

Hardware-Assisted Multi-Stage Ransomware Detection with Adaptive Machine Learning

Salma Aboelmagd*, Md. Rakibul Ahasan*, Zhixin Pan*, Abdulrahman Takiddin*, and Ziyu Shu†

* Florida State University, Tallahassee, FL, USA

† Stony Brook University, Stony Brook, NY, USA

Abstract—Ransomware is a fast-escalating privacy threat causing significant financial and data losses worldwide. As attacks grow more sophisticated, traditional static signatures and dynamic behavior analysis struggle to defend. In this paper, we propose an novel framework that couples software-level scanning with hardware-level microprocessor activity monitoring to achieve efficient ransomware detection and mitigation. The contributions are threefold: (1) we employ adversarial training to harden static analysis against obfuscation and leverage hardware-assisted monitoring for real-time detection; (2) we integrate neural architecture search (NAS) to automatically tailor models for stronger generalizability; and (3) We validate the proposed approach through extensive experiments, demonstrating substantially improved detection accuracy and reduced latency compared with existing methods. Moreover, it provides trace-level interpretability that explains the underlying decision process.

I. INTRODUCTION

Cybersecurity has remained a major concern since the emergence of networked computing systems. Among various security threats, *ransomware*, a special type of malicious software that encrypts critical data and demands payment for its release, has become one of the most destructive and financially damaging forms of cyberattacks. Its growing sophistication has resulted in significant economic losses, operational malfunction, and exposure of sensitive information worldwide. As shown in a recent study [1], ransomware attacks are increasingly targeting large organizations, with estimated global damages approaching \$6 trillion annually. Moreover, attackers have adopted advanced obfuscation and evasion strategies, rendering conventional defenses such as signature-based detection and fuzzing ineffective, highlighting the need for more adaptive detection and mitigation approaches.

and scanning the system for valuable files. The *execution phase* then rapidly performs destructive actions depending on the ransomware type: (i) Crypto (file encryption with ransom demand), (ii) Locker (blocking system access), (iii) Scareware (displaying fake alerts), or (iv) Hybrid (combining multiple attacks).

Existing ransomware detection methods can be broadly categorized into *static analysis* and *dynamic analysis* approaches. Static analysis inspects executable files or code structures before execution, typically through signature matching or feature recognition. While efficient, such techniques often fail against obfuscated or polymorphic malware, resulting in high false positive rates [1], [2]. Dynamic or behavioral analysis, instead, monitors runtime activities such as file operations and system calls to identify malicious patterns [3]. Although generally more accurate, these approaches are often platform-dependent and suffer from latency, while ransomware protection is a time-critical task. Furthermore, most behavior-based systems produce binary detection results without explaining the underlying reasoning, making it hard to perform post-detection actions like file restoration or infection localization.

To overcome these challenges, this work proposes a *hardware-assisted, machine learning (ML) based ransomware detection framework* that integrates static analysis with dynamic runtime monitoring in a unified architecture. Unlike prior hybrid approaches that rely on fixed rules or manual intervention to balance accuracy and overhead, our framework is centered on adaptive hardware-assisted monitoring and makes the following contributions:

- **Multi-stage detection pipeline:** We propose a hierarchical detection pipeline that integrates static inspection with dynamic hardware-assisted analysis, activating fine-grained runtime monitoring only when necessary to reduce detection latency and system overhead.
- **Dynamic hardware-assisted monitoring:** We develop an adaptive runtime monitoring strategy that combines coarse-grained observation with selective fine-grained runtime tracing, enabling low-overhead detection with timely confirmation under obfuscated ransomware behaviors.
- **Adaptive ML optimization:** A neural architecture search (NAS) mechanism automatically configures learning models for both static and dynamic stages, improving robustness and generalizability across diverse ransomware variants.

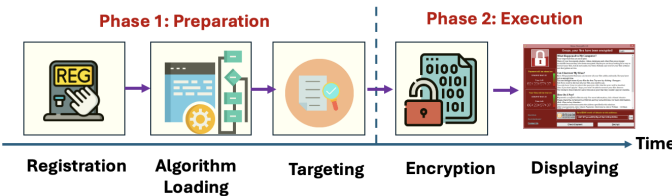


Fig. 1: Typical ransomware infection workflow consisting of two major phases.

Figure 1 illustrates a typical ransomware infection timeline, which generally consists of two major stages. The *preparation phase* involves initialization, loading encryption algorithms,

The remainder of this paper is structured as follows. Section II reviews the related studies. Section III presents the proposed detection framework. Section IV discusses experimental evaluations, and Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

The research on ransomware detection has gained growing attention in recent years. As discussed in Section I, the majority of existing methods can be broadly categorized into two types: *static analysis* and *dynamic analysis*.

Static analysis is conducted before program execution, where the executable or source code is examined to identify malicious intent. For instance, [4]–[8] utilized deep file-scanning and code-inspection techniques to detect potential ransomware, while [9]–[11] explored linguistic analysis of program code for early-stage detection. However, static approaches lack the behavioral information from runtime, which limits their ability. Moreover, certain benign applications, such as disk-encryption tools, may share similar binary patterns with ransomware, causing high false positive rates (FPRs). Additionally, many static methods rely heavily on signature-based pattern matching; when attackers could intentionally disguise or obfuscate code [12], these methods may fail to detect the threat accurately.

Dynamic analysis, on the other hand, involves executing suspicious programs to monitor runtime behavior. For example, [13] leveraged runtime performance traces to identify high-frequency encryption patterns, enabling faster detection, while [14] investigated a framework that tracks dynamic variations in portable executable headers. Several other studies [15], [16] have incorporated machine learning (ML) models to further improve detection accuracy. Despite promising results, these methods still face several challenges. First, many fail to enforce early termination of ransomware, allowing significant data encryption before detection. Second, ML-based systems often behave as black boxes without interpretable reasoning, which undermines user confidence in critical applications. Finally, most approaches are evaluated only on limited ransomware families, reducing their generalizability. As shown in Fig. 2, existing algorithms exhibit fluctuating performance across four representative variants—*WannaCry*, *Vipasana*, *Locky*, and *Petya*—with detection rates occasionally dropping below 70%.

III. PROPOSED METHOD

A. Overview

To highlight the motivation for our proposed method, we summarize the key limitations of existing anti-ransomware techniques discussed in Section II, as follows:

- Vulnerability to obfuscation.
- Lack of prioritization of early termination.
- Low generalizability among variants of ransomware.
- Lack of transparency in decision making.

To address these challenges, our proposed approach enables a synergistic integration of a robust static detection framework and hardware-assisted dynamic analysis for efficient detection.

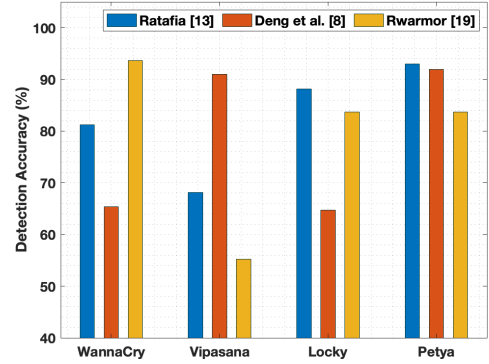


Fig. 2: Detection accuracy of various methods against four ransomware variants.

Figure 3 presents an overview of the proposed workflow, modeled as a finite state machine (FSM).

In the first stage, the target program undergoes static analysis for early-stage detection. If a program is flagged as ‘suspicious’ at this step, it is isolated to prevent further infection and passed to the dynamic analysis phase. This two-stage sequential framework offers several key advantages: it mitigates the issue of high false positive rates in conventional static analysis by labeling programs as suspicious rather than directly classifying them as malicious. Additionally, only programs flagged in the first stage proceed to runtime monitoring, which reduces unnecessary computational overhead. Next, based on the results of dynamic analysis, the program is either restored to its original state if deemed benign or, if identified as malicious, our method terminates its execution and restores encrypted files using backups.

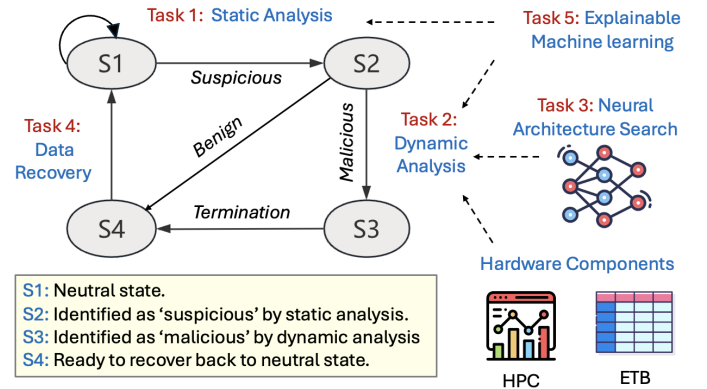


Fig. 3: State transitions of proposed framework.

The implementation of the workflow described above can be divided into five primary tasks:

- A **static analysis** phase, utilizing a boosted decision tree (DT) for early detection, where adversarial training is applied to mitigate the vulnerability to obfuscation.
- A **hardware-assisted dynamic analysis** phase, leveraging a long-short term memory (LSTM) model to monitor runtime behavior with features from hardware components, including hardware performance counters (HPC) and embedded trace buffer (ETB).
- A **neural architecture search** (NAS) algorithm to automate the selection and optimization of machine learning models for the first two tasks.

- A **data recovery** task, where backups of recently accessed files are used to restore encrypted data.
- A post-detection task focuses on **explainable machine learning** analysis, which provides human-interpretable explanations of detection results in both static and dynamic analysis stage.

B. Adversarial Training based Static Analysis

We employ a gradient-boosted decision tree (DT) model as the core static analysis engine. The entire workflow is shown in Figure 4. At the preprocessing stage, we extract a diverse set of static features from each program, covering API calls, opcode sequences, file metadata, Portable Executable (PE) header information, and opcode n -grams that capture recurring instruction patterns. These features provide complementary perspectives on program behavior, enabling the detector to capture both structural and semantic cues. For training, the dataset is split into 80% for model training and 20% for validation. An initial DT baseline is first constructed using the classical CART [17] algorithm and evaluated to identify samples that are misclassified, which serve as valuable “falsified labels” for further refinement.

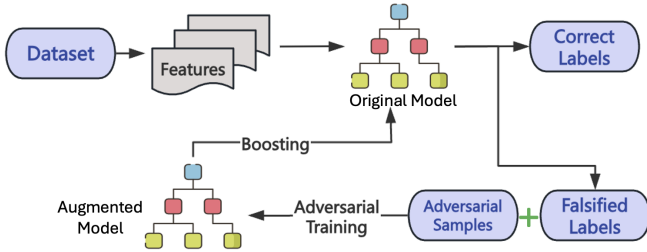


Fig. 4: Workflow of adversarial-training-enhanced static analysis. We synthesize adversarial Samples to emulate evasive/poisoned conditions. The adversarial set is combined with the clean dataset for adversarial Training, yielding *augmented models*. A lightweight *boosting* step integrates all models together, thereby improving the overall robustness.

After obtaining the initial DT model, we extend the framework with an adversarial training process to reinforce its robustness against evasive ransomware. Specifically, in addition to inserting benign code fragments, which simulate padding-based obfuscation, we apply a variety of advanced obfuscation strategies:

- 1) **Opcode substitution:** replacing selected instructions with semantically equivalent alternatives (e.g., swapping instruction sequences that perform the same computation);
- 2) **Control-flow manipulation:** reordering benign code blocks or injecting harmless jumps/NOPs to alter control-flow structure without changing functionality;
- 3) **Polymorphic transformation:** applying lightweight syntactic mutations such as register renaming and insertion of no-op instructions to produce many syntactic variants of the same logic.

These transformations mimic practical obfuscation techniques employed by modern ransomware to evade static signature-based tools. Both the adversarially crafted samples and the previously misclassified instances are fed back into the

training pipeline, ensuring that the model incrementally learns to recognize obfuscated ransomware while maintaining a low false positive rate.

A central challenge of adversarial training is achieving convergence under such adversarial conditions [18]. To overcome this, we adopt a boosting framework [19], where a sequence of lightweight DT models $\mathcal{M} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_N\}$ is progressively trained with adaptive weights $\{w_1, w_2, \dots, w_N\}$. In each iteration, a new augmented tree $\mathcal{T}_i$ is trained specifically on the challenging and adversarial samples. The ensemble prediction is obtained via weighted voting, where higher weights are assigned to trees with stronger performance on the augmented dataset. This iterative scheme steadily enhances the detector’s resilience to obfuscation while keeping the training process stable and computationally efficient. The overall process is summarized in Algorithm 1.

Algorithm 1: Adversarial training of boosting DTs

Input : Dataset($\mathcal{D}$), epochs (k)

Output: Model $\mathcal{M}$

Initialize: $\mathcal{M} = \emptyset, k = 0$

repeat

Split $\mathcal{D}$ into training set $\mathcal{D}_t$ and validation set $\mathcal{D}_v$

$\mathcal{T}_{(k)} = \text{train}(\mathcal{D}_t)$ ▷ Model training

$\mathcal{D}' = \text{validate}(\mathcal{T}_{(k)}, \mathcal{D}_v)$ ▷ Falsified labels

$\hat{\mathcal{D}} = \text{craft}(\mathcal{D})$ ▷ Craft adversarial samples

$\mathcal{T}_{(k+1)} = \text{train}(\mathcal{D}' \cup \hat{\mathcal{D}})$ ▷ Adversarial training

$\mathcal{D} = \mathcal{D} \cup \mathcal{D}' \cup \hat{\mathcal{D}}$ ▷ Dataset augmentation

$\mathcal{M} = \mathcal{M} \cup \{\mathcal{T}_{(k)}\}$

for each $\mathcal{T}_i \in \mathcal{M}$ **do**

$acc_i = 1 - |\text{validate}(\mathcal{T}_{(k)})|/|\mathcal{D}|$

$w_i = \frac{acc_i}{\sum_{i=1}^N acc_i}$ ▷ Weight Adjustment

$k = k + 1$

until $k \geq \text{max_epoch or convergence}$;

Return $\mathcal{M}$

C. Hardware-assisted dynamic analysis

Once a target program is flagged as “suspicious” by the static stage, it is passed to dynamic analysis for further inspection. The key challenge here is that ransomware must be identified and terminated before large amounts of files are encrypted. Relying solely on software-level monitoring often introduces unacceptable delays. To overcome this, we utilize hardware components that can stream low-level execution information in real time, thereby reducing detection latency.

a) HPC features.

The first hardware source we leverage is hardware performance counters (HPCs), and the potential of HPCs for ransomware detection has been explored in [13]. Specifically, HPCs are built-in registers capable of tracking processor events such as instruction counts, cache references, and branch mispredictions. These metrics can be collected using standard interfaces like *perf*. Importantly, many ransomware behaviors manifest distinct hardware footprints. For instance, Figure 5 illustrates the clear gap between a benign program and ransomware (WannaCry) in terms of branch and cache misses.

The frequent encryption loops in WannaCry lead to repeated branching and file access, which manifest as spikes in these HPC signals.

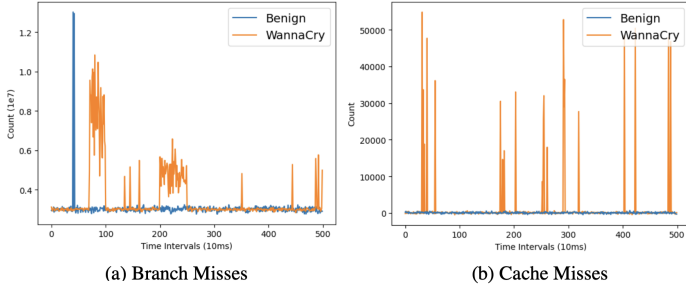


Fig. 5: Comparison of HPC events between the benign program and WannaCry. a) branch misses, b) cache misses.

b) ETB features.

Though HPC features act as a valid way to enable realtime ransomware trace monitoring, they only provide an aggregate view of processor activity, making it unable to pinpoint activities to a particular program. To address this limitation, we also integrate embedded trace buffers (ETBs) into our framework. ETBs capture detailed, cycle-by-cycle on-chip trace for designated process, offering more precise and granular insights than HPCs. However, it comes at the cost of increased overhead, making continuous monitoring with ETBs impractical.

c) Adaptive monitoring.

To combine the efficiency of HPCs with the precision of ETBs, we design an *adaptive monitoring strategy*. Instead of following a fixed alternation schedule between lightweight and heavyweight monitors, our method adaptively determines when to escalate from HPC-based monitoring to ETB confirmation. Specifically, each program execution is divided into time windows (500ms in our experiments). During each window, the system performs five major steps:

- 1) **Initial monitoring with ETBs:** In the initial window, we use ETBs for monitoring, since the program has already been deemed suspicious by the static analysis. The assumption is that it has a high likelihood of being ransomware, so ETBs are employed to detect any malicious behavior at the earliest stage.
- 2) **Risk-guided monitoring with HPCs** If the program passes the first ETB-based window without exhibiting malicious behavior, we assume it may either be benign or a ransomware using delayed infection tactics. To reduce the overhead of continuous ETB monitoring, we switch to HPC-based detection. The HPC classifier would compute the probability p_t indicating how likely the current process exhibits ransomware behavior, alongside two supporting signals:

- **Smoothed risk $\hat{p}_t$:** To avoid reacting to noisy fluctuations, we maintain an exponential moving average $\hat{p}_t = \alpha p_t + (1 - \alpha)\hat{p}_{t-1}$. Intuitively, this acts like a “memory” of recent windows, stabilizing risk estimation in the same way an investor smooths stock prices to spot trends.

- **Slope Δp_t :** We also monitor the change $\Delta p_t = p_t - p_{t-1}$. A sharp rise indicates that suspicious activity is accelerating, which is critical in ransomware detection because encryption routines often intensify abruptly once triggered.

- **Uncertainty u_t :** Finally, we measure how confident the model is in its prediction. We define uncertainty as the predictive entropy $u_t = -(p_t \log p_t + (1 - p_t) \log(1 - p_t))$, which is highest when the model is near 50/50 and therefore unsure. In this context, high uncertainty means we should not fully trust the HPC model and should seek more detailed verification.

- 3) **Escalation:** If $\hat{p}_t$ exceeds a high threshold T_{hi} , or if the slope Δp_t indicates a sudden jump, or if uncertainty u_t is large, then ETB tracing is activated in the next window. This escalation means we switch from coarse monitoring to fine-grained tracing for confirmation.
- 4) **ETB confirmation:** Once escalation is triggered, ETBs are enabled to capture fine-grained traces of the suspicious process. This step acts as a high-confidence verifier. If the ETB classifier confirms ransomware behavior, the process is immediately terminated. Otherwise, we consider that it indicates either the detection result of previous window is a false positive, or the ransomware applies obfuscation to circumvent detection.
- 5) **Rollback and recovery:** Either the ransomware activity was confirmed or a conflict between ETB and HPC happened, we perform data restoration with backup files to erase the possible infection in previous windows (details in Section III-E). This ensures that any partial encryption is reversed to mitigate the ransomware damage.

The workflow is shown in Figure 6. We select 500ms as the window size to efficiently monitor ransomware activities while keeping resource usage low. In order to handle time-sequential data dumped from HPCs/ETBs, we apply long-short term memory (LSTM) as our machine learning model for classification. The details of the model structure and hyperparameter tuning will be discussed in the next section.

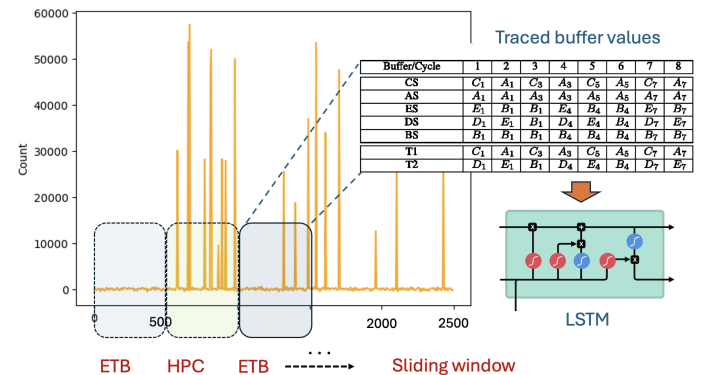


Fig. 6: Adaptive monitoring workflow combining HPC and ETB. An LSTM model is applied to classify the sequential hardware traces.

D. Neural Architecture Search

In this section, we elaborate on how we determine the structure and parameters for the applied LSTM model. As illustrated in Section II, another key challenge in ML-based ransomware detection is the limited generalizability of models across different variants. We attribute this challenge to the fact that in most existing approaches, model architectures were manually selected based on prior knowledge, which lacks the flexibility to adapt to unseen variants. A straightforward approach could be to exploit a large-scale model. However, such models introduce high latency and are more prone to overfitting [20], which is undesirable in a latency-sensitive application such as ransomware detection.

To address this issue, we employ Neural Architecture Search (NAS) to automatically discover optimal model architectures tailored for our task. NAS explores a large space of potential network configurations, evaluating candidate architectures with different layers and connections, and gradually converges to structures that achieve high task-specific performance [21].

Our framework employs a one-shot NAS paradigm for better efficiency. Instead of training each candidate architecture individually, we first construct a large, over-parameterized *supernet* that contains many possible sub-architectures. Then we progressively eliminate redundant components while retaining helpful ones, resulting in a final optimized model architecture that is both efficient and effective. The detailed workflow proceeds in three stages:

- 1) **Supernet training.** We first construct an over-parameterized supernet representing the entire search space, including LSTM units, convolutional and fully connected layers, pooling operations, and activation functions. The supernet is trained end-to-end for the detection task, so that all components inherit useful parameters.
- 2) **Subnet evaluation.** After supernet training, we sample a population of candidate subnets $a \in \mathcal{A}$. For each a , we measure its accuracy $\text{Acc}(a)$ and latency $\text{Lat}(a)$, and compute the composite loss as

$$\mathcal{L}_{\text{NAS}}(a) = \lambda(1 - \text{Acc}(a)) + (1 - \lambda) \frac{\text{Lat}(a)}{\max_j \text{Lat}(a_j)}.$$

Next, to relate these losses to architectural choices, we encode each sampled architecture as a binary vector over its m selected components, which gives us a matrix $X \in \{0, 1\}^{P \times m}$ for P samples. We then approximate the observed losses

$$L = [\mathcal{L}_{\text{NAS}}(a_1), \dots, \mathcal{L}_{\text{NAS}}(a_P)]^\top$$

as a linear combination of the selected components, with coefficient vector $\beta = \{\beta_1, \beta_2, \dots, \beta_m\}$ representing the contribution of each component. To encourage compactness and prevent overfitting, we further impose an ℓ_1 penalty, leading to the optimization problem:

$$\beta = \arg \min_{\beta} \left(\|L - X\beta\|_2^2 + \gamma \|\beta\|_1 \right),$$

where γ controls the sparsity of the solution. By solving this optimization problem, we obtain β as the evaluation

of every component involved in this iteration, which helps us to refine the architecture.

- 3) **Pruning.** Components with large positive coefficients, indicating their inclusion to increase the composite loss, are pruned from the search space using a threshold τ set relative to the distribution of coefficients. The supernet is then retrained on the reduced space, and this process repeats until convergence.

An illustration of this NAS pruning process is shown in Figure 7. We start with a dense supernet, where redundant nodes are progressively removed, resulting in a streamlined architecture optimized for both effectiveness and efficiency. The entire workflow incorporating all three major steps is presented in Algorithm 2.

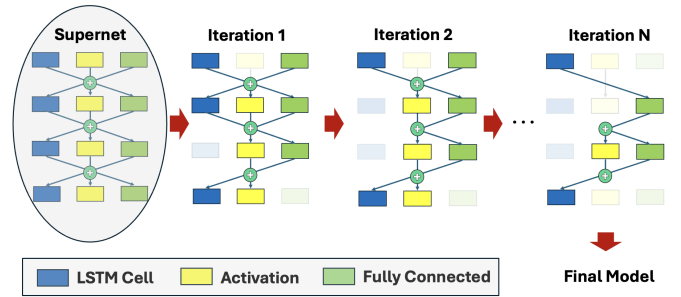


Fig. 7: The proposed NAS process: Starting from an over-parameterized supernet, candidate sub-architectures are sampled and evaluated with the composite loss. Then the components (cells) associated with higher loss contributions are pruned (shown faded), the process continues until a compact final model is obtained.

Algorithm 2: NAS with composite loss and sparse regression pruning

Input : Dataset $\mathcal{D}$, rounds K , trade-off λ , regularization γ , threshold τ , population P

Output: Optimized architecture $\mathcal{M}$

Initialize supernet $\mathcal{S}$; $r = 0$

repeat

 Train $\mathcal{S}$ on $\mathcal{D}$ (weight sharing)

 Sample P candidate architectures $\{a_1, \dots, a_P\}$

 For each a_i : compute $\text{Acc}(a_i)$, $\text{Lat}(a_i)$

$\tilde{L}(a_i) = \text{Lat}(a_i) / \max_j \text{Lat}(a_j)$

$L_i = \lambda(1 - \text{Acc}(a_i)) + (1 - \lambda)\tilde{L}(a_i)$

 Build design matrix $X \in \{0, 1\}^{P \times m}$

$\beta = \arg \min_{\beta} \|L - X\beta\|_2^2 + \gamma \|\beta\|_1$

 Prune components with $\beta_i \geq \tau$

 Update $\mathcal{S}$; $r = r + 1$

until $r \geq K$ or convergence;

 Select $\mathcal{M}$ with minimum L_i

return $\mathcal{M}$

E. Data Recovery

Although our proposed model allows for real-time detection and significantly reduces latency, some files may still

inevitably get encrypted by ransomware before the malicious activity is flagged. To mitigate this impact, we extend our framework with a dynamic backup and recovery framework that runs in parallel with the monitoring process. This module complements detection by providing a rapid rollback capability, thereby completing the end-to-end defense cycle. The entire framework consists of three major modules.

a) Lightweight snapshotting.

The first module adopts a rolling snapshot approach to keep track of file changes. During program execution, we run a shell script using `rsync` command to save lightweight copies of the modified files inside the focused directory. Notice that instead of maintaining full backups of entire directories, this script executes at one-minute intervals and captures only those files that have been accessed or modified during the latest window.

b) Integration with detection.

The snapshotting process runs in parallel with the dynamic analysis module (Section III-C). In each time window, a temporary backup is created using lightweight snapshotting for files that were accessed. If the LSTM detector concludes that the program behavior is benign, the snapshot is discarded to conserve resources. Otherwise, as the detector raises an alert indicating ransomware activity, the program is terminated and the snapshot is used to roll back any affected files. This parallel design keeps backups continuously available, enabling rapid recovery with minimal latency.

c) File event logging.

To make the framework more practical for real-world use, we incorporate an `inotify`-based logger that continuously records file system events (e.g., *open*, *modify*, *create*, *delete*) within the monitored directories. This logging helps confirm that snapshots and recoveries happen as intended, and it also keeps a detailed record of file activity. Such records allow administrators to verify what happened to each file and restore the sequence of actions if any incident happened.

In summary, the proposed data recovery strategy complements our detection pipeline, it minimizes storage overhead while ensuring effective file protection.

F. Explainable Machine Learning

Another major limitation of existing methods is the lack of interpretability. Due to the black-box nature of ML models, no additional information beyond the detection result is available. In such cases, users have no guidance on how to respond to misclassifications. Moreover, a binary label alone may be insufficient for follow-up tasks such as ransomware localization. To address this limitation, we integrate explainable machine learning into both static and dynamic analysis.

For static analysis, we adopt SHAP [22] to enhance interpretability. SHAP originates from cooperative game theory, where the goal is to quantify the marginal contribution of each player to the overall outcome. Formally, given a feature set N , a subset $S \subseteq N$ excluding feature i , and model function $f(\cdot)$, the SHAP value of feature i is defined as

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} [f(S \cup \{i\}) - f(S)]. \quad (1)$$

In our context, ϕ_i represents the average contribution of a static feature (e.g., an API call or opcode sequence) to the classification result. We employ SHAP for both misclassification handling and outcome interpretation. For each flagged sample, we report the top- k features with the highest $|\phi_i|$, which helps explain underlying ransomware patterns such as API-based triggers. In addition, features with consistently low contributions during training phase would be pruned, reducing noise and improving the performance of the detection model.

As for dynamic analysis, the model architecture is no longer fixed due to the utilization of NAS, and the input consists of sequential HPC/ETB traces with variable length. Therefore, we require a model-agnostic method that is also compatible with sequential data. We employ LIME [23], which addresses this by fitting a local linear regression surrogate to approximate the detector's decision based on input traces. Formally, for an input (hardware traces from HPC/ETB) $x = [x_1, x_2, \dots, x_T]$ within T time slots, the surrogate prediction is given by

$$\hat{y} = w_0 + \sum_{t=1}^T w_t x_t, \quad (2)$$

where w_t represents the contribution of time slot t . In this way, the absolute values of w_t actually indicate the relative importance of slots, allowing us to rank them and interpret the results in the context of ransomware detection. For example, clusters of important slots suggest concentrated malicious activity within specific intervals, while periodically separated slots may reflect recurring actions such as client-server communication. As a result, our detection model could offer localized insights into the trace patterns beyond the detection result, which will be demonstrated in Section IV-G.

IV. EXPERIMENTAL EVALUATION

In this section, we demonstrate the effectiveness of our proposed framework. We first outline the experimental setup. Next, we present the performance of the proposed framework.

A. Experiment Setup

Benchmark: We evaluate our framework using the publicly available *Ransomware Combined Structural Feature Dataset*, which contains structural features extracted from executable binaries collected and preprocessed. The dataset comprises a total of 2,675 samples, partitioned into a training/validation split of 2,157 samples (80%) and a test set of 518 samples (20%). The training portion comprises 1,023 ransomware samples spanning 25 families and 1,134 goodwill samples, while the test portion contains 385 ransomware samples and 133 goodwill samples. The dataset provides multiple feature sets (PE headers, imported DLLs, API calls, section entropy, and opcode n -grams), as well as explicit train/test splits that we adopt in our experiments.

The original dataset contains a large number of distinct families (40+ family IDs available across versions). To keep the evaluation results concise, we organize the dataset using a two-tier scheme:

- 1) **Category grouping** (main evaluation). We group related families into four high-level categories: *Crypto-Ransomware*, *Locker Ransomware*, *Scareware/FakeAV*, and *Miscellaneous/Hybrid*. For the main accuracy and latency tables, we report aggregated metrics per category (mean and standard deviation across families within the same category).
- 2) **Representative variants** (detailed studies). To keep the experimental evaluation compact while still enabling a fine-grained analysis, we select representative sample variants per category for detailed case studies. These selected representative families (i) are widely studied in the literature and (ii) reflect diverse behavioral patterns (for example, *WannaCry* (Crypto), *Reveton* (Locker), *FakeAV* (Scareware), and *Phobos/Stop* (Hybrid)).

TABLE I: Ransomware categories and representative families.

Category	Example Families
Crypto-Ransomware	<i>WannaCry</i> , <i>Locky</i> , <i>Ryuk</i> , <i>Maze</i> , <i>Petya</i>
Locker Ransomware	<i>Reveton</i> , <i>WinLock</i>
Scareware/FakeAV	<i>FakeAV</i> , <i>Rogue AV variants</i>
Misc./Hybrid	<i>DoppelPaymer</i> , <i>BlackCat</i> , <i>Phobos</i>

To further evaluate robustness against obfuscation, we extend the current dataset with augmented adversarial samples. For example, one strategy we applied is to inject non-functional and benign instructions into ransomware binaries to produce obfuscated variants (opcode substitution, NOP insertion, register renaming, and lightweight polymorphic edits). These adversarially-obfuscated samples are mixed into the training and test splits according to the protocol described in Section IV-C.

Implementation Packet: The proposed framework was implemented and evaluated on a Linux host equipped with an Intel Core i9-10900K processor (3.70 GHz), 64 GB RAM, and an NVIDIA RTX 3080 GPU. The operating system was Ubuntu 22.04 LTS with Linux kernel 6.16, and hardware performance counter (HPC) events were collected using the `perf` interface. For runtime monitoring, HPC features were collected at a fixed sampling interval of 500 ms. To improve process-level attribution and reduce cross-process interference, the monitored program was pinned to a dedicated CPU core using CPU affinity, and per-window counter deltas were aggregated into time-series feature vectors for the LSTM classifier. The fine-grained execution traces were collected using Intel Processor Trace (PT) and parsed into sequential trace representations aligned with the same 500 ms monitoring windows. Trace collection was enabled selectively by the adaptive monitoring policy described in Section III-C to limit runtime overhead.

As for the machine learning components, the entire model framework was implemented in PyTorch. The static classifier was trained for 200 epochs, and the LSTM model for dynamic analysis was trained for 500 epochs, using an initial learning rate of 0.05, which was increased to 0.1 after 200 epochs. A dropout rate of 0.5 was applied to both models, and cross-validation was used to reduce evaluation bias.

Approaches: To enable fair comparison, we compare the performance and efficiency of the following four approaches.

- **DRL:** A deep reinforcement learning based static analysis method [8].
- **Ratafia:** A dynamic analysis method using HPC values informed auto-encoders [13].
- **Rwarmor:** A hybrid detection framework using static-informed dynamic analysis [24].
- **Proposed:** The proposed hardware-assisted multistage detection method.

Evaluation: We consider the *Accuracy*, *Recall*, *Precision*, and *F-1* Scores as the metrics for evaluation.

B. Case Study: Detection Accuracy

Table II records the detection performances using the above-mentioned metrics. We compare three representative baselines (DRL, Ratafia, and Rwarmor) with our proposed method across various ransomware categories: Crypto, Locker, Scareware/FakeAV, and Hybrid. The final row shows the averages over all categories.

Overall, DRL reaches 78.4% accuracy, Ratafia achieves 83.2%, and Rwarmor falls behind at 76.1%. Our proposed method significantly outperforms these approaches, with 91.9% accuracy on average and improvements of 8–12% across the different categories. Precision and recall follow the same trend: while DRL and Rwarmor often fall below 80%, our method maintains values above 91% in all categories, leading to the highest F1 scores. This indicates that our framework achieves both low false positives and low false negatives consistently.

The relative performance trends are consistent with the characteristics of each detection method. DRL is a static analysis-based method that relies solely on static features. Consequently, it struggles to differentiate between benign encryption programs and ransomware, causing the highest number of false alarms, resulting in low precision. Ratafia leverages dynamic monitoring of HPC events but lacks complementary ETB information, which results in higher FNRs when ransomware families exhibit subtle or obfuscated runtime behaviors, lowering its recall. Rwarmor also adopts dynamic analysis, but its feature recognition algorithm depends heavily on communication patterns, which became fragile in front of categories such as Scareware and Hybrid, where behaviors are less uniform. For instance, in Scareware/FakeAV, Rwarmor’s accuracy drops to 70.8%, demonstrating the difficulty of detecting ransomware that primarily relies on deceptive user-facing alerts rather than heavy encryption workloads. In contrast, our method integrates both HPC and ETB features and leverages NAS to optimize the detection model. This combination captures both coarse-grained runtime patterns and fine-grained instruction-level traces to ensure robust precision and recall. This improvement is especially significant when detecting Crypto and Scareware, where the proposed approach achieves over 93% accuracy compared to 85.2% and 72.1% for DRL, respectively. These results confirm that the proposed method generalizes well across diverse ransomware behaviors and addresses the weaknesses of existing techniques.

TABLE II: Detection performance across ransomware categories.

Category	DRL				Ratafia				Rwarmor				Proposed				
	Acc	Prec	Rec	F-1	Acc	Prec	Rec	F-1	Acc	Prec	Rec	F-1	Acc	Prec	Rec	F-1	Improv(%)
Crypto	85.2	82.1	86.4	0.87	88.9	87.5	89.2	0.92	81.3	78.6	82.5	0.88	93.8	94.5	93.2	0.96	8.7
Locker	76.5	74.2	77.8	0.82	80.2	81.0	79.6	0.88	74.9	73.4	75.5	0.84	91.7	92.1	91.4	0.95	11.5
Scareware	72.1	70.8	73.2	0.80	79.4	80.2	78.6	0.86	70.8	71.0	70.6	0.82	89.9	90.3	89.6	0.94	12.3
Hybrid/Misc	79.6	77.1	80.5	0.84	84.3	85.0	83.7	0.90	77.5	76.9	77.8	0.86	92.3	92.9	91.8	0.95	10.9
Average	78.4	76.0	79.5	0.83	83.2	83.4	82.8	0.89	76.1	75.0	76.6	0.85	91.9	92.5	91.5	0.95	10.8

C. Case study: Adversarial Robustness

Robustness is central to ransomware detection, as real attacks routinely use obfuscations to evade conventional signature-based detection. To evaluate the robustness of selected methods, we generate adversarial variants that perturb with the following five strategies:

- *NOP*: insert short runs of no-ops or benign ops to pad code without changing control flow.
- *Opcode substitution*: swap instructions with semantics-equivalent forms to alter tokens and n -grams.
- *Sleep*: inject brief sleeps between phases to create idle windows in the trace.
- *API reordering*: permute independent API calls within a stage to disrupt sequence patterns.
- *Packing*: wrap binaries with a simple packer for polymorphism.

Table III compares the clean and adversarial accuracy for all four methods. As we can see, DRL shows the steepest decline under obfuscation, falling from 78.4% to 59.9% on average and dropping below 60% on Scareware, which is no better than a random guess. Rwarmor exhibits a similar pattern, with adversarial accuracy near 59% on Scareware, while Ratafia holds up better overall but still slips to mid 60% on Locker when sleeps and API reorderings disturb its HPC signatures. These outcomes align with their underlying properties: DRL focuses on static signatures which can be easily perturbed; Rwarmor relies on sequence regularities which are fragile against reordering; and Ratafia lacks ETB traces to anchor timing shifts. In contrast, our approach maintains 88.3% on adversarial samples (-4.1% from clean), with an average 22.6% improvement over baselines, and achieved the largest gap on Scareware detection(34.4%).

Figure 8 breaks down performance by adversarial strategies. Among the baselines, DRL is hit hardest by opcode substitution and packing, where tokens and structure-level signatures are muted; in both cases its accuracy falls below 50%. Ratafia degrades most with sleep/idle frames insertion, as idle gaps blur the timing patterns in HPC traces, leading to accuracy down to mid-60%. Rwarmor drops significantly under API reordering, where the control-flow features become messy and less informative, resulting in an accuracy around 59%. In contrast, our approach remains steady across all five strategies, typically between 86% and 90%. We attribute this advantage to two design choices. First, we adopt an adversarial training step for the static analysis, enhancing its robustness against such types of obfuscation during the training phase. Second, ETB-based tracing complements HPC-based tracing by providing fine-grained context. In this case, even obfuscation perturbs

aggregate HPC statistics (e.g., idle windows or delayed activation), the underlying malicious activity with characteristic ETB patterns can still be captured in subsequent windows.

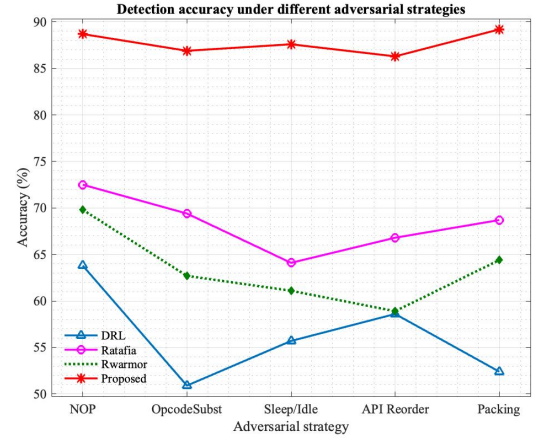


Fig. 8: The detection accuracy against adversarial strategies.

D. Case study: Detection Latency

Another critical performance metric is detection latency, which is the time elapsed from the initiation of the malicious behavior of ransomware to its detection. This latency is directly proportional to the number of files affected before discovery. Table IV records the average detection latency across all tested methods. As indicated, our approach outperforms the others. Specifically, DRL exhibits the longest latency as it involves heavy computational overhead required by its reinforcement learning framework. Ratafia demonstrates slightly better efficiency, while our method achieves the lowest latency. This improvement is attributed to the sliding-window monitoring with hardware assistance.

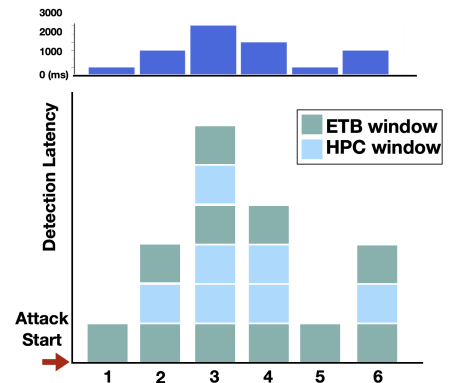


Fig. 9: Detection latency of six ransomware samples under sliding-window dynamic analysis with HPC and ETB.

TABLE III: Robustness to adversarial obfuscation, with improvements of the proposed method over baselines.

Category	DRL		Ratafia		Rwarmor		Proposed		Improvement (Proposed – Baseline)		
	Clean	Adv	Clean	Adv	Clean	Adv	Clean	Adv	vs DRL	vs Ratafia	vs Rwarmor
Crypto	85.2	65.3	88.9	74.2	81.3	70.5	93.8	89.6	+24.3	+15.4	+19.1
Locker	76.5	58.4	80.2	68.7	74.9	63.8	91.7	87.9	+29.5	+19.2	+24.1
Scareware/FAV	72.1	52.0	79.4	66.5	70.8	59.7	89.9	85.7	+33.7	+19.2	+26.0
Hybrid/Misc	79.6	62.7	84.3	73.1	77.5	66.9	92.3	88.1	+25.4	+15.0	+21.2
Average	78.4	59.6	83.2	70.6	76.1	65.2	91.9	87.8	+28.2	+17.2	+22.6

To illustrate, Figure 9 shows an in-depth inspection of six sample detection processes. In the lower diagram, the window-based monitoring progression is depicted, while the upper histogram tracks the total detection time for each sample. For example, the detector raises an alarm in the first window via initial ETB confirmation for sample 1 and 5. While for variants with delayed activation, such as sample 3, the system still identifies the threat within a small number of windows using the combined HPC–ETB pipeline as described in Section III-C.

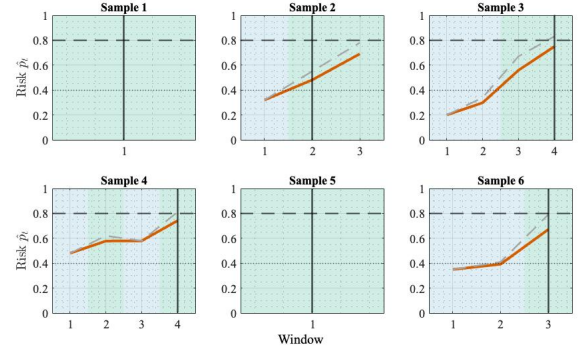
TABLE IV: The comparison of detection latency.

Methods	DRL	Ratafia	Rwarmor	Proposed	Improv
Delay(ms)	49984	13480	29670	2784	4.8x

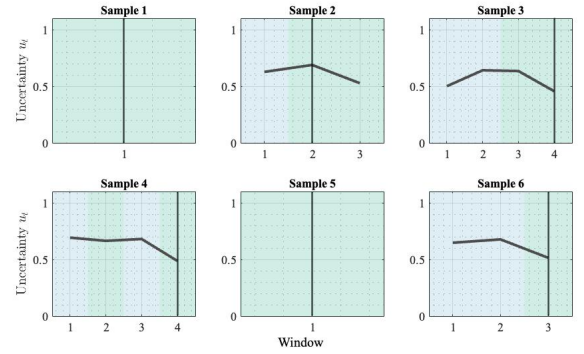
The reason behind these immediate responses is our risk-guided monitoring. Figure 10 plots the smoothed risk $\hat{p}_t$ and predictive uncertainty u_t across the monitoring process for these samples. As shown, the risks for samples 1 and 5 exceed the threshold (set to 0.8 in our experiment) in the first window with low uncertainty, leading to immediate alarms. Sample 3 exhibits a delayed start: risk stays modest for two windows, then rises quickly, triggering a short ETB burst that exposes malicious activity; the alarm follows in the next window. Sample 4 shows a brief escalation and de-escalation: ETB is triggered once, but no malicious activity is confirmed, and the monitor returns to HPC without thrashing, thereby avoiding extra overhead. Overall, the proposed method delivers a fast and accurate response to ransomware activities.

E. Case Study: Ablation Test of Dynamic Monitoring

We further conducted an ablation study to isolate the impact of different dynamic monitoring strategies, including an HPC-only variant, a PT-only variant, and the proposed adaptive monitoring policy that combines HPC with selective PT escalation. The results are summarized in Table V. Overall, the HPC-only variant exhibits the lowest detection accuracy, with an average drop of 17.5 percentage points compared to the adaptive policy, and incurs substantially higher detection latency. This behavior is particularly obvious for the *Locker* and *Scareware/FAV* categories, which are more staggered and delayed, causing the coarse-grained HPC-based detector to wait for additional window frames before reaching high confidence. In contrast, the ETB-only variant achieves higher accuracy and the lowest detection latency by leveraging fine-grained execution traces, but at the cost of significantly increased runtime memory overhead due to continuous fine-grained tracing, making it expensive to maintain in practice. Instead, the proposed adaptive policy effectively balances this trade-off:



(a) Smoothed risk $\hat{p}_t$ over 500 ms windows for six samples. Shading denotes monitoring mode (HPC vs. ETB); the vertical line marks the alarm window.



(b) Predictive uncertainty u_t for the same samples, aligned with (a).

Fig. 10: Risk and uncertainty traces aligned with the six detection processes in Fig. 9.

it attains detection accuracy close to the PT-only configuration while maintaining latency comparable to, or lower than, the HPC-only baseline, which highlights the effectiveness of the risk-guided escalation and early-exit mechanism. As we can see, by escalating to ETB tracing only when needed, the adaptive framework reduces unnecessary cost for fine-grained monitoring while preserving fast detection in practice.

F. Case study: Generalizability

In this section, we take an in-depth inspection of how our proposed NAS-guided architecture addressed the generalizability problem in ransomware detection. As previously presented in Figure 2 and Table II, existing detection approaches yield unstable performance across different ransomware variants. To understand the reason, we plot HPC feature values of several selected ransomware samples in 3-D space as presented in Figure 11. As we can see from the scatter plot, a distinct observation is that different ransomware possess dif-

TABLE V: Ablation of dynamic monitoring strategies: adversarial detection accuracy and detection latency (ms) by category.

Category	HPC-only Acc	HPC-only Lat.	ETB-only Acc	ETB-only Lat.	Adaptive Acc	Adaptive Lat.	Acc gain (pp)	Latency gap
Crypto	74.6	5600	90.8	2680	89.1	3900	14.5	1220
Locker	65.8	6200	88.9	2850	87.4	4200	21.6	1350
Scareware/FAV	68.9	6400	87.2	3000	86.3	4500	17.4	1500
Hybrid/Misc	73.7	5800	91.5	2600	90.4	4000	16.7	1400
Average	70.8	6000	89.6	2784	88.3	4150	17.5	1367.5

ferent feature clusters. In other words, there exist significant, distinct feature patterns for different ransomware variants. For example, as previously mentioned in Section IV-B, Vipasan possesses completely different patterns in the feature space, with low branch misses and low-level cache references due to the lack of communication with the control server. As a result, the designed ML model must possess flexibility so that it can capture various types of feature patterns. But it would be very hard to design such a machine learning model architecture manually, so that it could adapt to cover all variants.

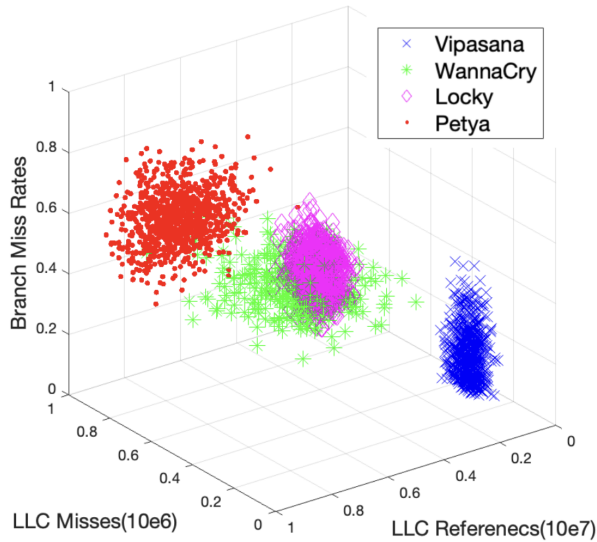


Fig. 11: The scatter plot of selected HPC values for various benchmarks.

Our proposed method applies neural architecture search (NAS) to address this challenge. Figure 12 depicts the simplified outline of our model generated from NAS. We interpret the workflow of this generated model in this way: after receiving the input values, blocks 0-2 are immediately utilized. These blocks serve as a “classifier” to firstly determine what type of variant it is, then different subsequent blocks and layers are applied after the “classification” step, so that different modules can be assigned to capture different types of feature patterns subsequently, therefore successfully perform detection across various ransomware variants. Though the rationale is straightforward and we could also manually design such an architecture, but it requires huge amount of tries and prior-knowledge. In our case, this architecture is generated automatically without costing any extra effort.

G. Case Study: Explainable Machine Learning Analysis

In this section, we perform explainable machine learning analysis on the proposed method’s outcome, to provide both

feature-level and trace-level evidence for each model decision, increasing its transparency.

First, as illustrated in Section III-F, we perform SHAP analysis on static analysis results. Figure 13 shows waterfall plots of SHAP values for two representative samples: (a) a true positive Crypto case and (b) a true negative benign encryption program. We plot contributions on the model’s features, with the decision boundary at zero; red bars push the score toward ransomware, blue bars push it toward benign. Only the top eight features are shown for readability. In Figure 13 (a), the decision is dominated by *Crypto-constant score*, while *Import Breadth* and *n-grams opcode* provide side support. A few features do push back, but their magnitudes are modest, so the model’s final output ($f(x)$) lands well above the boundary, consistent with the detection outcome of this Crypto-type ransomware sample.

Instead, Figure 13 (b) illustrates the opposite outcome for a benign program that legitimately uses cryptography. Although a crypto signal is present, the other stronger features dominate. A broad, regular *import breadth* and a benign *n-gram opcode* profile contribute the largest negative shifts. Low reliance on *dynamic-resolve* and only moderate *section entropy* provide additional evidence, and the *valid code signature* reinforces the benign prior, leaving the final score well below the decision boundary.

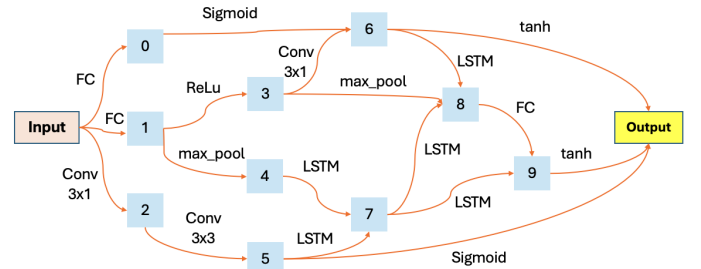
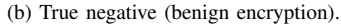
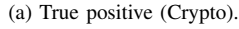


Fig. 12: A simplified demonstration of our LSTM model generated from our neural architecture search (NAS) algorithm.

Collectively, these two examples show that our proposed framework does more than return a label at the static stage. It also reports per-feature contributions, making the decision path explicit. Analysts can verify whether the dominant signals align with the sample’s nature (e.g., crypto intent for encryptors) for more accurate detection. Meanwhile, this breakdown can also be used to debug why an alarm was raised and to decide whether proceed to dynamic monitoring stage.

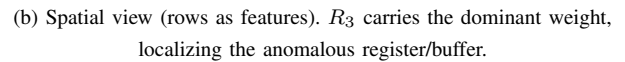
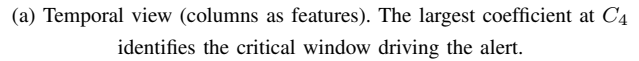
As for dynamic monitoring, we complement the explainability view with a local, LIME-based analysis over the runtime trace table. Specifically, we record the register/buffer values captured by HPC/ETB during a focused window, and fit a



lightweight linear surrogate, where the resulting coefficients could be viewed as attributions.

Together, these views provide human-understandable evidence to interpret the detection process. In this way, an analyst can verify the correctness of detection and trace back to the raw ETB/HPC signals for ransomware localization. The temporal-spatial attributions also help separate brief obfuscation from sustained malicious activity, reducing false positives and guiding when to terminate the target program.

To quantify the computational overhead of the proposed framework, we measure both offline and online costs under a controlled execution environment. Offline costs correspond to one-time expenses incurred during model preparation, including model training and NAS. Online overhead is measured during deployment and reported on a per-window basis, where



each monitoring window spans 500 ms of program execution. For online measurements, we instrument the runtime monitoring pipeline and record the wall-clock time consumed by each component. All runtime measurements were conducted on an otherwise idle system to minimize interference from background processes.

TABLE VI: Cost breakdown of the proposed framework. Offline costs are one-time expenses for training and NAS, while online overheads are measured per monitoring window.

Phase	Component	Avg. time	Worst-case
Offline	Static model training (DT)	2.1 h	–
	Dynamic model training (LSTM)	12.6 h	–
	Neural architecture search (NAS)	5.2 h	–
Online	HPC monitoring	0.6 ms	0.8 ms
	ETB tracing	2.9 ms	3.5 ms
	Decision & aggregation	0.2 ms	0.3 ms
Summary	End-to-end (avg.)	1.1 ms	–
	End-to-end (worst)	–	4.6 ms

V. CONCLUSION

Ransomware has become a significant threat to cybersecurity in recent years. Existing ransomware detection methods suffer from four major limitations: vulnerability to obfuscation, detection latency, and low generalizability. To address these challenges, this paper presents a dynamic hardware-assisted ransomware detection framework that leverages adaptive runtime monitoring to balance detection accuracy and efficiency. Experimental results demonstrate that the proposed approach achieves consistent improvements in detection accuracy and reduced detection latency compared to prior works. Additionally, by incorporating adversarial training, we strengthen static analysis against obfuscation techniques, and the utilization of one-shot Neural Architecture Search (NAS) automates the optimization of model architectures, enabling the framework's flexibility across ransomware variants.

REFERENCES

- [1] T. McIntosh, A. Kayes, Y.-P. P. Chen, A. Ng, and P. Watters, "Ransomware mitigation in the modern era: A comprehensive review, research challenges, and future directions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–36, 2021.
- [2] M. N. Olaimat *et al.*, "Ransomware anti-analysis and evasion techniques: A survey and research directions," in *2021 3rd international cyber resilience conference (CRC)*. IEEE, 2021, pp. 1–6.
- [3] A. Arabo, R. Djouj, T. Poulain, and G. Chevalier, "Detecting ransomware using process behavior analysis," *Procedia Computer Science*, vol. 168, pp. 289–296, 2020.
- [4] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, "Cryptoransomware early detection model using novel incremental bagging with enhanced semi-random subspace selection," *Future Generation Computer Systems*, vol. 101, pp. 476–491, 2019.
- [5] A. Cuzzocrea, F. Martinelli, and F. Mercaldo, "A novel structural-entropy-based classification technique for supporting android ransomware detection and analysis," in *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE, 2018, pp. 1–7.
- [6] H. Faris, M. Habib, I. Almomani, M. Eshtay, and I. Aljarah, "Optimizing extreme learning machines using chains of salps for efficient android ransomware detection," *Applied Sciences*, vol. 10, no. 11, p. 3706, 2020.
- [7] Y. Feng, C. Liu, and B. Liu, "Poster: A new approach to detecting ransomware with deception," in *38th IEEE symposium on security and privacy*, 2017.
- [8] X. Deng, M. Cen, M. Jiang, and M. Lu, "Ransomware early detection using deep reinforcement learning on portable executable header," *Cluster Computing*, vol. 27, no. 2, pp. 1867–1881, 2024.
- [9] A. Alzahrani, A. Alshehri, H. Alshahrani, R. Alharthi, H. Fu, A. Liu, and Y. Zhu, "Randroid: Structural similarity approach for detecting ransomware applications in android platform," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0892–0897.
- [10] A. Cimitile, F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Talos: no more ransomware victims with formal methods," *International Journal of Information Security*, vol. 17, pp. 719–738, 2018.
- [11] A. Karimi and M. H. Moattar, "Android ransomware detection using reduced opcode sequence and image similarity," in *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2017, pp. 229–234.
- [12] S. Lee, S. Lee, J. Park, K. Kim, and K. Lee, "Hiding in the crowd: ransomware protection by adopting camouflage and hiding strategy with the link file," *IEEE Access*, 2023.
- [13] M. Alam *et al.*, "Ratafia: Ransomware analysis using time and frequency informed autoencoders," in *2019 IEEE International Symposium on Hardware Oriented Security and Trust*. IEEE, 2019, pp. 218–227.
- [14] M. T. Nguyen, V. H. Nguyen, and N. Shone, "Using deep graph learning to improve dynamic analysis-based malware detection in pe files," *Journal of Computer Virology and Hacking Techniques*, vol. 20, no. 1, pp. 153–172, 2024.
- [15] S. Gulmez, A. G. Kakisim, and I. Sogukpinar, "Xran: Explainable deep learning-based ransomware detection using dynamic analysis," *Computers & Security*, vol. 139, p. 103703, 2024.
- [16] J. A. Herrera-Silva and M. Hernández-Álvarez, "Dynamic feature dataset for ransomware detection using machine learning algorithms," *Sensors*, vol. 23, no. 3, p. 1053, 2023.
- [17] R. J. Lewis, "An introduction to classification and regression tree (cart) analysis," in *SAEM*, vol. 14, 2000.
- [18] Y. Wang, X. Ma, J. Bailey, J. Yi, B. Zhou, and Q. Gu, "On the convergence and robustness of adversarial training," *arXiv preprint arXiv:2112.08304*, 2021.
- [19] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.
- [20] X. Ying, "An overview of overfitting and its solutions," in *Journal of physics: Conference series*, vol. 1168. IOP Publishing, 2019, p. 022022.
- [21] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019.
- [22] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.
- [23] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you?: Explaining the predictions of any classifier," in *International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 1135–1144.
- [24] M. A. Ayub, A. Siraj, B. Filar, and M. Gupta, "Rwarmor: a static-informed dynamic analysis approach for early detection of cryptographic windows ransomware," *International Journal of Information Security*, vol. 23, no. 1, pp. 533–556, 2024.

AUTHORS' BIOGRAPHIES

Salma Aboelmagd (Graduate Student Member, IEEE) received the B.Sc. in Electrical and Computer Engineering from Texas A&M University in 2020 and the M.Sc. in Data Science and Engineering from Hamad Bin Khalifa University in 2022. She is currently pursuing the Ph.D. degree in Electrical Engineering at Florida State University.

Md. Rakibul Ahasan (Graduate Student Member, IEEE) received the B.S. degree from Rajshahi University of Engineering and Technology, Bangladesh, in 2011, the M.S. degree from BRAC University, Bangladesh, in 2022, and another M.S. degree from Miami University, Ohio, in 2024. He is currently pursuing the Ph.D. degree in Electrical Engineering at Florida State University under Dr. Abdulrahman Takiddin.

Zhixin Pan (Member, IEEE) received the Ph.D. degree in Electrical and Computer Engineering from the University of Florida in 2022. He is currently an Assistant Professor in the Department of Electrical and Computer Engineering at Florida State University. His research interests include hardware security, explainable artificial intelligence, and efficient machine learning systems.

Abdulrahman Takiddin (Member, IEEE) received the Ph.D. degree from Texas A&M University in 2023. He is an Assistant Professor in the Department of Electrical and Computer Engineering at the FAMU–FSU College of Engineering, Florida State University. His research interests include machine learning, cyber-physical systems, smart grids, smart transportation, and security.

Ziyu Shu (Member, IEEE) is a Postdoctoral Researcher in the Department of Radiation Oncology at Washington University in St. Louis. He received the Ph.D. degree in Computer Science from the University of Florida in 2023. His research interests include machine learning, medical imaging, and computer vision.