SUNDEW: An Ensemble of Predictors for Case-Sensitive Detection of Malware

Sareena Karapoola, Nikhliesh Singh, Chester Rebeiro, and Kamakoti V

Abstract—Malware programs are diverse, with varying objectives, functionalities, and threat levels ranging from mere pop-ups to financial losses. Consequently, their run-time footprints across the system differ, impacting the optimal data source (Network, Operating system (OS), Hardware) and features that are instrumental to malware detection. Further, the variations in threat levels of malware classes affect the user requirements for detection. Thus, the optimal tuple of ⟨`data-source`, `features`, `user-requirements`⟩ is different for each malware class, impacting the state-of-the-art detection solutions that are agnostic to these subtle differences.

This paper presents SUNDEW, a framework to detect malware classes using their optimal tuple of ⟨`data-source`, `features`, `user-requirements`⟩. SUNDEW uses an ensemble of specialized predictors, each trained with a particular data source (network, OS, and hardware) and tuned for features and requirements of a specific class. While the specialized ensemble with a holistic view across the system improves detection, aggregating the independent conflicting inferences from the different predictors is challenging. SUNDEW resolves such conflicts with a hierarchical aggregation considering the threat-level, noise in the data sources, and prior domain knowledge. We evaluate SUNDEW on a real-world dataset of over 10,000 malware samples from 8 classes. It achieves an F1-Score of one for most classes, with an average of 0.93 and a limited performance overhead of 1.5%.

Index Terms—Dynamic Malware Analysis, Machine Learning for Security, Cross-dimensional Malware Analysis, Case-sensitive Detection, Multi-input Ensemble

✦

1 INTRODUCTION

MALWARE attacks against enterprises have proliferated at an alarming scale. Industry analysis reports almost 17 million malware programs targeting businesses in 2021, with estimated financial losses in billions [1]. The ramifications of these attacks range from user-annoying popups to ex-filtration of sensitive data, financial loss, extortion, and even sabotaging critical infrastructures. Accordingly, malware programs can be grouped into classes based on their objectives and functionalities – Potentially Unwanted Applications (PUA) pop up unwelcome advertisements; Bankers stealthily steal financial credentials; Backdoors open hidden access paths for a remote adversary; Spyware stealthily exfiltrates sensitive data of its victim; Downloaders install a malicious payload; Cryptominers mine cryptocurrencies for the adversary while Ransomware encrypts the data victim for extortion.

The diversity in malware classes can impact the *data-source*, *features* and the *user-requirements* that are instrumental in analyzing and detecting malware, as illustrated in Figure 1. First, the optimal run-time *data-source* that can detect a malware class differs based on the functionality. Backdoors maintain consistent communication with a remote adversary, leaving strong indicators on the network, whereas spyware are likely to leave indicators on the operating system (OS) when they scan a large number of files. On the other hand, ransomware are prone to trigger distinct hardware events due to the encryption they perform. Second, malware
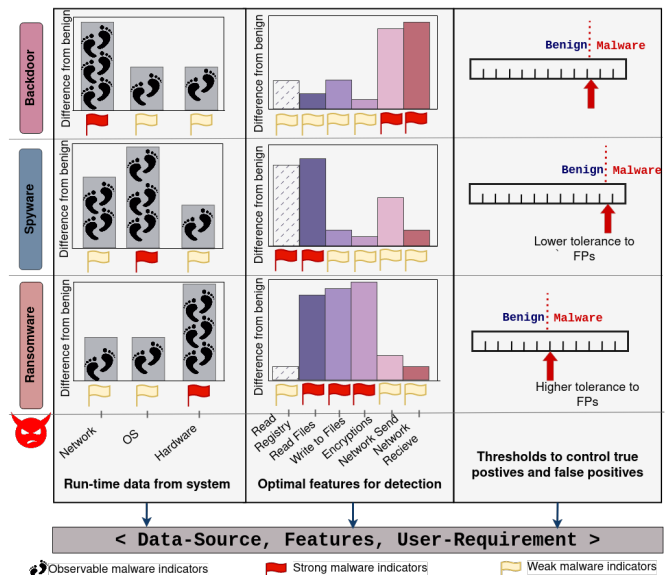
• *S. Karapoola, N. Singh, C. Rebeiro and Kamakoti V. are with the Department of CSE, Indian Institute of Technology Madras, Chennai, India.*

E-mail: {sareena, nik, chester, kama}@cse.iitm.ac.in.

Fig. 1: The optimal tuple ⟨`data-source`, `features`, `user-requirements`⟩ varies for each malware class. The optimal run-time `data-source` and `features` that can distinguish a malware class from benign applications differ based on the malware functionality. Further, low-risk malware (e.g., spyware) typically have stricter classification thresholds than high-risk malware (e.g., ransomware).

classes differ in the *features* that best identify them. For example, in the OS, a high number of encryptions and write-to files are indicators of ransomware, whereas a high rate of file-system and registry reads are indicators of spyware. Third, the *user's requirements* vary for different malware classes. For high-risk malware like ransomware, users are more likely to tolerate false positives than low-risk mal-
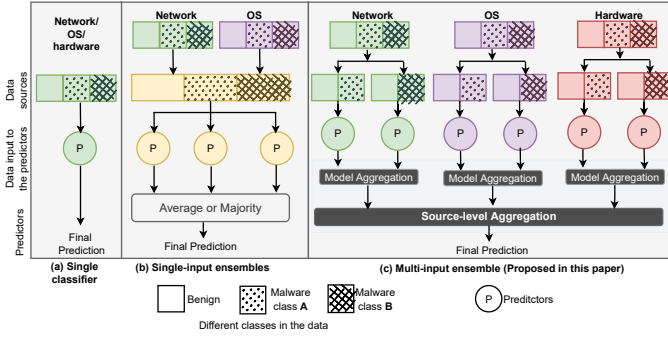
Fig. 2: Malware detection mechanisms based on (a) single classifier; (b) single-input ensembles; and, (c) multi-input ensemble (proposed in this paper), for an example case of three classes (benign, malware classes A and B).

ware like spyware and PUAs. Thus, a model for high-risk malware would ideally need lower classification thresholds than low-risk malware. Hence, detecting a malware class can benefit in accuracy and false-positive guarantees with models trained with the optimal data-source (network, OS, or hardware) and fine-tuned for class-specific features and classification thresholds. In fact, we observe that the optimal tuple of ⟨data-source, features, user-requirements⟩ is unique for each malware class. Additionally, the optimal tuple is also sensitive to the system load conditions, which can infiltrate noise into the run-time data.

The intuitive approach to leverage the optimal tuple for any malware class is to have different specialized predictors, wherein each predictor is fine-tuned for a specific data-source, class-specific features, and requirements. Naïve solutions include hyper-specialized predictors for classes such as ransomware [32], which fail on other classes of malware. On the other hand, most works in literature employ single generic classifiers, that are too generalized to support such a specialized handling of different classes (Refer to Figure 2a) [2]–[12]. Alternative works explore an ensemble-based classifiers for detection [13]–[23]. Such solutions train a collection of predictors either in parallel or sequentially on the *same input data* formed by combining data from one [13]–[20] or multiple sources [21]–[23]. Figure 2b illustrates one such single-input ensemble that trains predictors in parallel. Primarily, these approaches aim to minimize the detection error by averaging the predictions from the individual predictors (when trained in parallel) or learning adaptively (when trained in sequence). Either way, *same-input ensembles* cannot achieve the optimal tuple for detection as they do not support fine-tuning individual predictors to pursue class-specific features and requirements.

Figure 2c illustrates our proposed ensemble, which we call a *multi-input ensemble* of specialized predictors, wherein each predictor is trained using a different input data-source to detect a specific malware class from benign applications. However, given an unknown program sample, such fine-grained specialization alone is not sufficient to leverage the optimal predictor (i.e., achieve the optimal tuple) in the ensemble. The primary challenge lies in arriving at a *consensus* from the independent, conflicting predictions from different specialized predictors for any input test program. As each predictor is tuned differently, their inferences are likely to conflict. Naïve approaches such as majority or

average of individual predictions [13], [17], [18], [20] may not be optimal. This is because each predictor has different definitions of boundaries between malware and benign behavior and deals with different noise levels in the input from different data sources.

This paper presents SUNDEW, a detection framework that employs a *multi-input ensemble* of predictors with an *insightful aggregation* mechanism to leverage the optimal tuple of ⟨data-source, features, user-requirements⟩ for any input test program. SUNDEW has three *components*, each utilizing different data sources across the system stack to provide a holistic view of malware activity. Internally, each component has multiple specialized predictors, each tuned to differentiate a specific malware class from benign applications using data collected from the particular data source (network, OS, or hardware) (Refer to Figure 2c). To resolve the conflicts between the predictors, SUNDEW uses a two-level hierarchical structure of aggregator functions that first collates the inferences from different specialized predictors inside each component and later aggregates the inferences from the three data sources (Refer to Figure 2c). These functions employ a combination of predictor statistics, prior knowledge of the capabilities of each predictor, risk factor, and the current system load to aggregate inferences to an optimal prediction. Thus, unlike prior works, the holistic view of malware activity and specialization enable SUNDEW to achieve a case-sensitive analysis and detection, thus improving the accuracy and resilience while ensuring class-specific false-positive guarantees. Following are the contributions of the paper:

1) A reliable malware analysis framework, SUNDEW, with a holistic view of malware activity across the system, an ensemble of specialized predictors, and aggregator functions to help derive the best-case prediction for any malware class (Section 5).

2) An evaluation of various design choices for the aggregator functions that resolve conflicts between the independent specialized predictors. Given an unknown sample, the two-level aggregation in SUNDEW relays the inference of the specialized predictor to the output without any loss, while boosting the performance of the optimal predictor by at least 1.42% (Section 6).

3) An evaluation of SUNDEW on a rich dataset that presents precise and comprehensive real-world malware behavior of more than $10,000$ malware samples, including cryptominers, bankers, spyware, backdoors, ransomware, downloaders, deceptors, and potentially unwanted applications (PUAs). SUNDEW can achieve an F1-Score of 1 for most malware classes, an average of 0.93 for any malware class, and 0.82 even under highly noisy conditions, with an average overhead as low as 1.5% at the end-host machines (Section 7).

4) To the best of our knowledge, SUNDEW is the first to provide a multi-input ensemble for a case-sensitive detection of malware classes. It evaluates the run-time inputs from three system components, risk factors, and the dynamic system noise, to detect malware reliably. SUNDEW is 10% more accurate, with 89% lower false positives, than prior state-of-

the-art predictors based on network [2], OS [19], hardware [20] and ensembles [23] that do not consider a holistic view of malware activity and multi-input ensemble of specialized predictors [2]–[28].

Following is the organization of the rest of the paper. Section 2 provides the necessary background for the paper. Section 3 highlights the motivation for the need for an ensemble of predictors. Section 4 presents the related work. We discuss the high-level overview of SUNDEW in Section 5. Section 6 discusses the use of different insights to effectively aggregate predictions in SUNDEW. Section 7 presents the implementation of SUNDEW and results of our evaluation. Section 8 discusses the limitations of SUNDEW and future work. Finally, Section 9 concludes the paper.

2 BACKGROUND

Malware are programs with malicious intents. With differing attack objectives, they pose varying levels of risk to system users. Ransomware that can sabotage an entire system is a high-risk malware, whereas adware or potentially unwanted applications (PUA) that are mere user-annoying in nature are low-risk malware [29]. Table 1 presents a few notable malware classes with their objectives and corresponding risk levels.

Run-time data sources for malware detection. Malware behavioral analysis and detection is a widely studied and mature field [2]–[28], [30]. The detection mechanisms use trails of malware activity, observable at different system components which include – **(1)** Network (e.g. malware communications to its command-and-control server); **(2)** Operating system (e.g. system calls); and, **(3)** Hardware (e.g. micro-architectural events).

These behavioral trails provide different abstractions of malware behavior. The network trails provide insights into malware communications to external entities, including its command-and-control servers. The features of interest include the unencrypted meta-data about connections, domains contacted, TLS handshakes, and X.509 certificates from HTTPS/HTTP flows. In contrast, the OS component captures the malware interactions with the system software when it attempts to remain stealthy, achieve persistence, and execute its objective. These interactions include the file system, registry, process, and network-related system call traces of the malware. Though the system call traces include network communications (TCP Send/Receive), they are at a higher abstraction as compared to that captured at the network component. Additionally, malware activities also trigger specific micro-architectural events that can be observed using special registers called Hardware Performance Counters (HPCs) [31]. A few notable HPC events used for malware detection include cache hits/misses at various levels, branch instruction response, and CPU activity. These behavioral trails are used to train detection models that predominately rely on machine learning (ML) to differentiate malicious and benign behavior.

Relevance of HPC in malware detection. Though the use of HPCs for malware detection is much debated, they are found to be beneficial when used in the right way, using interrupt and context switch management [32]. Additionally,

TABLE 1: Objectives and risk levels of various malware classes

Malware Class	Objectives	Risk level [29]	User requirements
Cryptominer	Exploit computing resources of the victim to mine cryptocurrencies	High	High TPR
Banker	Steal financial credentials	High	High TPR
Spyware	Infiltrate and keep gleaning sensitive information for an extended period	Medium	Low FPR
Backdoor	Grant alternate covert pathways to system resources, bypassing access control	Very High	High TPR
Ransomware	Sabotage user files and extort a ransom from the user for restoration	Very High	High TPR
PUA	Pop-up annoying advertisements and inappropriate content	Low	Low FPR
Downloader	Covertly download other malware from a remote server to execute and infect	Low	Low FPR
Deceptor	Bypass the system with close to benign behavior and adware-like payload	Low	Low FPR

with minimal instrumentation required in the HPC trails before they can be fed to the models and their limited performance overheads, HPC-based detection enables a trade-off between accuracy and overhead in malware detection.

User Requirements. The detection models are fine-tuned for an optimal trade-off between two orthogonal user requirements – **(1)** a high true-positive rate (TPR), with some tolerance to mispredictions, to detect as many malware as possible; or a **(2)** a low false-positive rate (FPR), with no tolerance to mispredictions, to prevent any impact on benign applications. These requirements are, in turn, dependent on the risk level of the malware. Users would prioritize a high TPR for high-risk malware while preferring a low FPR for adware/PUA that are very similar to benign applications in behavior. Table 1 provides preferred user requirements based on the risk level of different malware classes.

3 MOTIVATION

In this section, we analyze the differences in malware runtime activity and present our observations that motivate the need for a case-sensitive analysis of malware. To this end, we study the difference in the behavior of malware classes from benign applications observable across the three data-sources. For each class, we assess the difference using specialized binary ML models, each trained with behavioral data of 1000 programs, including the malware class and benign applications. Based on our observations, we make the following claims on the benefits of exploring multiple data-sources and class-specific features and addressing user-specific requirements.

C-1: Detection can benefit in performance from a cross-dimensional view of malware activity. Malware classes differ in their actions. In turn, the actions determine the quantum of malware activity across the system: network, OS, and hardware. Figure 3 indicates the distinguishability of activities of different malware classes from benign applications across the three data-sources. The darker the
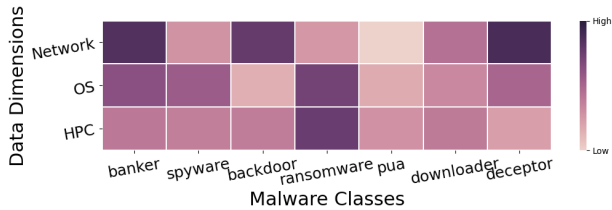
Fig. 3: Differences between malware and benign behavior for different classes across the three data data-sources. The darker the color, the more distinguishable are the activities from benign applications.

color, the more distinguishable are the activities from benign applications. The network shows a high distinction in activities for banker, backdoor, and deceptor, whereas OS shows distinguishable activity for ransomware and spyware. Similarly, hardware shows distinct activity in the case of ransomware. Hence, some classes are better aligned to be detected using a specific data-source than others. We explain this with an example of backdoor, spyware, and ransomware (Refer to Figure 1). A backdoor creates a reverse shell, escalates privileges, and provides code injection capabilities to a remote adversary. However, the constant factor in its attempt to execute any command from the adversary is its sustained communication with the remote server. We observe that the average duration of network flows for a backdoor is notably different from other classes. On the other hand, spyware aims to gather information about the victim. Hence, it scans the filesystem, leaving distinct trails at the OS, while its network activities are not significantly distinguishable from benign applications. Similarly, ransomware scans the files at the victim to encrypt and make them inaccessible. In the process, a high rate of reading and writing files is observable at the OS. However, the high encryption rate leaves a significant fingerprint of micro-architectural events visible at the hardware. Hence, it is beneficial to *explore multiple data-sources to get a complete picture of malware activity*, and build appropriate defenses.

C-2: Detection can benefit in resilience by employing all data-sources. The data collected at the network and hardware is affected by other processes executing in the system. With an increase in system load (i.e., the number of processes), the network communications of other processes get induced into the network data. Similarly, other processes sharing micro-architectural resources in the system can affect the hardware performance counters. In contrast, the OS data is collected only for the specific PID and hence is agnostic to the system load. Thus, it is beneficial to *employ multiple data-sources for resilient malware detection.*

C-3: Detection can benefit from class-specific features. Within each data-source, malware classes differ in the features that best express their maliciousness. As an example, in hardware, the performance counter event, which counts the number of switches from the Decode Stream Buffer (DSB) to the Micro-instruction Translation Engine (MITE), DSB2MIT_SW_CNT is empirically one of the most important features in classifying ransomware from benign applications. However, for spyware, the corresponding feature is the event M_LD_Ret_L1Hit, which counts retired loads that encounter a hit in the L1 cache in a specific cache coherency state. Thus, *it is beneficial to have predictors specialized for class-*
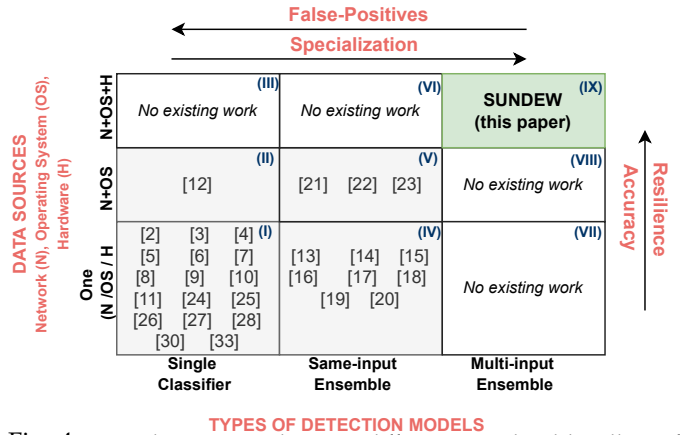

Fig. 4: Prior detection mechanisms differ in specialized handling of malware classes based on the run-time data source and the machine learning models they employ.

specific features rather than a generic approach.

C-4: Detection can benefit if fine-tuned to appropriate user requirements. System users respond differently to different malware classes. For instance, users would want to kill ransomware as soon as possible to restrict further damage. Hence, models for high-risk malware (e.g., ransomware, backdoor) target a high true-positive rate to detect every malware sample, while tolerating some false positives. In contrast, users are comparatively lenient to low-risk malware (e.g., PUA and deceptor) that are mere user annoying in nature. Users would prefer to kill such malware only if the prediction is precise to reduce any impact on benign applications. Thus, false positives are a concern for such malware. The predictor classification threshold controls the trade-off between the true-positive rate (TPR) and the false-positive rate (FPR). High-risk malware would require lower thresholds to promote high TPR, whereas low-risk malware would require stricter thresholds to reduce FPR. Hence, *it is beneficial to have predictors specialized for class-specific user requirements.*

In essence, the optimal tuple ⟨data-source, features, user-requirement⟩ for each malware class is different, making it essential to have a holistic view of the data-sources and specialized models to improve detection efficiency.

4 RELATED WORK

Malware analysis and detection using run-time behavior have been extensively explored in literature [2]–[28], [30], [33]. Figure 4 compares the highly cited prior works in the last decade based on the run-time data sources and the machine learning models they employ to cater to different malware classes.

Run-time data sources. Most prior works employ run-time data from a *single* system component to detect malware (refer to cells I and IV in Figure 4). These include trails from the network [2], [4], [6], [7], [13]–[16], [24], [25], operating system (OS) [5], [11], [19], [28], or hardware [3], [8]–[10], [17], [18], [20], [26], [27], [30], [33]. Alternatively, few works employ a combination of features extracted from the network and OS (refer to cells II and V in Figure 4 [12],

[21]–[23]). While the multiple inputs from network and OS can increase the detection accuracy, these solutions can miss indicators of classes like ransomware that have high micro-architectural activity (refer to C-1 in Section 3). Unlike prior solutions, SUNDEW employs a comprehensive view of malware activity across the computing stack (cell IX), including hardware, thus improving the accuracy and resilience of detection (refer to C-1 and C-2 in Section 3).

Generic vs. Specialized models. State-of-the-art solutions in dynamic analysis use different detection models, as shown in Figure 2 and in the columns of Figure 4. While these models can provide a binary or multi-class prediction, internally, they either use a *single* classifier (cells I and II) or a *same-input ensemble* of predictors (cells IV and V). A single classifier can use data from a single (cell I [2]–[11], [24]–[28], [30], [33]) or multiple sources (cell II [12]). Though relatively light-weight, most single predictors are too generalized to support any specialization [2]–[12], [24]–[28], [33]. On the other hand, hyper-specialized predictors [30] that are fine-tuned for specific classes, such as ransomware, can fail on other classes of malware.

Alternatively, a *same-input* ensemble (cell IV) consists of multiple predictors, wherein all predictors train on the same data. These predictors are trained either in sequence or parallel to improve the predictive performance [13]–[20]. While the former approach trains the predictors sequentially in an adaptive manner [14]–[16], [19], the latter employs all predictors in parallel and outputs the average of their predictions [13], [17], [18], [20]. Inherently, such ensembles do not support training predictors with different data pertaining to a specific class and benign behavior. The addition of alternate data sources [21]–[23] (cell V) does not help either, as the features from different input sources are transformed to a single representation before feeding to the model. Unlike prior works, SUNDEW proposes a *multi-input* ensemble of predictors, wherein each classifier trains on a different run-time data source and malware class and is specialized to maximize the class-specific user requirement (cell IX).

When predictors in a same-input ensemble (that are trained in parallel) test any given unknown sample, their independent predictions are *aggregated* (e.g., by averaging) to minimize the cumulative errors of the individual predictors and form a final prediction [13], [17], [18], [20], [22]. In contrast, aggregation in the multi-input ensemble of SUNDEW, aims to relay the inference of the specialized predictor corresponding to the given sample to the final output. Challenges in such an aggregation are two-fold. First, the individual predictors in a multi-input ensemble deal with different noise levels in the input data from different data sources. Second, the predictors have different definitions of positive and negative class. We discuss how SUNDEW addresses these aggregation challenges to relay the optimal prediction of the specialized predictor to the final output in Section 6. Other industrial solutions such as [34] present class-specific behavioral analysis similar to the goals of SUNDEW. However, based on our limited understanding, these closed-box solutions do not support aggregation to present a final prediction of the input sample.

Thus, the comprehensive view of malware run-time ac-tivity, the multi-input ensemble and the aggregation mechanism ensure that SUNDEW is **(1)** more accurate in detecting malware (unlike [2]–[28], [33] that do not exploit C-1 and C-3); **(2)** more resilient to infiltrating noise (unlike [2]–[4], [6]–[10], [13]–[18], [20], [24]–[27] that do not exploit C-2); **(3)** can support class-specific false positives (unlike [2]–[28] that do not exploit C-4); and, **(4)** caters to a diverse set of malware classes (unlike [30]).

Comparison with SIEM. The comprehensive analysis in SUNDEW has similarities with industrial efforts such as System Information and Event Management (SIEM) [35] that consider a holistic correlation of events across an enterprise to detect threat scenarios. However, there are differences between the two. First, SIEM collects data from diverse sources such as antivirus software, security appliances, firewalls, and other organizational applications, including human interactions (such as repeated access attempt failures). It correlates these events against pre-defined rules to detect threats and create alerts. In contrast, SUNDEW is an alternative input source to SIEM that can replace the antivirus software to provide accurate and resilient detection of malware activities. Second, the machine learning ensemble in SUNDEW is more sophisticated than the correlation rules predominantly used in SIEM and thus capable of detecting zero-day threat scenarios.

Comparison with multi-input solutions in static analysis. Few prior efforts based on static analysis have explored malware detection using multi-input ensembles [36], [37]. These solutions employ a heterogeneous ensemble of predictors, where each classifier trains on different static features extracted from the malware binary. However, unlike dynamic analysis, static techniques can be easily evaded by polymorphic and metamorphic malware that are popular today. To the best of our knowledge, SUNDEW is the first multi-input ensemble for dynamic analysis, considering a comprehensive view of run-time activity to provide a case-sensitive detection of malware.

5 THE SUNDEW FRAMEWORK

In this section, we first present a high-level overview of SUNDEW followed by a formal description of the multi-input ensemble.

5.1 High-Level Overview

SUNDEW is a multi-input ensemble of predictors that leverages the optimal tuple of ⟨`data-source`, `features`, `user-requirements`⟩ for accurate and resilient detection of any malware class. Figure 5 presents a high-level overview of the working of SUNDEW. The ensemble has a component for each data-source, namely, network, OS, and hardware (shown by the dashed boxes in Figure 5). During program execution, a collection engine collects the program's network, OS, and hardware behavioral data and invokes the respective components. Internally, each component has multiple predictors, each of which is specialized for different malware classes with class-specific features and user requirements (refer to **C-3** and **C-4** in Section 3). Each predictor is a binary classifier trained for a specific class,
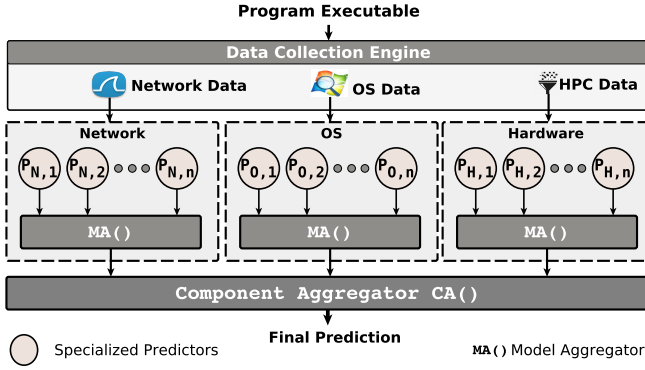
Fig. 5: **Multi-input ensemble of predictors in** SUNDEW. The ensemble consists of three components corresponding to each data source (network, OS, hardware). Internally, each component has a specialized predictor for each class. `MA()` aggregates the inferences of the specialized predictors inside each component, whereas the `CA()` aggregates the inference of the three components to make the final output of SUNDEW .
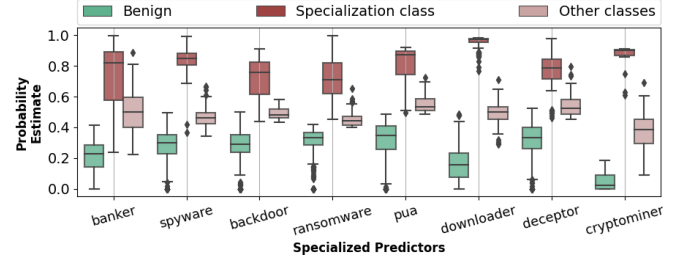

Fig. 6: Distribution of probability estimates of different specialized predictors in the network component, when tested with data of benign class (green), the corresponding specialization (red), or from any other class (light red). Each predictor has different definitions of class boundaries.

where it can infer if the program is of the malware class of its specialization or benign. For example, a backdoor-specialized predictor predicts if a program is a backdoor, likewise, a PUA-specialized program predicts if a program is a PUA.

The predictors are likely to output conflicting inferences. Primarily, each specialized predictor has a different definition of the boundary between benign and malware. Figure 6 shows the distribution of probability estimates of different specialized predictors in the network component when they are tested with benign applications (green boxes), and the class of their specialization (dark-red boxes). As evident these distributions overlap. Thus, due to similarities between deceptor and some benign applications, the backdoor-predictor might infer a deceptor program as benign while, the deceptor-predictor infers it as a deceptor. Further, while a predictor can predict its class of specialization with high confidence (dark-red boxes), it predicts other classes as malware with varying likelihood (light-red boxes). As SUNDEW starts with no notion of the program's class; the challenge lies in choosing the right prediction from the set of independent predictions. SUNDEW addresses this challenge by aggregating predictions using a configurable model-aggregator within each component. The model-aggregator $MA()$ multiplexes the output of the best-case specialized predictor to the output. For this, it leverages predictor statistics (e.g. probability estimates) and prior knowledge to assess the *confidence* of inferences inside each component. Prior knowledge can include the capabilities of components to reveal certain classes as observed during the training phase. For instance, prior knowledge that hardware trails have strong indicators of ransomware can help assess the confidence of a ransomware-specialized predictor in hardware. Based on the statistics and prior knowledge, $MA()$ computes the confidence score of each predictor and relays the most confident inference as the output of the component.

The outputs from the three $MA()$s are likely to differ on the class of the program due to two reasons. First, each data-source varies on its capability to distinguish a specific malware class from benign (Refer to Figure 3 and Claim C-1 in Section 3). Second, the data-sources have varying levels of noise from other processes running in the sys-

tem (Refer to Claim C-2 in Section 3). For this, SUNDEW uses a component-aggregator function $CA()$ to aggregate the outputs from the model-aggregators and yield the most confident inference as the final classification output. Similar to $MA()$, it exploits the components' statistics (the output confidence score from $MA()$) and its prior knowledge. Its prior knowledge can include a broader understanding of the distinguishing capabilities of different data sources. Further, the $CA()$ also checks the system load and considers the output of a resilient data source, with the least infiltrating noise for aggregation to the output. At both model and component aggregators, multiple predictors/components likely can end up with similar confidence scores. In such scenarios, both $MA()$ and $CA()$ leverage the known risk-level [29] of the classes as a tie-breaker. They choose the riskiest class as their aggregated inference to resolve the tie. For instance, if two conflicting inferences, backdoor and deceptor have similar confidence scores, the aggregators choose the higher-risk class (backdoor) of the two.

Thus, given any input test program, relaying the predictive benefits of the corresponding specialized predictor (that employs the optimal data-source, features, and user-requirements) to the output of SUNDEW is important to improve accuracy and resilience. The choice and weights of statistics and prior knowledge control this relay and the effectiveness of the aggregation. We explore this aspect and evaluate different designs for $MA()$ and $CA()$ in Section 6.

5.2 Formal Description of SUNDEW

Let $\mathbb{B} = \langle \mathcal{D}, \mathcal{M}, \mathcal{P}, \mathcal{A} \rangle$ represent the SUNDEW ensemble (Refer to Figure 5). $\mathcal{D} = \{\mathtt{N}, \mathtt{O}, \mathtt{H}\}$ is the set of components (i.e. data sources), namely network ($\mathtt{N}$), OS ($\mathtt{O}$) and hardware ($\mathtt{H}$). $\mathcal{M} = \{\mathtt{m_1}, \mathtt{m_2}, \dots \mathtt{m_n}\}$ is the set of n malware classes. $\mathcal{P}$ is the set of specialized predictors, while $\mathcal{A}$ is the set of aggregator functions. Algorithm 1 describes SUNDEW. It takes as input a program $\mathtt{z}$ from the set of programs $\mathtt{Z}$ to test. For each component $\mathtt{k}$, it first gets the behavioral data for the program (Line 5).

Behavioral data. Given a program $\mathtt{z}$, its behavioral data in component $\mathtt{k} \in \mathcal{D}$ corresponds to a time series of snapshots collected during the execution of the program. These snapshots are captured at different granularities across components, as shown in Figure 7. At the network, we log the

Fig. 7: Behavioral snapshots collected at different granularities at network, OS, and hardware components.

Fig. 8: Percentage of malware rows per program varies for different classes, across the three components (data sources).

data for every flow[1] while we log every system call at the OS. The hardware component logs the HPCs at a fixed time interval of 100ms. The collected data is pre-processed and converted to a matrix $d_{z,k}$, with features as columns and rows representing each snapshot (Line 5). These rows are labeled with the class of the program, $m_j \in \mathcal{M}, \forall j \in [1, n]$.

Prediction. After getting the behavioral data, Algorithm 1 invokes all the predictors in $\mathcal{P}$, each of which is specialized to detect one of the classes in $\mathcal{M}$ (Line 6). $\mathcal{P}$ is given by,

$$\mathcal{P} = \{P_{k,j}(), \forall k \in \mathcal{D}, \forall j \in [1, n]\} , \quad (1)$$

where predictor $P_{k,j}$ is specialized in component k to classify a program as malware m_j or benign. These predictors are trained to predict row-wise inferences along with the probability estimate of a row in $d_{z,k}$ being malicious. Accordingly, the probability-estimates for $d_{z,k}$ is the cumulative average of probability estimates of its rows. As the predictors are trained row-wise, the prediction of a test program might contain some rows inferred as malware (malicious) while others as benign. Interestingly, the percentage of malicious rows per program, i.e. malicious-row-percentage varies for different classes (Refer to Figure 8). Accordingly, the specialized predictors $P_{k,j}()$ are fine-tuned to these class-specific thresholds (in Figure 8) to conclude the class of the test program. For instance, the network-based backdoor-specialized predictor infers a program as malware if at least 40% of the rows are identified as malicious. Similarly, spyware-specialized predictor infers malware if at least 30% of the rows in the program are malicious.

Hence, for data $d_{z,k}$ of a program z in component k, $P_{k,j}$ outputs a tuple of its prediction $r_{k,j}$ and statistics $s_{k,j}$ as follows:

$$\langle r_{k,j}, s_{k,j} \rangle = P_{k,j}(d_{z,k}), \forall k \in \mathcal{D}, \forall j \in [1, n] , \quad (2)$$

where,

$$r_{k,j} = \begin{cases} 1 & \text{, for a malware of class } m_j \\ 0 & \text{, for a benign program} \end{cases} , \quad (3)$$

and, $s_{k,j}$ is a tuple of $\langle$ probability-estimates, malicious-row-percentage$\rangle$. The statistics $s_{k,j}$ is an indicator of *confidence* of the prediction of $r_{k,j}$. Thus, the output from the predictors in component k are the set of predictions ($R_k = \{r_{k,j}, \forall j \in [1, n]\}$) and statistics ($S_k = \{s_{k,j}, \forall j \in [1, n]\}$) (Line 7 in Algorithm 1). Each element in these sets corresponds to a malware class.

---

1. All communications having the same source and destination IP address, and source and destination port belong to a flow. Thus the network packets are grouped into traffic flow summaries
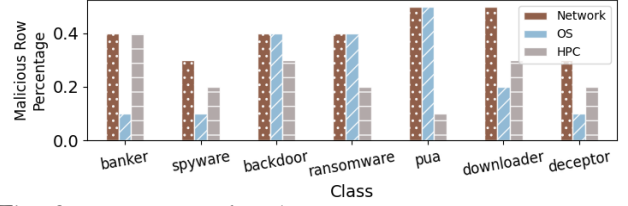

Algorithm 1: SUNDEW

Input: z: Program to test
Result: $\langle \hat{r}_{\mathbb{B}}, \hat{c}_{\mathbb{B}} \rangle$: Final label and confidence.
1 **begin**
2    $\mathcal{D} \leftarrow \{\text{N}, \text{O}, \text{H}\}$ components
3   PriorKnowledge $\leftarrow$ Prior knowledge on predictors in $\mathcal{P}$
   /* Aggregating predictions at model-level */
4   **for** $k \in \mathcal{D}$ **do**
5      $d_{z,k} \leftarrow$ Behavioral data of z in component k
6      $P_k \leftarrow$ Specialized predictors in k
7      $\langle R_k, S_k \rangle \leftarrow P_k(d_{z,k})$ ▷ Get predictions
8      $\text{Expert}_k \leftarrow$ Expert set of predictors in k ▷ (derived from PriorKnowledge)
9      $\langle \hat{r}_k, \hat{c}_k \rangle \leftarrow$ MA $(R_k, S_k, \text{Expert}_k)$
10   ▷ Highly confident prediction in R_k
   /* Aggregating predictions at components Level */
11    $\hat{R} \leftarrow \{\hat{r}_k, \forall k \in \mathcal{D}\}$ ▷ Prediction of each component
12    $\hat{C} \leftarrow \{\hat{c}_k, \forall k \in \mathcal{D}\}$ ▷ Confidence of each component
13    $\hat{E} \leftarrow$ {Prior-known strength of component k to predict $\hat{r}_k$} ▷ Derived from PriorKnowledge
14    $L \leftarrow$ Number of processes in the host machine ▷ System load
15    $\langle \hat{r}_{\mathbb{B}}, \hat{c}_{\mathbb{B}} \rangle \leftarrow$ CA $(\hat{R}, \hat{C}, \hat{E}, L)$
16   ▷ Highly confident prediction among components
17   **return** $\langle \hat{r}_{\mathbb{B}}, \hat{c}_{\mathbb{B}} \rangle$

Aggregation. The independent predictions in R_k are likely to conflict, as shown in Table 2A, which shows an example output of the specialized predictors in the network component when tested with a backdoor sample. While four predictors (e.g., the ones specialized for banker and backdoor) detect the sample as malware, others predict it as benign. The conflicts arise as each predictor has different estimates for maliciousness (S_k), including probability-estimates (Figure 6) or the number of malicious rows in a program (Figure 8). Similarly, the components may differ in their prediction (Refer to Table 2B), primarily due to the varying noise levels that affect the probability-estimates. While the network component flags the sample as malware, OS and hardware component predicts it as benign. For an unbiased aggregation, while leveraging the benefit of the optimal predictor in each component, SUNDEW adopts a two-level aggregation as shown in Figure 5. Thus, $\mathcal{A} = \{\text{MA}(), \text{CA}()\}$ is a set of aggregator functions that compare the confidence of each predictor to resolve conflicts inside each component and among the components. In either case, the statistic S_k is not sufficient, as aggregating based on the maximum or average of S_k may not relay the optimal prediction (gray cells) to the output (red cells) in most cases (Refer to Table 2).

TABLE 2: **(A)** Conflicts among specialized predictors when tested with a backdoor sample in the network component. While four specialized predictors (e.g., banker, backdoor) detect the sample, four others predict it as benign. Naive aggregation schemes based on S_k (probability) may not be optimal to relay the output of the backdoor-specialized predictor (gray cell) to the output (red cells). Similarly, **(B)** illustrates the conflict among different components about the sample. While the network component detects it as malware, OS and hardware components flag it as benign.

(A) Output from predictors inside a component									Aggregation		
Specialized predictor	Cryptominer	Banker	Spyware	Backdoor	Ransomware	PUA	Downloader	Deceptor	Majority?	Maximum probability	Average probability
Prob-1	0.47	0.38	0.33	0.73	0.33	0.34	0.19	0.49	-	0.73	0.41
Prob-0	0.53	0.62	0.67	0.27	0.67	0.66	0.81	0.51	-	0.81	0.59
Prediction	1	1	0	1	0	0	0	1	?	0	0

(B) Output from each component*			Aggregation		
	Network	OS	Hardware	Majority?	Maximum	Average
Prob-1	0.73	0.58	0.20	-	0.73	0.50
Prob-0	0.27	0.42	0.80	-	0.80	0.49
Prediction	1	0	0	0	0	0

* Assuming each component outputs its best prediction.
Prediction of 1 indicates malware class, and 0 indicates benign class. Accordingly, **Prob-1** indicates the probability of the sample being malware. **Prob-0** indicates the probability of the sample being benign.

Prior knowledge. To validate the confidence put forward by S_k, SUNDEW also leverages the knowledge built using past experience or domain insights. Thus, $\texttt{PriorKnowledge}(P_{k,j})$ is a comparative measure of prior-known efficiency of predictor $P_{k,j}$ to detect malware m_j in component k, $\forall k \in \mathcal{D}$, $\forall j \in [1, n]$. An example of such a measure is the F1-Score of $P_{k,j}()$ observed in the train-validate phase or past deployments of SUNDEW. Based on this measure, some predictors are experts (more confident than others) in a component. For instance, backdoors' operations are known to be network-intensive from domain insights. Thus, if the backdoor-predictor in the network component predicts a program as a backdoor, its prediction is likely to be the most accurate. Accordingly, the expert set $\texttt{Expert}_k$ of a component k (Line 8 in Algorithm 1) is the set of predictors having high prior-known strengths in k, given by,

$$\texttt{Expert}_k = \{m_j, | \texttt{PriorKnowledge}(P_{k,j}) > \eta, \; m_j \in \mathcal{M}\} \; , \quad (4)$$

for a domain-specific configurable threshold η.

Model-level Aggregation. To deduce the most confident prediction from the output of all predictors inside a component k (i.e., $R_k = \{r_{k,j}, \forall j \in [1, n]\}$, per Equations 2 and 3), SUNDEW invokes the function $\texttt{MA}()$ with inputs: the predictions (R_k), statistics ($S_k = \{s_{k,j}, \forall j \in [1, n]\}$), and expert set ($\texttt{Expert}_k$) of predictors in k (Line 9). Internally, $\texttt{MA}()$ uses S_k and $\texttt{Expert}_k$ to evaluate the confidence of each prediction in R_k, and return $\hat{r}_k$ and $\hat{c}_k$, the most confident prediction of the component and its confidence measure (Line 9).

Component-level aggregation. At the components level, SUNDEW has three independent predictions $\hat{R}$ from each $\texttt{MA}()$ in Line 9, wherein $\hat{R} = \{\hat{r}_k, \forall k \in \mathcal{D}\}$ (Line 11). Similarly, SUNDEW has their corresponding confidence values $\hat{C} = \{\hat{c}_k, \forall k \in \mathcal{D}\}$ (Line 12). As the predictions in $\hat{R}$ are likely to differ, SUNDEW again leverages the prior known

strengths of the specialized predictor for the predicted malware class $\hat{r}_k \in \hat{R}$ in component k, given by,

$$\hat{E} = \{ \texttt{PriorKnowledge}(P_{k,\hat{r}_k}) \; \forall k \in \mathcal{D} \} \; . \quad (5)$$

These scores are indicative of the confidence of a component k in predicting $\hat{r}_k$. As noise induced by system load could affect detection, SUNDEW also considers the system load, L. For example, L can be the number of processes executing at the host machine (Line 14). Finally, SUNDEW invokes the component-aggregator $\texttt{CA}()$ to aggregate the three predictions. The function $\texttt{CA}()$ takes as input the predictions ($\hat{R}$), confidences ($\hat{C}$), and prior-known strengths ($\hat{E}$), and the system load (L). Similar to $\texttt{MA}()$, it evaluates the confidence of each component, and outputs the highly confident prediction ($\hat{r}_{\mathbb{B}}$) and its confidence ($\hat{c}_{\mathbb{B}}$), as the final output of SUNDEW (Line 15). In the next section, we discuss how SUNDEW builds insights from predictor statistics and prior-known strengths for case-sensitive detection of malware.

6 INSIGHTFUL AGGREGATION OF PREDICTIONS

The functionalities of the model and component aggregators are different. A model aggregator ($\texttt{MA}()$) resolves conflicts among predictors that use the same data source to test a sample, but have different definitions of the boundary between positive and negative classes. In contrast, a component aggregator ($\texttt{CA}()$) resolves conflicts between predictors that use different data sources (having varying levels of noise) for the same sample. While the predictions can be aggregated in different ways, the optimal aggregation mechanism for different malware classes and components varies due to the differences in the boundary definitions and the noise levels. For instance, naive comparisons such as majority [18], [22] or averaging of statistics [13], [20]) may not lead to the optimal aggregation for all malware classes. Thus, SUNDEW leverages a configurable $\texttt{MA}()$ and $\texttt{CA}()$ that explore different mechanisms on test-time predictor statistics, prior-known strengths, and system load to relay the optimal prediction as the final output, as discussed next.

6.1 Model-Aggregator

Algorithm 2 describes the model-aggregator $\texttt{MA}()$ function. For a given component k, it takes as input the predictions (R_k), corresponding statistics (S_k), and expert set of predictors $\texttt{Expert}_k$. As each predictor has different definitions of malware-benign boundary (Figure 6), $\texttt{MA}()$ aggregates predictions in a two-step process, by achieving consensus first on the maliciousness of the program and then on the specific class of the malware.

Binary consensus on maliciousness. Firstly, $\texttt{MA}()$ assesses the predictions and statistics to vote if the test program is malware or benign using a function $\texttt{ConsensusIfMalware}()$(Line 2 of Algorithm 2). Multiple alternatives are possible for realizing $\texttt{ConsensusIfMalware}()$, including consensus based on logical-OR, majority-vote, confidence, or learning. Naive mechanisms infer malware if at-least one of the predictions (*logical-OR*), or most predictions are malware (*majority-vote*). However, such approaches can significantly increase the false positives.

Algorithm 2: Model-Aggregator (for component k)

Input: $R_k = \{r_{k,j},$ Predictions from $P_{k,j} \forall j \in [1, n]\}$,
  $S_k = \{s_{k,j},$ Statistics from $P_{k,j} \forall j \in [1, n]\}$,
  Expert_k: Expert set of component k.
Result: $\langle \hat{r}, \hat{c} \rangle$: Final vote and confidence.

1 **begin**
    // Check if predictions in R indicate malware
2   $(\text{label}, \text{probmalware}) \leftarrow$
    $\text{ConsensusIfMalware}(R_k, S_k)$
3   **if** label *is* malware **then**
      // Identify the set of most confident predictor(s) from R | $r_{k,j}$ is 1.
4     $\text{C\_Set} \leftarrow \text{GetConfidentSet}(R_k, S_k, \text{Expert}_k)$
5     $\hat{r} \leftarrow$ Malware class of the most risky in C_Set
6     $\hat{c} \leftarrow$ Probability estimate of $\hat{r}$
7   **else**
8     $\langle \hat{r}, \hat{c} \rangle \leftarrow \langle \text{benign}, 1 - \text{probmalware} \rangle$
9   **return** $\langle \hat{r}, \hat{c} \rangle$

A *most-confident* ConsensusIfMalware() infers malware if the aggregated confidence of malware predictions is higher than benign predictions. It aggregates confidences using the *mean-probability-difference*, which is the mean difference between the probability-estimates of malware (Prob-1 in Table 2) and benign class (Prob-0) of all predictors. It infers malware when mean-probability-difference > 0, and benign otherwise. Another potential metric to aggregate confidences is *mean-maliciousness-difference*, which is the mean difference between the percentage of malicious and benign rows inferred for a program by all predictors. However, we find that mean-maliciousness-difference is sub-optimal for ConsensusIfMalware() as the percentage of malicious rows varies for each class (See Figure 8).

Alternatively, a learning-based ConsensusIfMalware() uses trained models to infer malware. Two configurations are possible for such models. A *booster* learns to minimize the loss function of the specialized predictors. On the other hand, a *multiplexer* learns to multiplex the output of the specialized predictor to the final output of SUNDEW. Both these configurations train their models with the predictor outputs (Refer to Equation 2) observed for all programs $z \in Z_{\text{train}}$, which is the set of programs in the training phase. Thus, these models train on $X_k = [\{P_{k,j}(d_{z,k}), \forall j \in \mathcal{M}, \forall z \in Z_{\text{train}}]$ to predict target labels $Y_k = [y_{z,k} | y_{z,k} \in \mathcal{M}, \forall z \in Z_{\text{train}}]$. The target label $y_{z,k}$ for a program z is different for booster and multiplexer. As the booster minimizes the loss function of the component, its $y_{z,k}$ is the actual-class $c \in \mathcal{M}$ of the program z. On the other hand, the multiplexer aims to relay the best-case prediction of z to the output. Thus, its $y_{z,k}$ is the predicted class of z when tested on the predictor specialized for class c in component k.

Table 3 enlists the *aggregation-loss* of MA(), which is the difference between the F1-Score of ConsensusIfMalware() as compared to the baseline, i.e., the F1-Score achieved by a specialized predictor that is optimum for the program. An aggregation-loss of zero indicates that MA() is able to relay the inference of the optimum specialized predictor to the output of SUNDEW. On the other hand, negative loss indicates that MA() can boost the detection performance

TABLE 3: **Aggregation-loss with different alternatives for** MA() **and** CA(). The comparison baseline is the F1-Score of the specialized predictor that is optimum for the program. An aggregation-loss of zero indicates that the MA() and CA() can relay the prediction of the optimum specialized predictor to the final output for any program. A negative loss indicates that the aggregation is able to improve the detection performance beyond that of the specialized predictor.

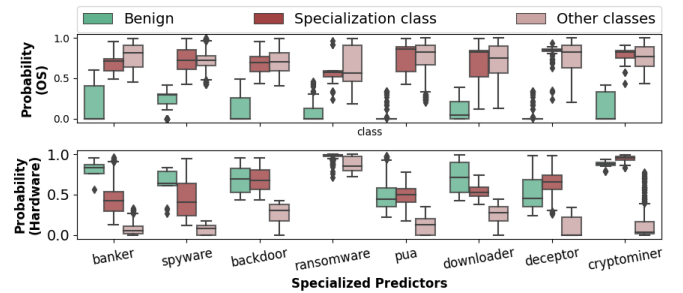|			**Network**	**OS**	**Hardware**
MA()	**Binary Consensus** ConsensusIfMalware()	Baseline	0%	0%	0%
		Logical-OR	41.92%	80.05%	80.05%
		Majority-vote	33.66%	6.81%	68.82%
		Most-Confident	59.04%	7.12%	7.12%
		Multiplexer	5.18%	4.64%	4.64%
		Booster	4.76%	−0.28%	−1.35%
			Network	**OS**	**Hardware**
	Multi-Class GetConfidentSet()	Baseline	0%	0%	0%
		Confidence	4.75%	21.51%	0.56%
		Prio-knowledge	34.16%	−1.74%	0.64%
		Confidence-Window	4.79%	21.51%	−2.18%
			Binary		**Multi-Class**
CA()		Most-Confident	−1.42%		12%
		Prior-Knowledge	−1.42%		7.86%

Fig. 9: Probability distribution of different specialized predictors in the OS and hardware component, when tested with data from the benign class (green), their corresponding specialization (red), or any other class (light red). The distributions overlap significantly in OS as compared to that in the network (Figure 6) and hardware, making confidence (probability-estimate), a poor metric to aggregate class in MA().

beyond that of the specialized predictor by minimizing its loss function. Naive voting mechanisms such as logical-OR or majority-vote have high aggregation-losses. While the logical-OR function leads to high false positives, majority-vote fails when less than half of the predictors can detect the malware sample. Similarly, the performance of the most-confident ConsensusIfMalware() can be sub-optimal as the range of probability estimates that differentiates malware and benign are different for each specialized predictor (refer to Figure 6). In contrast, the learning-based mechanisms can learn these class-specific probability estimates effectively to reduce aggregation losses. Specifically, the booster improves the performance of both OS and hardware components by at least 1% while reducing the aggregation-loss to as low as 4.76% in the network component.

Multi-class consensus. After consensus on the maliciousness of the program, the probability estimates (S_k) of individual predictors could help identify the most *confident* predictor, and hence the class. However, these estimates get unreliable when a specialized predictor attempts to predict on data of any other class (light-red boxes in Figure 6). The figure plots these estimates of different specialized predictors in the network component. Figure 9 plots the corresponding distributions in the OS and hardware component when tested with data of benign class (green), the corresponding specialization (red), or from any other class
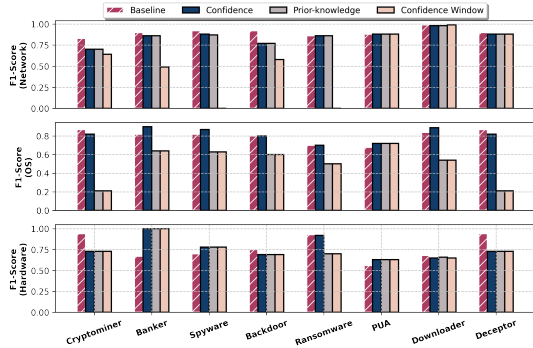
Fig. 10: Detection F1-Score observed for different classes for different alternatives for MA() at network, OS, and hardware. The baseline for comparison is the F1-Score achieved with the specialized predictor optimum for each class.

(light red). The overlapping distributions especially in the OS and hardware component makes identifying the malware objective class non-trivial. Alternatively, some predictors have a *confidence-window*, wherein the aforementioned distributions (distance between their inter-quartile range) are far apart (e.g., spyware in network component in Figure 6). If the test-time statistics of a predictor fall in such confidence windows, the predictor can be considered highly confident. As evident, confidence windows are beneficial in the network (Figure 6) or hardware component (Figure 9), whereas in OS, they can get unreliable. An alternative is to employ prior-knowledge and prioritize predictions of the *expert-set* alone (refer Equation 4).

Accordingly, if the test program is a malware, MA() evaluates the confidences of all predictions $r_{k,j} \in R_k$ which predicted malware (i.e. $r_{k,j} = 1$ in Equations 2 and 3). For this, it employs a configurable function GetConfidentSet() that evaluates the statistics (S_k) and expert set (Expert$_k$) to return a confident set of predictor(s), C_Set (Line 4 of Algorithm 2). To compute such a set, GetConfidentSet() can use one of the following metrics: **(1)** *confidence*, that returns the predictor with high probability-estimates; **(2)** *prior-knowledge*, that prioritizes expert predictors in Expert$_k$ to choose predictors with high probability-estimates; or **(3)** *confidence-window*, that returns the predictors whose statistics fall within their respective confidence window. These options can return a set of confident predictors. To resolve the contention in C_Set in such cases, MA() prioritizes the classes in accordance to the risk categories [29] and outputs the most risky class in C_Set as the prediction of the component (Line 6).

Figure 10 evaluates the F1-Score of MA() when tested with any program, for different alternatives of GetConfidentSet(), against the baseline F1-Score achieved with the corresponding specialized-predictor that is optimum for the program. The detection is inferred as correct if the risk level of the predicted class is the same or higher than that of the program. MA() can restrict the aggregation-losses to as low as 4% at the component outputs to aggregate the class for *any* program (Refer to Multi-class row in Table 3). While the confidence metric is the most effective for the network component, prior-knowledge and confidence-window metrics are effective for the OS and hardware, respectively.

Algorithm 3: Component-Aggregator

Input: $\hat{R} = \{\hat{r}_N, \hat{r}_O, \hat{r}_H\}$:Predictions from MA()for all $k \in \mathcal{D}$, $\hat{C} =$
  $\{\hat{c}_N, \hat{c}_O, \hat{c}_H\}$:Confidences from MA() for all $k \in \mathcal{D}$,
  $\hat{E}$: Prior-knowledge, L: System Load (Number of processes in the system).
Result: $\langle \hat{r}_\mathbb{B}, \hat{c}_\mathbb{B} \rangle$: Final label and confidence.

1 **begin**
2   **if** L $< \tau$ **then**
     // At low system loads, all components are reliable.
3   C_Set $\leftarrow$ Compute confident set($\hat{R}, \hat{C}, \hat{E}$)
4      $\hat{r}_\mathbb{B} \leftarrow$ Class of the most risky among C_Set
5      $\hat{c}_\mathbb{B} \leftarrow$
       Confidence of the most risky among C_Set
6   **else**
     // At higher system loads, OS is the most reliable.
7      $\langle \hat{r}, \hat{c} \rangle \leftarrow \langle \hat{r}_O, \hat{c}_O \rangle$
8   **return** $\langle \hat{r}_\mathbb{B}, \hat{c}_\mathbb{B} \rangle$

6.2 Component Aggregator

The CA() is responsible for choosing the most confident prediction in $\hat{R} = \{\hat{r}_k, \forall k \in \mathcal{D}\}$, where $\hat{r}_k$ is the output prediction aggregated by the MA() in each component k (Line 9 of Algorithm 2). CA() can weigh components based on the empirical confidence of their prediction observed at test-time ($\hat{C} = \{\hat{c}_k, \forall k \in \mathcal{D}\}$ (Line 9 of Algorithm 2)), or their prior-known strengths ($\hat{E}$, Equation 5) in predicting $\hat{R}$. Alternatively, it can weigh components based on their resilience to noise and system load. At higher loads, the OS component is the most stable and noise-free, as OS logs are collected specifically to the process PID. In contrast, the network and hardware can get noisier with an increase in the number of processes.

Accordingly, CA() (presented in Algorithm 3) takes as input the predictions from each component ($\hat{R}$), corresponding confidences ($\hat{C}$), prior-knowledge ($\hat{E}$), and the system load (L) which is the number of processes in the system. At lower system loads (Line 2), it computes a confident set C_Set using multiple options such as : **(1)** *most-confident* selects the prediction which has high $\hat{c}$; **(2)** *prior-known* selects the prediction which has high prior-knowledge scores; or, **(3)** *majority* selects the prediction that is common between at least two components (Line 3). Similar to MA(), CA() resolves contentions in C_Set by choosing the most risky class in C_Set as the final aggregated prediction (Lines 4 and 5). On the other hand, at higher system loads, the OS component is the most stable, and hence CA() outputs predictions of the OS component directly (Line 7).

Figure 11 evaluates the F1-Score obtained for different alternatives of CA() against the performance of the component that is optimum for each class. Given any program, we consider the prediction to be correct if the risk of the predicted class is the same or higher than that of the program class. Exploiting prior-knowledge, CA() is able to detect any malware boosting the performance beyond that of its best-case specialized predictor by at least 1.42%, and detect the objective class of the program with a loss as low as 7.86% (Refer to the row CA() in Table 3).
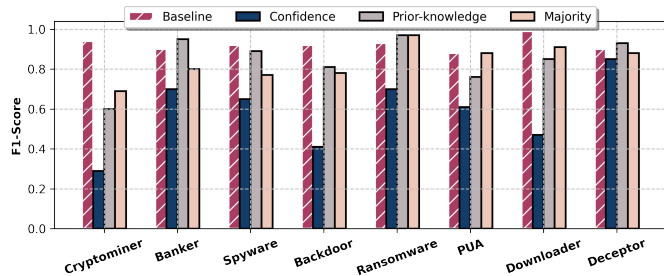
Fig. 11: Detection F1-Scores for different classes using different confident computing functions in CA(). Employing prior-knowledge gives the best F1-Score for most malware classes.

7 IMPLEMENTATION AND EVALUATION

In this section, we discuss the real-world behavioral data used to build the ensemble, followed by the implementation and evaluation of SUNDEW.

7.1 Real-World Behavioral Dataset

For unbiased cross-dimensional analyses, SUNDEW requires access to a simultaneous capture of network, OS, and hardware run-time trails of a large corpus of malware samples of different classes. SUNDEW relies on the RaDaR dataset [38] that provides such a comprehensive view of the real-world activity across the system stack of diverse Windows malware families labeled with their attack objective. RaDaR is collected by executing live malware samples (2017 ongoing) on a real-world testbed [39] with Internet connectivity, in a timely manner, when their remote command-and-control servers are highly likely to be active. Each sample is executed for 2 minutes in an automated manner, which is known to be sufficient to elicit malicious activities of most malware samples [40]. For a fair comparison, the benign samples are executed in an automated manner similar to malware, as user interactions are easily distinguishable, unlike the stealthy malware activities.

The dataset [38] provides a comprehensive set of popular features extracted based on prior works [2], [15], [16], [20], [28], [41], [42], from 7 million network packets, 11.3 million OS system call traces, and 3.3 million hardware events collected for $10,434$ samples. These features include 58 features at network, 11 at OS, and 54 micro-architectural events at the hardware [38]. Each row in the data represents a snapshot of network flow[2], system call in OS, and periodic HPC measurement in 100ms intervals in hardware. Table 4 summarizes the number of snapshots corresponding to each malware class from the three data-sources. With data of 10,434 samples evenly spread across 30 well-known malware families belonging to 8 different classes (attack objectives) and benign applications, RaDaR [38] provides a diverse representation of malware classes for evaluations.

Train-validate-test partitions. Finally, we split the dataset in a 70:15:15 ratio into the train, validate and test sets. Specifically, we ensure that the train set does not contain samples, whose data is collected at a later point of time than a sample in validate/test sets to prevent experimental biases [43]. To ensure unbiased learning, the train set contains an even distribution of benign and malware classes.

2. All communications having the same source and destination IP address, and source and destination port belong to a flow. Thus the network packets are grouped into traffic flow summaries

TABLE 4: Summary of behavioral snapshots of different malware classes from the three data-sources in RaDaR [38]. Snapshots indicate the number of flows in the network, system call traces in OS, and periodic HPC logs in the hardware.

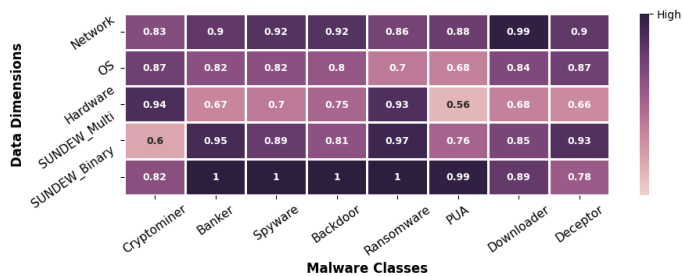|	Cryptominer	Banker	Spyware	Backdoor	Ransomware	PUA	Downloader	Deceptor	Benign
Network	992	4878	11588	7845	2239	7152	9277	4617	8964
OS	293K	772K	1.9M	1.5M	807K	2M	1.7M	440K	1.9M
Hardware	158K	51K	59K	371K	182K	914K	502K	478K	578k

Fig. 12: The F1-Score observed with SUNDEW_Binary and SUNDEW_Multi, in comparison with the specialized predictors fine-tuned for each class in each component. SUNDEW_Binary infers if a program is malware/benign, whereas SUNDEW_Multi infers the class of the program. SUNDEW_Binary achieves an F1-Score of 1 for most classes as compared to their best-case specialized predictors in network, OS, or hardware.
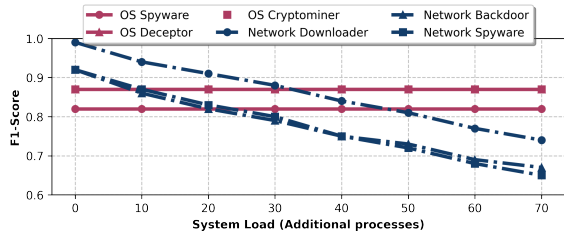
7.2 SUNDEW Implementation

We implement the specialized predictors in Python v3.6.2 using XGBoost[3] v1.4.2 library. We train each specialized predictor with the train-validate set containing data of the specific malware class and benign programs. Next, we test every specialized predictor with the train-validate sets of all other malware classes to generate conflicting predictions. The resultant predictions and statistics form the train-validate sets for the aggregators. We implement the aggregators using Python LightGBM library v3.3.1[4]. Finally, we evaluate the performance of SUNDEW using the test set of malware samples.
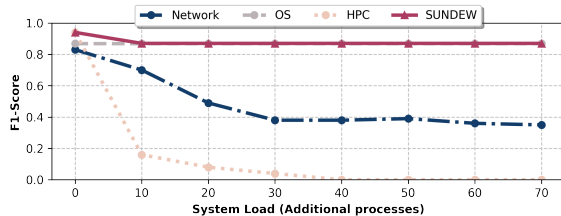
7.3 Evaluation

We compare the performance and resilience of SUNDEW against the best-case specialized predictors of all malware classes, as well as the state-of-the-art malware classifiers. Finally, we evaluate the overheads incurred by SUNDEW.

Specialized predictors. Figure 12 compares the performance of SUNDEW to detect a malware class against the corresponding predictor specialized for that class (and hence the optimum) in different components (data sources). We consider two configurations: SUNDEW_Binary measures the F1-Score of detecting if a test sample is malware/benign, whereas SUNDEW_Multi measures the F1-Score of inferring the class of the sample. As evident, SUNDEW_Binary, with its holistic view of malware activity from the three data sources, an ensemble of specialized predictors, and aggregation, can achieve performance similar to the corresponding specialized predictor for any malware class. The aggregation in SUNDEW_Binary boosts

3. https://xgboost.readthedocs.io/en/stable/python/
4. https://lightgbm.readthedocs.io/en/latest/

(a) F1-Scores of detection models based on network and OS under various load conditions for their best detectable classes. The OS-based models are resilient to infiltrating noise from increasing system load.

(b) F1-Scores of detection models based on network, OS, or hardware and that of SUNDEW under varying load conditions for Cryptominer class. SUNDEW is able to leverage the best of the three components for detection accuracy while benefiting from the resilience of the OS component.

Fig. 13: Impact of system load on detection efficacy.

TABLE 5: Comparison of SUNDEW with prior state-of-the-art solutions including single classifiers [2], [20], [42] and single-input ensembles [23] based on **(A)** F1-Score, and **(B)** False-positive rate, of detection observed on the RaDaR dataset.

Detection model	Detection Performance (F1-Score)					(B)False-Positive Rate				
	Single Classifier			SIE*	SUNDEW	Single classifier			SIE*	SUNDEW
Component ⇒ Class ⇓	N [2]	O [42]	H [20]	N + O [23]	N+O+H	N [2]	O [42]	H [20]	N+O [23]	N+O+H
Cryptominer	0.80	0.87	0.93	0.87	0.82	0.2	0.01	0.14	0.11	0.013
Banker	0.85	0.83	0.76	0.83	1	0.15	0.49	0.23	0.16	
Spyware	0.89	0.87	0.81	0.86	1	0.12	0.35	0.13	0.15	0
Backdoor	0.82	0.83	0.79	0.70	1	0.11	0.32	0.13	0.28	0
Ransomware	0.78	0.64	0.75	0.73	1	0.25	0.04	0.27	0.35	0
PUA	0.83	0.72	0.74	0.81	0.99	0.12	0.35	0.28	0.2	0.003
Downloader	0.96	0.88	0.84	0.91	0.89	0.04	0.21	0.11	0.07	0.055
Deceptor	0.87	0.88	0.78	0.86	0.78	0.13	0.51	0.24	0.15	0.051
Mean	0.85	0.82	0.8	0.82	**0.935**	0.15	0.31	0.19	0.19	**0.015**

*SIE- Same Input Ensembles, N - Network, O - OS, H - Hardware

the average detection performance beyond that of the best-case specialized predictors in any of the three components, by 1.14%. SUNDEW_Binary has an F1-Score of 1 for most malware classes and an average score of 0.93 for any malware class. While gaining performance in high-risk malware, the performance of low-risk malware slightly drops due to aggregation. On the other hand, aggregating the correct objective class of the sample in SUNDEW_Multi incurs an aggregation loss of 7.86%. This is because the evaluation considers a detection successful only if the ensemble predicts the actual class or a riskier class for the test sample. Hence, though PUA is successfully detected, its measure in SUNDEW_Multi drops as the riskier class Deceptor is chosen when resolving conflicting predictions during aggregation.

Resilience to noise. We next evaluate SUNDEW under varying noise infiltration induced by system load. We use the number of processes in the system to measure noise. To generate data for the experiment, we run benign applications from CNET [44] in multiples of 10 in the background while running the malware programs and collect the corresponding data at network, OS, and hardware. While these benign applications represent use-case scenarios, an extensive characterization covering a wide range of system loads is planned for future work. Figure 13a plots the F1-Score of specialized predictors in network and OS under varying system load conditions. As evident, the performance of the network component decrease, while the OS component is agnostic to system load.

We next compare the resilience of SUNDEW using the case of cryptominer, which is best detected in hardware. Figure 13b plots the F1-Score of the cryptominer-specialized predictors based on network, OS, and hardware and SUNDEW on cryptominer data collected under varying system load conditions. As evident, the performance of hardware-based predictor though higher than OS and network at lower system loads, decreases significantly as load increases. In contrast, the OS-based predictor, agnostic to system load, outperforms both the network and hardware-based predictors as soon as more than 10 additional user applications start executing simultaneously. Hence, the OS-based predictor is the most resilient to noise. In contrast, SUNDEW leverages the best of three worlds to achieve accurate and resilient malware detection (claims C-1 and C-2 in Section 3). At lower system loads, SUNDEW prioritizes network and hardware components for higher accuracy, whereas, at higher system loads, it uses the reliable OS component for prediction.

Comparison with prior art. Table 5 compares SUNDEW against our implementation of prior state-of-the-art predictors including single classifiers that rely on a single data source (network [2], operating system [42], or hardware [20]); and, same-input ensembles that do not employ class-wise specialization [23]. We compare these works based on the detection F1-Score and false-positive rate observed on the RaDaR dataset (Table 4). The cross-dimensional view of malware activity, specialization, and insightful aggregation of predictions in SUNDEW improve the detection F1-Score by at least 10% as compared to these prior works (Table 5A). Similarly, the class-specific specialization in SUNDEW decreases the false-positive by at-least 89% as compared to the prior works (Table 5B). Finally, with the incorporation of different data sources, we observe that SUNDEW is as resilient as the state-of-the-art OS-based works, even under noisy conditions.

Overheads. We next evaluate the overheads of SUNDEW considering an example deployment in an enterprise network. To measure the overheads, we first present the design and workflow of SUNDEW in the deployment in Figure 14. SUNDEW runs as a service on a middle-box server in the enterprise network, whereas the host machines run the client agents that enable the hosts to access the service to test any program. While the client agents collect the OS and hardware trails of the program under test, the gateway in the network collects the network behavior. Figure 14b illustrates the workflow of SUNDEW when a host accesses
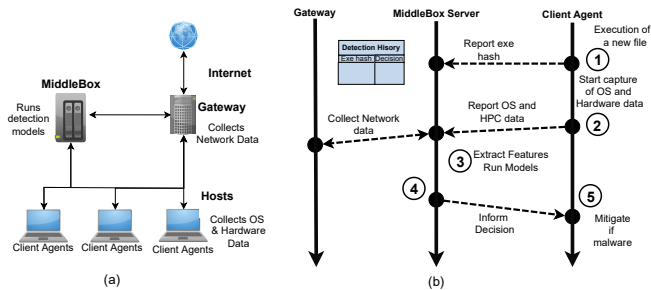
Fig. 14: (a) An example deployment in an enterprise network. The network data is collected at the gateway, whereas the OS and hardware data are collected at the host machines. The middlebox server runs the SUNDEW framework. (b) The workflow of SUNDEW. .

its service. At the host machine, whenever a new program executes, the client agent reports the hash of the program to the server (Step 1 in Figure 14b). To avoid repeated testing of the same program, SUNDEW maintains a detection history of program hashes at the server. Thus, if the program is not analyzed previously, SUNDEW starts the capture of respective logs at the host, as well as the gateway for 2 minutes (Step 2). After the duration, the server collects the OS and hardware data from the host, and the network data from the gateway (Step 3). The server then extracts the features from each data source, invokes the respective components of SUNDEW to predict the class of the program, and informs the final prediction to the host (Step 4). Finally, the client agent at the host takes the necessary action based on the prediction.

We use GeekBench [45] tool and observe that end-hosts incur an average overhead of 1.5% at the first execution of a test program. Note that the end-hosts (client-agents) are responsible only for the collection of OS and hardware data, whereas the heavy-weight operations of feature extraction and specialized predictors run on the middle-box server (refer to Figure 14a). While the middle-box server would require a dedicated provision of resources, the impact on users is minimal (1.5%) and is restricted to the execution of new applications alone.

8 DISCUSSION

In this section, we first discuss the applications of SUNDEW. Next, we discuss its limitations and present plausible directions for future work.

Applications. The multi-featured approach and aggregation in SUNDEW can serve as an *analysis framework* for anti-virus companies and *defense solutions* for securing enterprises. As an analysis framework, the holistic view and specialization enable precise characterization of samples, thus reducing the manual efforts to label thousands of newer samples reported daily. Alternatively, SUNDEW can serve as defense solutions in enterprise networks to provide accurate and resilient detection of malware attacks.

Incremental update of predictors. The SUNDEW ensemble involves predictors specialized for a set of malware classes. Further, aggregator functions are customized based on statistics from these predictors. With malware behavior evolving, the specialized predictors and aggregators would require updates. While mechanisms for incremental updates need to be explored in the future, we propose an auto-configuration engine that auto-configures the SUNDEW en-

semble for any update or any deployment setting. Such an engine takes as inputs the labeled data from the three data sources and the user requirements per malware class. It outputs the ensemble, including its specialized predictors and aggregator functions.

Scalability. With a rampant increase in newly reported malware classes, the number of specialized predictors is bound to increase 3x (one for each data source), increasing the complexity of aggregator functions and overheads. Hence, specialized predictors for each class can get infeasible. A viable solution is to club models that share common features and user requirements in the 3-tuple to reduce the number of specialized predictors for each data source. We intend to build an automated framework to configure SUNDEW with an optimal number of specialized predictors in future work. Alternatively, Locality Sensitive Hashing (LSH) can assist in identifying the similarity of test programs to previously tested program hashes. Accordingly, LSH can assist in enabling only the relevant specialized predictor or data components to decrease overheads.

Extensive Characterization of Noise. We analyze the impact of noise on SUNDEW using well-known benign applications. However, an extensive characterization of varying system load conditions and impact on the three data sources is planned for future work.

9 CONCLUSION

In this paper, we emphasize that malware classes are inherently different, and catering to the differences can improve the efficiency and resilience of detection. We propose SUNDEW, a novel multi-input ensemble of predictors and aggregator functions that leverages a multi-dimensional view of malware execution, considering its activities at the network, OS, and hardware and the system noise to provide a case-sensitive prediction. Our evaluations of SUNDEW on a real-world dataset indicate that the multi-dimensional view and specialization enable SUNDEW to avert infiltrating noise into the behavioral data while improving the accuracy, resilience, and false-positive guarantees. To the best of our knowledge, SUNDEW is the first to provide a multi-dimensional case-sensitive characterization of malware. The holistic approach and aggregation strategies open new avenues for malware research and detection models.

REFERENCES

[1] "AVTest: Malware Stastics." Accessed: 2021-09-26.

[2] K. Bartos, M. Sofka, and V. Franc, "Optimized invariant representation of network traffic for detecting unseen malware variants," in *25th USENIX Security Symposium*, pp. 807–822, 2016.

[3] X. Wang and R. Karri, "NumChecker: detecting kernel control-flow modifying rootkits by using hardware performance counters," in *The 50th Annual Design Automation Conference, DAC*, 2013.

[4] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Computing*, vol. 20, no. 1, pp. 343–357, 2016.

[5] M. S. Alam and S. T. Vuong, "Random forest classification for detecting android malware," in *IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing*, pp. 663–669, 2013.

[6] B. Anderson and D. McGrew, "Machine learning for encrypted malware traffic classification: accounting for noisy labels and non-stationarity," in *23rd ACM SIGKDD International Conference on knowledge discovery and data mining*, pp. 1723–1732, 2017.

[7] N. Nissim, R. Moskovitch, L. Rokach, and Y. Elovici, "Novel active learning methods for enhanced pc malware detection in windows os," *Expert Systems with Applications*, vol. 41, pp. 5843–5857, 2014.

[8] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. J. Stolfo, "On the feasibility of online malware detection with performance counters," in *The 40th Annual International Symposium on Computer Architecture, ISCA*, 2013.

[9] H. Peng, J. Wei, and W. Guo, "Micro-architectural features for malware detection," in *11th Conference on Advanced Computer Architecture, ACA* (J. Wu and L. Li, eds.), Springer, 2016.

[10] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised Anomaly-Based Malware Detection Using Hardware Features," in *17th International Symposium on Research in Attacks, Intrusions and Defenses, RAID*, pp. 109–129, 2014.

[11] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Network and Distributed System Security Symposium, NDSS*, 2009.

[12] M. R. Watson, A. K. Marnerides, A. Mauthe, D. Hutchison, *et al.*, "Malware detection in cloud computing infrastructures," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, 2015.

[13] Y. Zhou, G. Cheng, S. Jiang, and M. Dai, "Building an efficient intrusion detection system based on feature selection and ensemble classifier," *Computer networks*, vol. 174, p. 107247, 2020.

[14] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dl-droid: Deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, p. 101663, 2020.

[15] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection," in *17th USENIX Security Symposium*, pp. 139–154, USENIX Association, 2008.

[16] R. Perdisci, W. Lee, and N. Feamster, "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, 2010.

[17] P. Feng, J. Ma, C. Sun, X. Xu, and Y. Ma, "A novel dynamic android malware detection system with ensemble learning," *IEEE Access*, vol. 6, pp. 30996–31011, 2018.

[18] D. Li, Q. Li, Y. Ye, and S. Xu, "A framework for enhancing deep neural networks against adversarial malware," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 1, pp. 736–750, 2021.

[19] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Trans. Inf. Forensics Secur.*, 2016.

[20] H. Sayadi, N. Patel, S. M. P. D., A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: a comprehensive analysis and classification," in *Proceedings of the 55th Annual Design Automation Conference, DAC*, pp. 1:1–1:6, ACM, 2018.

[21] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *computers & security*, vol. 77, pp. 578–594, 2018.

[22] F. Salo, A. B. Nassif, and A. Essex, "Dimensionality reduction with ig-pca and ensemble classifier for network intrusion detection," *Computer Networks*, vol. 148, pp. 164–175, 2019.

[23] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian, "EC2: ensemble clustering and classification for predicting android malware families," *IEEE Trans. Dependable Secur. Comput.*, vol. 17, 2020.

[24] S. Wang, Q. Yan, Z. Chen, B. Yang, C. Zhao, and M. Conti, "Detecting android malware leveraging text semantics of network flows," *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 5, 2018.

[25] P. M. Comar, L. Liu, S. Saha, P.-N. Tan, and A. Nucci, "Combining supervised and unsupervised learning for zero-day malware detection," in *2013 Proceedings IEEE INFOCOM*, pp. 2022–2030, 2013.

[26] M. B. Bahador, M. Abadi, and A. Tajoddin, "HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition," *4th International Conference on Computer and Knowledge Engineering (ICCKE)*, pp. 703–708, 2014.

[27] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware Performance Counter-Based Malware Identification and Detection with Adaptive Compressive Sensing," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, 2016.

[28] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *International Symposium on Software Testing and Analysis, ISSTA*, pp. 122–132, ACM, 2012.

[29] "Types of Malware: Kasperkey Solutions ." Accessed: 2021-09-26.

[30] M. Alam, S. Bhattacharya, S. Dutta, S. Sinha, D. Mukhopadhyay, and A. Chattopadhyay, "Ratafia: ransomware analysis using time and frequency informed autoencoders," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[31] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, August 2007.

[32] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 20–38, IEEE, 2019.

[33] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "RHMD: Evasion-Resilient Hardware Malware Detectorss," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 315–327, 2017.

[34] JoeSandbox, "Automated malware analysis." Accessed: 2021-08.

[35] D. R. Miller, *Security information and event management (SIEM) implementation*. McGraw-Hill Higher Education, 2011.

[36] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multi-modal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[37] J. Yan, Y. Qi, and Q. Rao, "Detecting malware with an ensemble method based on deep neural network," *Security and Communication Networks*, vol. 2018, 2018.

[38] S. Karapoola, N. Singh, C. Rebeiro, and K. Veezhinathan, "RaDaR: A real-world dataset for AI powered run-time detection of cyber-attacks," in *The 31st ACM International Conference on Information and Knowledge Management (CIKM)*, ACM, 2022.

[39] S. Karapoola, N. Singh, C. Rebeiro, and K. Veezhinathan, "JU-GAAD: Comprehensive malware behavior-as-a-service," in *Cyber Security Experimentation and Test Workshop*, pp. 39–48, 2022.

[40] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti, "Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes," in *28th Annual Network and Distributed System Security Symposium, NDSS*, The Internet Society, 2021.

[41] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2009.

[42] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *IEEE Trans. Inf. Forensics Secur.*, 2016.

[43] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time," in *28th USENIX Security Symposium*, pp. 729–746, 2019.

[44] "CNET: Windows application repository." Accessed: 2021-09-26.

[45] "Introducing Geekbench-5." Accessed: 2021-09-26.

Sareena Karapoola is a Ph.D. scholar at the Department of Computer Science and Engineering, Indian Institute of Technology, Madras. Her research interests include the cyber-security, malware analysis and detection, machine learning for security, development of novel attack mitigation strategies, and testbeds for security research.

Nikhilesh Singh is a Ph.D. student at Department of Computer Science and Engineering, Indian Institute of Technology, Madras. His research interests include the deployment of Machine Learning for safety and system security including malware defenses, micro-architectural security, operating system security, secure hardware designs.

Chester Rebeiro is an Associate Professor at the Department of Computer Science and Engineering, Indian Institute of Technology, Madras. His research interests include hardware security, operating systems security, and applied cryptography. He also leads the effort at designing secure RISC-V micro-processors for embedded platforms at IIT Madras.

Kamakoti V. is a Professor and the Director of IIT Madras. He specializes in the areas of computer architecture, secure systems engineering, and network security and privacy. He is a coordinator of the Information Security Education and Awareness program of the Department of Information Technology, Government of India and the Chairman of the Task Force on Artificial Intelligence for India's Economic Transformation. He has also won several awards such as the IBM Faculty Award (2016).