20$ on CLEAR with the nominal type of chunk with two different feauture sets: the nominal one with 23 features and the PLT/UNET feature set of dimension 18,100 (see discussion in the text). Results were obtained from a subset of CLEAR containing 2725 recordings.

Feature set	MBD_3
RF nominal 23 features	264 ± 33
The PLT/UNET tokens concatenated	299 ± 36

D.4 PLT

The PLT transformer architecture was explored along the number of attention heads, the number of encoder layers, the embedding dimension (the feedforward dimension was always x4 larger). The results of this HPO are presented in Table 27, showing why we chose the parameters listed in Table 15.

D.5 The CLT and the LSTM models

The hyper parameter optimization we performed for the command-level models was across the following axes:

1. Token quantization scheme: as per the description in Appendix C we tested several quantization schemes that we describe in Table 28.
2. Model size: we detail the model sizes tested for each model in Table 29.
3. We trained each model on different context lengths.
4. We also tested a continuous version of the unidirectional LSTM where the inference was done continuously without resetting the LSTM memory every slice; the two continuous models in our tests differ by the context length used for training the models.

We present the HPO results for the MDR, the MBD_3, and the AUC in Table 30.

Table 25: Performance of the DeftPunk model with or without the second step applied on the ByVolume chunk of size 0.5GiB.

DeftPunk Architecture	MBD_3
Decision tree only	7878 ± 2122
Decision tree + XGBoost	282 ± 34

Table 26: Performance of the UNET model as a function of the chunk type, UNET depth d, and UNET convolutional kernel k. All results in this table were obtained for the nominal encoder and decoder blocks that are described in Appendix B.5.

Chunk type	d	k	MDR [%]	MBD_3	AUC	ROC Threshold
ByVolume	0	3	1.47±0.16	219±13	0.999133±0.000067	0.232454±0.014314
	1	3	1.72±0.34	241±25	0.998914±0.000085	0.257555±0.031964
	2	3	1.49±0.27	211±24	0.999059±0.000090	0.225127±0.028576
	3	**5**	**1.34±0.41**	**195±41**	**0.998994±0.000093**	**0.191552±0.051176**
	3	7	-	199±50	0.998965±0.000132	0.193606±0.070926
	3	9	-	260±89	0.998861±0.000138	0.281482±0.125255
ByCommand	3	3	-	587±303	-	0.466428±0.072315

Table 27: Performance of the PLT model as a function of the number of attention heads n_{head}, number of encoder layers n_{layers}, embedding dimension d_{model}, number of tokens N_t and chunk type. By * we denote models that were trained with a learning rate schedule that decreases the learning rate by $\times 0.8$ every 75 epochs, and by ** we denote models applied onto 512 patch tokens (rather the nominal 100 patch tokens).

Chunk type	d_{model}	n_{head}	n_{layer}	MBD_3	ROC Threshold
ByVolume	256	4	6	183±23	0.17942±0.034166
	512	4	6	177±25	0.16844±0.036366
	512	4	3	162±22	0.150087±0.027717
	512	**4**	**6**	**157±18**	**0.12212±0.027323**
	512	16	16	169±22	0.145905±0.030539
	512*	4	6	180±23	0.159498±0.035502
	256**	6	4	196±36	0.17822±0.044973
	512**	16	16	217±39	0.215373±0.053777
	512**	16	16	183±30	0.156018±0.038017
ByCommand	512*	4	6	353±175	0.37189±0.079117
	512	4	6	746±442	0.497779±0.099673
	256	4	6	717±605	0.497168±0.134967
	512	4	3	207±100	0.254427±0.079234

Table 28: The token bit quantization schemes.

Quantization scheme	Number of tokens per command	Number of Bits used for each Attribute							
		δt	$size$	$opcode$	$offset_{\text{msb}}$	$offset_{\text{lsb}}$	WAR	RAR	RAW
Nominal	2	4	4	1	4	2	1	1	1
1	2	4	4	1	5	4	1	0	0
2	1	3	4	1	4	2	1	0	0
3	1	4	4	1	4	2	1	1	1

Table 29: Model size.

Architecture	Nominal size	Large size
CLT	embedding dim=128, FF dim=128, layers=3, heads=4	embedding dim=512, FF dim=512, layers=6, heads=4
uLSTM	embedding dim=128, FF dim=128, layers=3	embedding dim=512, FF dim=512, layers=5
biLSTM	embedding dim=128, FF dim=128, layers=3	embedding dim=512, FF dim=512, layers=5

Table 30: The results of the hyper-parameter exploration for the command level models. In cases where the quantization scheme or the model size were not nominal we denote them in the model name: the *i*-th quantization schemes are marked as 'quantization-i' (see Table 28), the large size models are marked by a subscript *L*, and the continuous LSTM by a subscript *C*. We note in passing that attempts to train a nominal sized CLT with context = 4000, and any models with context = 8000, 16500, and did not converge.

Context	Model Name	AUC	MDR	$MBD_3/1000$	ROC Threshold
62	CLT	0.998818±0.000168	0.0318±0.012	0.118±0.03	0.2399±0.04673
	biLSTM	0.998746±0.00018	0.031±0.0156	0.122±0.049	0.23132±0.05483
	uLSTM	0.98609±0.001364	0.1355±0.017	0.156±0.107	0.29042±0.05906
125	CLT	0.99918±0.00007	0.0144±0.0028	0.092±0.009	0.12703±0.01505
	biLSTM	0.999032±0.000156	0.0199±0.007	0.095±0.015	0.17432±0.03439
	uLSTM	0.98609±0.001364	0.1355±0.017	0.156±0.107	0.29042±0.05906
250	CLT	0.999043±0.000152	0.0201±0.0065	0.097±0.012	0.17253±0.03187
	CLT quantization-1	0.998993±0.000159	0.012±0.0033	0.082±0.009	0.12496±0.020198
	CLT quantization-3	0.999184±0.000135	0.0146±0.0042	0.089±0.01	0.12543±0.02305
	biLSTM	0.999161±0.000142	0.0158±0.0057	0.088±0.012	0.14498±0.02899
	uLSTM	0.998884±0.000175	0.0241±0.008	0.104±0.016	0.20845±0.03513
	uLSTM$_C$	0.999492±0.000091	0.0062±0.0016	0.063±0.007	0.06051±0.01235
500	CLT	0.999148±0.000139	0.0163±0.0051	0.087±0.011	0.14868±0.02684
	biLSTM	0.999231±0.000128	0.0152±0.0051	0.088±0.012	0.13812±0.02645
	uLSTM	0.999077±0.000148	0.0195±0.0069	0.094±0.013	0.1752±0.03362
1000	CLT	0.999276±0.000112	0.0142±0.0043	0.077±0.01	0.12485±0.02379
	CLT quantization-2	0.999213±0.000127	0.0114±0.0025	0.072±0.007	0.100498±0.015934
	biLSTM	0.999322±0.000106	0.0124±0.0029	0.076±0.007	0.11677±0.01829
	uLSTM	0.999167±0.000133	0.0163±0.0049	0.082±0.009	0.15064±0.02528
2000	CLT	0.999237±0.000114	0.0172±0.0044	0.082±0.01	0.1438±0.0229
	CLT$_L$	0.999476±0.000082	0.0124±0.0031	0.075±0.008	0.11038±0.01885
	CLT quantization-2	0.999271±0.000112	0.0127±0.0033	0.078±0.009	0.101097±0.020642
	biLSTM	0.999307±0.000109	0.0123±0.0034	0.075±0.007	0.11696±0.02154
	uLSTM	0.999218±0.000128	0.0132±0.0031	0.078±0.007	0.12739±0.01696
	uLSTM$_C$	**0.999664±0.000065**	**0.0036±0.0008**	**0.05±0.003**	**0.02615±0.00391**
4000	CLT$_L$	0.999514±8.2e-05	0.009±0.0029	0.074±0.008	0.07487±0.02038
	CLT quantization-2	0.999033±0.000139	0.0164±0.0053	0.084±0.011	0.137523±0.032655
	biLSTM	0.99916±0.000128	0.0143±0.004	0.08±0.009	0.13622±0.02303
	biLSTM$_L$	0.999305±0.000101	0.0117±0.0032	0.072±0.007	0.11422±0.02062
	uLSTM	0.999214±0.000121	0.014±0.0035	0.082±0.009	0.14095±0.02149
	uLSTM$_L$	0.999359±9.8e-05	0.0103±0.0024	0.069±0.007	0.1049±0.0142