MUSTARD: Adaptive Behavioral Analysis for Ransomware Detection

Davide Sanvito, Giuseppe Siracusano, Roberto Gonzalez, Roberto Bifulco
NEC Laboratories Europe
{name.surname}@neclab.eu

ABSTRACT

Behavioural analysis based on filesystem operations is one of the most promising approaches for the detection of ransomware. Nonetheless, tracking all the operations on all the files for all the processes can introduce a significant overhead on the monitored system. In this paper, we present MUSTARD, a solution to dynamically adapt the degree of monitoring for each process based on their behaviour to achieve a reduction of monitoring resources for the benign processes.

1 INTRODUCTION

Ransomware attacks are a growing source of concern [1–3]. Among the solutions proposed to address them [7, 8, 10], dynamic system behavioural analysis is a promising approach. Indeed, most ransomwares access and modify many files in small time, thereby exposing a peculiar behaviour, which is detected by solutions in the state-of-the-art [4, 5, 11, 12]. These solutions apply a set of common steps: (i) collection of features by monitoring Operating System (OS) interfaces and accesses to the filesystems; (ii) analysis of the collected features to build per-process behaviors; (iii) classification of the modeled behaviors. These steps introduce a large overhead on running systems, limiting their applicability in the real world.

In general, dynamic behavioural analysis methods rely on a set of features that are collected at the OS level. Often, individual features require the combination of raw counters collected at different system/OS interfaces. Furthermore, while some features are as easy as e.g., counting the number of file accesses, others include more complex operations. Therefore, the monitoring overhead depends both on the number of monitored features and on their types. For instance, the time to copy a 10GB file is 10% longer when just enabling the OS interfaces required for monitoring.[1] Adding the simple read and increment of a counter in a hash-table increases the overhead to 20%. Furthermore, some features may require significantly more compute intensive steps, e.g., computing the entropy of the written data [4–6]. In our simple file copy test, computing such complex features might increase the copy time by a factor of over 10x (from GB/s to 100MB/s of write throughput).

We focus on reducing this monitoring overhead. Our idea is to investigate the opportunity to monitor only a subset of the required features at most times, enabling the collection of additional features only when a monitored process exhibits a *behavior* compatible to a ransomware. We build on the intuition that ransomwares perform a set of unavoidable steps: listing files; accessing many of them; reading and writing to the file; writing encrypted content. Only a relatively small number of the OS processes perform a similar set of steps, creating an opportunity to focus our monitoring on those.

[1] We run the test using Linux, ext4, and the recently introduced eBPF framework.

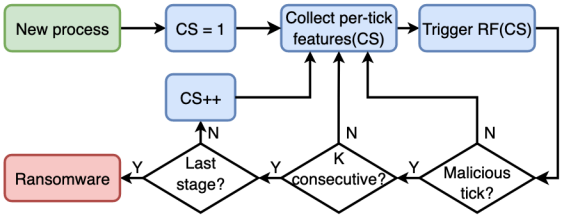

Figure 1: Workflow of the Multi-Stage approach: `CS`, `features(CS)` and `RF(CS)` denote respectively the Current Stage and its corresponding input features and Random Forest model.

Therefore, we build MUSTARD[2], a monitoring system that dynamically cascades a series of detection models, each of them using an incremental number of features, in order to use few simpler features for most monitored processes, and more complex ones only for the processes that are suspected to be ransomwares. Using a popular public dataset [9], we show that MUSTARD can keep detection performance, reducing monitoring overhead, at the cost of a small increase in detection latency.

2 MUSTARD

We build MUSTARD, a Multi-Stage detection mechanism that uses an increasing number of features at each detection stage. For each process in the system, only few features are initially monitored, then, depending on the result of the current step (i.e., suspect ransomware or not), the number of monitored features is incremented for the next stage. This keeps the monitoring overhead low, since complex monitoring operations are progressively switched-on depending on the needed features. That is, instead of always monitoring any possible system/process feature, we increase the set of monitored features depending on the behaviour of the monitored processes.

We perform analysis for each OS' process using a pipeline of N stages. Each stage implements a machine learning classifier trained to differentiate among normal and malicious processes. The machine learning models in later stages have an increasing number of input features. Fig. 1 summarizes the MUSTARD high-level workflow.

At runtime, all processes are periodically analyzed, i.e., at each *detection interval* (`tick`). Nonetheless, each process may be analyzed over time using a different model, i.e. one of the pipeline's stages. In fact, all processes are initially analyzed using the first stage. If a process is marked as *malicious* (i.e., suspected ransomware) for K consecutive `ticks`, then its analysis in the next `tick` will be executed by the next stage based on an increased number of features. This continues until the a process is classified as malicious in K consecutive `ticks`, when using the last stage in the pipeline. In such a case, the system classifies the process as ransomware.

[2] **Mu**lti-**St**age **A**daptive **R**ansomware **D**etection

In the remainder, we denote as *MS(f_1, f_2 ... f_N)* the Multi-Stage pipeline comprising N models in cascade based on f_1, f_2 ... f_N input features, where $f_1 < f_2 < ... < f_N$.

Detection interval. We perform detection at regular intervals, called *ticks*. Instead of using time-based triggers, we rely on the approach used in [4], where the detection mechanism is executed every time a process interacts with a predefined number of files. This approach: (i) minimizes the time to decision (avoiding waiting the entire execution of the monitoring period); (ii) avoids unnecessary execution, e.g. when a process is frequently modifying a single file; (iii) and makes the system more robust to ransomwares that attempt time-based cloaking.

Features. The set of used features may vary depending on the specific methods. In this paper we consider six main features: folder listing; file read/write/rename operations; file type coverage; and write entropy. This set of features was successfully tested in [4], and covers most of the features used in the state of the art.

Classifier. We use a Random Forest (RF) as classifier in each stage. For the tests in the next section, we implemented the RF using scikit-learn using similar parameters to those of [4]: RFs are based on 100 Decision Trees, $K = 3$ and the detection interval is based on a threshold of 0.1% of files. In our preliminary implementation, we manually define the sequence of stages, i.e., which features are added as input in each pipeline's stage. This might introduce a bias, since we are familiar with the dataset used in evaluation. In our future work we plan to address this issue by automating the pipeline design. Here, we are guided by two goals: (i) earlier stages in the pipeline should introduce the lowest monitoring overhead; (ii) earlier stages should have higher recall. In fact, for the first pipeline stages we favor recall over precision, since false positives can be filtered out by the downstream stages. At the same time, if the early stages introduce little monitoring overhead, we have an opportunity to avoid heavier monitoring for a large number of non-malicious processes.

3 EVALUATION

For evaluation, we use a public dataset [9]. The dataset includes one month of normal data captured from 11 real machines (1.7B IRPs[3] from more than 2000 applications for personal use, office and software development). Malicious data comes instead from the execution of 381 Windows Ransomware. Further details about the features and the data collection are reported in [4].

Baseline. Our goal is to evaluate how incrementally adding features impacts on the classification accuracy, latency and on the monitoring overhead. Therefore, we compare MUSTARD against a baseline classifier that uses all features at all times, and which is representative of the current state-of-the-art methods, e.g., [4].[4] The baseline classifier is equivalent to the *MS(6)* configuration.

We test MUSTARD changing the length of the pipeline (N) and the number of input features in each stage (f_x, $1 \leq x \leq N$). Results are computed on 10 random train/test splits of the dataset (each split takes into account processes, i.e., each process is either in the train or in the test split). In all cases we report both the average and standard deviation for the measured metrics, which are:

- True Positive Rate (TPR) or Recall (R): `TP/(TP+FN)`
- True Negative Rate (TNR) or Selectivity (S): `TN/(TN+FP)`
- Precision (P): `TP/(TP+FP)`
- F1-score: `2*P*R/(P+R)`

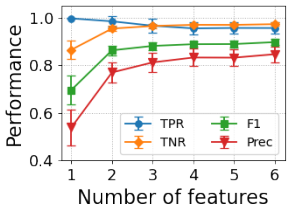

Figure 2: Detection performance with RF model (K consecutive ticks) when using from one to six features.

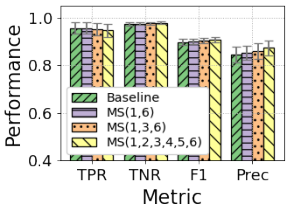

Figure 3: Detection performance when features monitoring is incrementally increased depending on the detected behaviour.

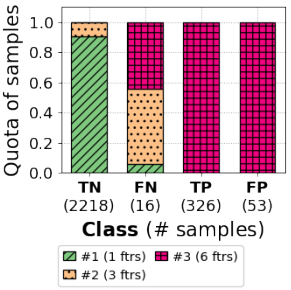

Figure 4: Final decision stage for *MS(1,3,6)* configuration.

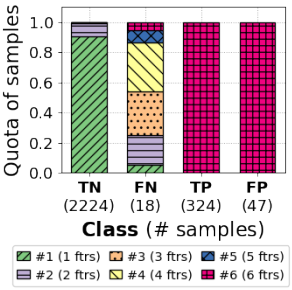

Figure 5: Final decision stage for *MS(1,2,3,4,5,6)* configuration.

Baseline vs number of features. First, we train six independent RF models on an increasing number of input features (*MS(1)* to *MS(6)*). The six models represent the individual per-stage models of MUSTARD, with *MS(6)* being our baseline. Fig. 2 shows that the TNR, F1-score and precision improve with more features. TPR instead lowers. The high TPR of the first stages is good news for MUSTARD, since this effectively means that such stages will select most processes that are actual ransomwares (true positives) for further analysis, while avoiding more features collection for a large share of normal processes.

Multi-Stage performance. We then evaluate the performance of three different MUSTARD configurations, with 2, 3 and 6 stages, respectively, and for the baseline. Fig. 3 shows that all configurations achieve performance comparable with the baseline. The TPR slightly decreases with longer pipelines, whereas Precision improves due to a reduced number of false positives (FPs), reflected also in the TNR and F1-score improvements.

Last activated stage. We provide further insights on the results checking in which stage each analyzed process is finally classified. Fig. 4 shows this for the *MS(1,3,6)* pipeline, dividing the processes in the 4 possible classification categories (true/false positive/negative). The vast majority (90%) of true negatives (TN), i.e., normal processes, are classified in the first stage. These processes are monitored with a single feature for their entire lifetime. True and false positives (TPs and FPs) are always classified in the last stage, by design. It is

[3] I/O Request Packets

[4] It should be noted that previous methods generally use a larger set of features than our preliminary implementation. Nonetheless our evaluation shows that the first stages achieve close to 100% recall, making our results valid even for cases that use a larger set of features at later stages.

	Detection latency (baseline)	*Additional* ticks wrt baseline		
		MS(1,6)	*MS(1,3,6)*	*MS(1,2,3,4,5,6)*
min	3.0	+3.0	+6.0	+15.0
avg	8.2	+4.4	+8.6	+20.1
95p	41.2	+14.0	+17.0	+45.0
max	241.0	+49.0	+151.0	+166.0

Table 1: Detection latency when using the baseline and additional detection latency when using the Multi-Stage approach with 3 different configurations. All the numbers are measured in ticks.

interesting to see that the (few) false negatives, i.e., ransomwares classified as normal processes, are mostly mis-classified at the second and last stage. The mis-classification in the middle stages might be problematic, since ransomwares will never be analyzed by the more powerful classifiers in the last pipeline stages. We see this issue becoming more evident for the *MS(1,2,3,4,5,6)* pipeline (Fig. 5), where 85% of the mis-classification happen in the middle stages. This result suggests that, in operational settings, both the type of classifiers and the pipeline length should be carefully selected to minimize false negatives classifications in the middle stages.

In any case, MUSTARD achieves its original design goal: it keeps comparable classification performance to the baseline, while monitoring only 10% of the TN processes with more than a single feature. It should be noted that in terms of generated overhead, the TN processes are the ones responsible for the most load, since they are executed until termination. True and false positives, conversely, are terminated as soon as they are classified.

Ransomware detection latency. MUSTARD requires a process to progress until the last pipeline stage, before providing a ransomware verdict. This increases the classification latency, which might be problematic since a ransomware might have more time to operate. We measure the detection latency in the number of ticks produced by a process before a ransomware verdict is provided.

Table 1 reports the detection latency for the baseline and the number of *additional* ticks required by MUSTARD under three configurations. For both metrics we computed the min, average, 95-percentile and max values. We conservatively report the worst measured values across the 10 random splits.

As expected, we can observe that shorter pipelines provide lower classification latency, with at most an additional 14 and 17 ticks for *MS(1,6)* and *MS(1,3,6)*, respectively, in 95% of the cases. *MS(1,2,3,4,5,6)* requires instead +45 ticks. The max value shows that in some cases the detection latency might grow significantly.

Waiting an extra amount of ticks translates in a larger quota of files being encrypted, before taking mitigation actions. In fact, the tick definition is related to the number of files accessed by a process (cf. Sec. 2). In our tests, a tick corresponds to 0.1% of all files in the filesystem. Thus, in the best case the baseline classifies a ransomware after it encrypted 0.3% of the files, whereas with MUSTARD at minimum we measure 0.6%, 0.9% and 1.8% for the three configurations. The worst case is 24.1% for the baseline, and 29%, 39.2% and 40.7% for the three MUSTARD configurations.

4 CONCLUSION AND FUTURE WORK

Our preliminary results show that MUSTARD could significantly reduce the monitoring overhead, limiting the number of collected features for over 90% of the monitored processes. This is important, since monitoring overheads can be large enough to make system deployment unfeasible in real world settings. However, our work is still preliminary and we are extending it to investigate several issues.

Latency. First, MUSTARD introduces an increase in terms of classification latency, which suggests a need to investigate the trade-off between performance overhead and increased risk.

Overhead measurement. Second, using a public dataset we could not measure the actual overhead introduced by feature collection and classification. This requires collecting the dataset from scratch, to assess the impact on the running system and the hosted workloads. We plan to extend our evaluation by creating our own dataset to make a complete evaluation of the monitoring costs. This is an aspect which is inherently system-dependent, and should be measured while capturing the dataset.

Hyperparameters. Third, MUSTARD includes (and inherits from the state-of-the-art methods) several hyperparameters. Evaluating their impact on the classification performance using multiple datasets is crucial to avoid the pitfalls of data biases.

Cloaking. Finally, new ransomwares may devise techniques to hide from the monitoring system. Evaluating MUSTARD in the context of a challenging threat model that assumes attackers with full system knowledge is also part of our future work.

Despite the challenges listed above, we believe MUSTARD provides an interesting direction for the exploration of the trade-offs between monitoring overhead and risk mitigation. We believe that investigating these trade-offs will become increasingly important in the near future, to ensure the ability to deploy such systems in production environments.

REFERENCES

[1] 2017. *State of Malware Report.* Technical Report. Malwarebytes. https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf Accessed July 2022.

[2] 2020. *2020 State of Malware Report.* Technical Report. Malwarebytes. https://www.malwarebytes.com/resources/files/2020/02/2020_state-of-malware-report.pdf Accessed July 2022.

[3] 2022. *2022 Threat Review.* Technical Report. Malwarebytes. https://www.malwarebytes.com/resources/malwarebytes-threat-review-2022/mwb_threatreview_2022_ss_v1.pdf Accessed July 2022.

[4] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proceedings of the 32nd annual conference on computer security applications*. 336–347.

[5] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX security symposium (USENIX Security 16)*. 757–772.

[6] Kyungroul Lee, Sun-Young Lee, and Kangbin Yim. 2019. Machine learning based file entropy analysis for ransomware detection in backup systems. *IEEE Access* 7 (2019), 110205–110215.

[7] Timothy McIntosh, ASM Kayes, Yi-Ping Phoebe Chen, Alex Ng, and Paul Watters. 2021. Ransomware mitigation in the modern era: A comprehensive review, research challenges, and future directions. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–36.

[8] Routa Moussaileb, Nora Cuppens, Jean-Louis Lanet, and Hélène Le Bouder. 2021. A survey on windows-based ransomware taxonomy and detection mechanisms. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–36.

[9] NECSTLab. 2022. ShieldFS dataset. http://shieldfs.necst.it/. (2022). [Online; accessed July 2022].

[10] Harun Oz, Ahmet Aris, Albert Levi, and A Selcuk Uluagac. 2021. A survey on ransomware: Evolution, taxonomy, and defense solutions. *ACM Computing Surveys (CSUR)* (2021).

[11] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin RB Butler. 2016. Cryptolock (and drop it): stopping ransomware attacks on user data. In *2016 IEEE 36th international conference on distributed computing systems (ICDCS)*. IEEE, 303–312.

[12] Kimberly Tam, Aristide Fattori, Salahuddin Khan, and Lorenzo Cavallaro. 2015. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS Symposium 2015*. 1–15.