Hawkware: Network Intrusion Detection based on Behavior Analysis with ANNs on an IoT Device

1st Sunwoo Ahn
ECE and ISRC
Seoul National University
Seoul, South Korea
swahn@sor.snu.ac.kr

2nd Hayoon Yi
ECE and ISRC
Seoul National University
Seoul, South Korea
hyyi@sor.snu.ac.kr

3rd Younghan Lee
ECE and ISRC
Seoul National University
Seoul, South Korea
yhlee@sor.snu.ac.kr

4th Whoi Ree Ha
ECE and ISRC
Seoul National University
Seoul, South Korea
wrha@sor.snu.ac.kr

5th Giyeol Kim
ECE and ISRC
Seoul National University
Seoul, South Korea
gykim@sor.snu.ac.kr

6th Yunheung Paek
ECE and ISRC
Seoul National University
Seoul, South Korea
ypaek@snu.ac.kr

***Abstract*—The *network-based Intrusion detection system* (NIDS) plays a key role in Internet of Things (IoT) as most IoT services are network-driven. However, the existing NIDSes for IoT systems are either too costly to scale or vulnerable against advanced attacks such as traffic mimicry. In this paper, we propose a novel IDS named *Hawkware*, a lightweight ANN-based distributed NIDS that runs on an IoT device and analyzes the device's runtime behavior in tandem with its network traffic. By analyzing device behavior, Hawkware is able to replace expensive, deep data analysis that has traditionally been used to detect advanced attacks. Our evaluations show that Hawkware is lightweight enough to be distributed and deployed on a Raspberry PI, and yet capable of detecting such attacks at a satisfactory level.**

***Index Terms*—Internet of Things, Security, Network Intrusion Detection, Artificial Neural Networks**

I. Introduction

A variety of services have been proposed using Internet of Things (IoT) devices. IoT devices have been one of the frequent targets for adversaries because they are generally cheap with a lack of security awareness. One of the security mechanisms is the intrusion detection systems (IDSes) for detecting on-going intrusions in such IoT networks. There are mainly two types of IDS: *network-based intrusion detection system* (NIDS) and *host-based intrusion detection system* (HIDS). Unlike HIDS which is specialized in detecting attacks within a single device, NIDS analyzes data traffic in a networked system to discover the existence of attacks. NIDS has been preferred by researchers for IoT security [1], [10], [11] because an IoT system can be viewed not as a standalone computing device but as a cluster of devices networked to form an ecosystem.

There are two deployment strategies for deploying NIDS: *centralized* and *distributed*. In a distributed manner, NIDS is distributed over multiple strategic points such as routers or gateways instead of sitting at a single centralized point. As suggested by the latest study [1], it is more reasonable to use a distributed strategy that can detect not only malicious traffic from/towards outside but also malicious traffic traversing inside the IoT system. NIDS typically examines the header of a network packet to glean information about the network transaction. It can succeed in identifying naive and common network-driven attacks to some degree only by merely analyzing the header information. However, it is not as quite effective to the advanced attacks [6], [13], which normally necessitates deeper analysis of the packet main body (i.e., *payload*).

Since a packet payload contains the actual data contents involved in a network transaction, NIDS may have a better chance to disclose the hidden maliciousness of a packet that may be propagated and inflicted over the network. For example, a payload inspection, also known as *deep packet inspection* (DPI), would enable NIDS to reveal elaborated attempts of adversaries to deliver malicious payloads undetected by crafting packet headers to appear innocuous. However, there is a potential limitation of DPI that the inspection itself is impossible in the middle of delivery when the payload is encrypted for the intended user at the end device. More importantly, there is also a critical downside of DPI that it is not always straightforward to represent packet payloads in a structured format, thus requiring a heavy computation for extensive, real-time data analysis.

To resolve these issues, we propose, *Hawkware*, our lightweight ANN-based distributed NIDS that detects attacks on a device without actual data analysis for DPI, yet attaining better accuracy than latest NIDS [1] for IoT devices. Since resource consumption is a primary concern of every embedded device in an IoT system, efficiency must be of top priority for any techniques targeting most embedded devices with strict resource constraints. For maximum efficiency with little loss of accuracy, Hawkware monitors the device's runtime behavior during its process of network transactions in addition to the basic inspection on the packet header. In principle, Hawkware replaces DPI with device behavior monitoring. The rationale behind this strategy of Hawkware is that in a network transaction, the device usually acts and reacts according to the payload data because the payload basically conveys the message or instructions for the device from/toward the outside. The behavior analysis has an advantage over the payload data analysis as the runtime behavior is relatively easy to be represented in structured formats (e.g., branch sequences and call graphs), which expedites and facilitates the analysis process.

Running on a host IoT device, Hawkware monitors all its behaviors relevant to the computations driven by network transactions with the outside. To model relations within network and device behavior, we opted for an *artificial neural network* (ANN) model that can analyze complex non-linear data. Given as inputs to the ANN model, the runtime behaviors are structured into sequential formats, such as network packet sequences or system call sequences. To effectively process the sequentialized input data, our ANN is based on *long short-term memory* (LSTM) *recurrent neural networks* (RNNs) [14] which have demonstrated an excellent accuracy in analyzing sequential data. Unfortunately, existing LSTM based ANNs, which mostly accept a single sequence of inputs, are not suitable for our task. Therefore, we designed Hawknet, LSTM based ANN, to correlate and analyze both sequences of device and network behavior in its task to detect network intrusions.

Generally, complex analysis with heavy computations would require an abundant computing resource that is not available in a typical IoT device. Therefore in our implementation, we optimized our Hawknet such that Hawkware can perform intrusion detection with minimal resource consumption. We applied ANN weight quantization to reduce the memory pressure of complex computations [15]. The experiments with our Hawkware implementation are encouraging in that we were able to gain up to 30x speedup for Hawknet while achieving almost the same detection accuracy as without the optimizations. To evaluate Hawkware, we implemented our prototype on a Raspberry PI, which is an ARM-based single-board computer that has similar computing resources as smart IoT devices. One objective of our work is to demonstrate the effectiveness of SIMD capabilities in a state-of-the-art processor for ANN-based NIDS. For this, Hawkware engages ARM's NEON SIMD engine because the engine has an excellent parallel processing capability that can accommodate the high degree of parallelism inherent in our Hawknet model. We measured the performance enhanced by utilizing the SIMD engine and achieved about 66x speedup.

II. Related Work

Learning-based IDS: Anomaly-based network intrusion detection can be classified into mainly three techniques: statistical-based, knowledge-based and machine learning-based according to the work [3], [4]. There have already been various approaches proposed to develop learning-based NIDSes [3], [4]. Most of them were designed to work with KDD-99 or NSL-KDD dataset, which contains features collected without examining the packet payload. As discussed earlier, however, relying on such a shallow analysis with incomplete information will leave NIDS blind to more elaborated attempts for attacks [6], [13]. Therefore, a group of subsequent studies has made efforts to counter these advanced attacks by enhancing the accuracy of NIDS with help of DPI [2], [8], [9]. All the above-mentioned NIDS solutions differ from ours since they are mainly interested in detection accuracy, and relatively indifferent to resource requirements for computing their proposed solutions.

Lightweight IDS for IoT: There have been previous attempts on NIDS to reduce resource consumption in IoT systems. The previous work [5], [12] proposed NIDSes which were designed to detect bots without DPI. By confining their task to the bot detection, they were able to attain fairly high detection accuracy even without using DPI, but their systems cannot cope with other types of network attacks. In [7], they proposed PAYL, a lightweight NIDS that identifies anomalies in packet payloads. One of its problems is that it oversimplifies the representation of packet payload data and loses detection accuracy as indicated by [1].

Distributed NIDS: From the observation on previous research in centralized NIDS, we can see that there has always been a tug of war between detection accuracy and computational efficiency. In fact, the same war still continues when researchers veer their efforts toward distributed NIDS as the importance of IoT security increases. The approaches of recent research on distributed NIDS are largely two-fold. In one approach, researchers proposed NIDSes that can be distributed and deployed in local *fog computing* servers for IoT systems [10], [11]. In the other, researchers proposed, NIDSes such as *Kitsune* [1], that can be distributed on simple routers. The former is different from ours because they assume that the local machine housing each of the distributed NIDSes is full of computing power and resource. In [1], it was argued that the latter has the advantage

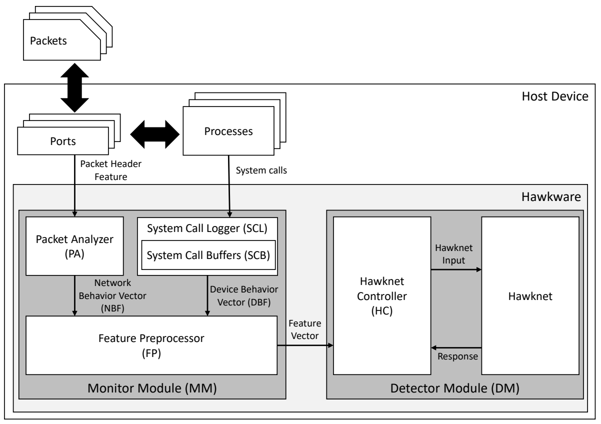

Fig. 1. Hawkware architecture overview

of being economically scalable. However, the latter approach also has a challenging problem to address; the devices hosting NIDS are subject to stringent resource constraints. Like Hawkware, Kitsune was designed to be lightweight enough to efficiently operate in resource-restricted environments. However, as Kitsune only examines network behavior features similar to those available in the KDD99 dataset, it would exhibit the same vulnerability discussed above.

III. Hawkware Design

A. Overview

The overall architecture of Hawkware is depicted in Figure 1. Hawkware is divided into two main modules: the monitor module (MM) and the detection module (DM). MM monitors both the network and device behaviors and extracts the relevant features for the ANN named as *Hawknet*. DM detects any suspicious behaviors indicating network intrusions by utilizing Hawknet.

Hawkware consists of five components and their functions are summarized as follows: (1) the *packet analyzer* (PA) analyzes network packet headers and extracts relevant features; (2) the *system call logger* (SCL) records the device behavior and extracts features related to incoming/outgoing network packets; (3) the *feature preprocessor* (FP) aggregates both extracted features and transfer them as inputs to Hawknet; (4) the *Hawknet controller* (HC) examines the Hawknet's outputs and determines the existence of intrusions; (5) the *Hawknet* quantifies the *degree of anomaly*;

B. Threat models and assumptions

Our assumptions in the design of Hawkware are as follows. We assume that (1) Hawkware resides in an uncompromised OS kernel. This implies that adversaries cannot directly tamper with the code of Hawkware or its supporting kernel modules; (2) the W$\oplus$X security protection is enforced, hence preventing adversaries from directly running their code by modifying existing programs; (3) local attacks on low-end IoT devices are relatively rare because IoT services are primarily *network-driven*, and thus for most of the time, the devices act or react according to network transactions. Therefore, Hawkware was designed to detect attacks that result from/in network transactions and does not intend to cope with local attacks whose origins are from the inside of the device such as those directly injected into the device via an I/O interface or those living inside the device from the start;

C. Monitor Module

For successful monitoring, it is crucial to properly represent the raw behavioral data by refining it into feature vectors for DM. Since such feature vectors, generated by MM, are the only information that

TABLE I
FEATURE VARIABLES AND PROCESSING METHOD IN HAWKWARE

Feature variable	Preprocessing method
timestamp	timestamp difference between current and previous packet
payload size	no transformation
network protocol	one-hot encoded format
remote IP	classify as special-use address
remote port	output vector from embedding layer
host port	output vector from embedding layer
system call sequence	one-hot encoded format

DM can use, irrelevant or redundant features will only increase the computation complexity, let alone hinder anomaly detection accuracy. Therefore, Hawkware incorporates two sets of feature vectors, respectively called the *network behavior feature vectors* (NBFVs) and the *device behavior feature vectors* (DBFVs) as explained below.

1) Packet Analyzer (PA): For every outgoing/incoming network packet, PA in MM captures the raw network packets arriving at the device where Hawkware is operating. Since a packet header contains necessary features for discerning common network behavior, it is examined by Hawkware to determine network behavior in the same way as done by other NIDSes. An NBFV is a collection of features extracted by PA from the packet headers. It consists of six elements as described in Table I. An NBFV is delivered to FP for further processing as described in Section III-C3 in detail.

2) System Call Logger (SCL): SCL in MM records the system calls invoked only by the network-driven processes, which include listening to host ports, and their child processes. This proposed tactic is reasonable for three intentions. First, as most IoT services are network-driven, we can deduce that the primary pathway for the attack would be through the network-driven processes. This, in turn, clinches the fact that only examining the behaviors of such processes is sufficient to protect the device by perceiving the first sign of intrusion. Second, it eliminates the overhead of deep analysis of packet payload (e.g., DPI) while retaining high detection accuracy. DPI is sometimes necessary for NIDS to detect advanced and deceptive attacks. However, such analysis is too heavy for low-cost devices to be implemented. Therefore, Hawkware acquire the necessary information from an alternative source that is a chain of system call sequences, which has been proven by earlier work [16], [18] as an excellent feature representing device's software behavior. Last of all, examining and storing system call sequences for each and every process would strain the resources of an IoT device. To handle this issue, SCL maintains separate ring buffers of fixed size, called *system call buffers* (SCBs), each of which is used to remember the most recent system call sequence of a different process. When a system call is invoked, SCL checks the process ID (PID) of the caller process and records the call into an appropriate SCB. A DBFV is a collection of features consisting of system calls stored in SCB. Similar to NBFVs, DBFVs are delivered to FP for further processing.

3) Feature Preprocessor (FP): For each outgoing/incoming network packet, DM correlates network and device behaviors by pairing NBFV to its relevant DBFV. These two vectors are refined by FP before being delivered finally to DM. FP performs in four steps as follows: (1) FP takes an NBFV and identifies the host port number in the NBFV and the process associated with that port number; (2) FP fetches the DBFV corresponding to the process; (3) The DBFV are transformed into a vector of one-hot encoded vectors (i.e., *device behavior feature matrix* (DBFM)) for the Hawknet in DM; and (4) The correlated feature vectors are combined and delivered to the Hawknet through HC in DM.

In the 2nd step, FP fetches a DBFVs from SCB differently for incoming network packets and outgoing ones. Note that regardless

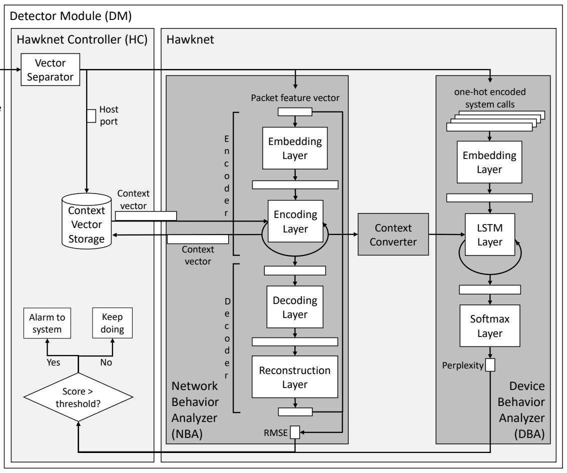

Fig. 2. Architecture of Hawkware's detector module (DM)

of the existence of incoming/outgoing packets, SCL keeps storing system calls into SCBs. For an incoming packet, as soon as it arrives, the SCB of corresponding processes related to the packet is flushed out and SCL will refill the SCB with new system call data. Once the SCB is full, the DBFV is formed by SCL with the contents in it. This DBFV represents how a process (or processes) in the device react(s) to newly arrived incoming packets (i.e., the device behavior). For an outgoing packet, once it is transmitted, the contents in its relevant SCB were transformed to the DBFV by SCL. The DBFV at this moment summarizes the device behavior for creating an outgoing packet.

D. Detector Module

As depicted in Figure 2, at the core of DM lies HC that governs the operations of Hawknet. HC supplies Hawknet with two inputs, NBFV and DBFM, processed by FP. Then, Hawknet measures the degree of anomaly of the current network transaction represented by the inputs. HC determines the existence of network attacks on the device by checking if the measured value surpasses the predefined threshold. To do this, HC interacts with the three ANN models of Hawknet: network behavior analyzer (NBA), device behavior analyzer (DBA) and context converter (CC).

1) Network Behavior Analyzer (NBA): The augmented sequence of NBA vectors is delivered to NBA as two separate vectors: the new NBFV and *context vector*. The latter is a condensed realization of a sequence of the earlier NBFVs that share the same port with the new vector. To accomplish this, NBA has an autoencoder which tries to produce the same output as the given input values. NBA is trained with the NBFVs that represent the *normal* behavior in a way that autoencoder tries to restore the original vectors by decoding the encoded ones. NBA will fail to restore the input vectors extracted from anomalous (thus unfamiliar) network behavior. Therefore, by leveraging this property, NBA measures the degree of anomaly through calculating the root mean square error (RMSE) between the input NBFV and the vector reconstructed by NBA's autoencoder.

As network communication typically entails transmission of multiple network packets with the same party, NBA must be able to identify all those related packets. To enable this, we design NBA to incorporate an LSTM RNN layer, which exhibits excellent accuracy in analyzing sequential data. In a conventional design [17], the layer should have been placed in both encoder and decoder components. However, this design is unsuitable for NBA as the input vectors are not fixed, but rather in a form of streams of data. When a network packet arrives in the device, NIDS should infer the degree of anomaly to detect intrusion as fast as possible. Therefore, we place an LSTM layer only in the encoder and employ a simple fully connected layer for the decoder to calculate RMSE swiftly. The encoder encodes both the given NBFV and the past history of NBFVs, while decoder tries to reconstruct only the current NBFV to calculate RMSE. In addition, whenever a new NBFV is delievered to NBA, HC offers the corresponding context vector that represents the network behavior history of a single network port.

As depicted in Figure 2, NBA has four ANN layers: the embedding layer, encoding layer, decoding layer and reconstruction layer. The embedding layer converts the NBFV into a new vector of a higher dimension space to express various relations between the features within the input vector.This vector is then compressed to an encoded vector by the encoding layer.Note that, during this process, a newly updated context vector is created to replace its predecessor in HC's context vector database for future use. The encoded vector is then taken as input by the decoder and reconstruction layer reconstructs the original NBFV. With this reconstructed vector and the original NBFV, RMSE is calculated by NBA.

2) Context Converter (CC): In order to include the network context in its analysis of device behavior, DBA requires the information about the network behavior history pertaining to its input DBFM. NBA maintains this contextual information in the encoding layer as a context vector. However, the values in the vector would not directly constitute the network context because they are meant to represent the information needed to encode future NBFVs. Thus, this context vector is taken as input by CC that is trained to extract the network behavior history embedded in the vector and generate a new *history vector* representing the current network context. Lastly, within the network context represented by the history vector, DBA can analyze a DBFM after setting its initial state. The only objective of CC is to perform a vector conversion, and thus we have implemented CC with a simple fully-connected layer.

3) Device Behavior Analyzer (DBA): As we just mentioned, in order to discern device behavior anomaly from a DBFM, DBA first makes use of the history vector to set the initial network context. To relate contextual information between the sequential data, DBA also employs an LSTM layer. DBA starts consuming the system call vectors delivered by HC as soon as CC delivers the converted history vector. Just before entering the LSTM layer, the system call vectors are projected into a continuous vector space by the embedding layer which helps the LSTM layer to better recognize the data distribution within the vector. Now, this projected vector goes into the LSTM layer which is initialized with the history vector from CC. Finally, the output vector of the LSTM layer is fed into the softmax layer which calculates the probability of each system call occurring next in the sequence. With this probability, DBA quantifies the degree of anomaly by calculating its perplexity, the negative log-likelihood of the system call's occurrence. The calculated perplexity for each system call vector is given to HC.

IV. Evaluation

Implementation: We have implemented a prototype of Hawkware on a Raspberry Pi 3 Model B+ board which has a 1.4 GHz quad-core ARM Cortex-A53 processor with 1 GB RAM as it resembles many ARM-based IoT devices. We bound Hawkware to a single core for its computation with a 32 bit Linux OS.

We incorporated Tshark[1], a network packet capturing and analyzing tool, in implementing PA and used ftrace, an event tracing framework available in Linux kernels, in SCL. FP and HC are implemented in Python. Hawknet is trained offline on a separate server and then deployed on devices to perform detection. Hawknet and its training code are implemented with Tensorflow[2], which is one of the most popular frameworks for machine learning.

[1]https://www.wireshark.org/docs/man-pages/tshark.html

TABLE II
The number of network packets and system calls in the dataset

	Network packets	System calls
Train data	214967	2148760
Validation data	143312	1432506
Benign test data	875242	9722007
Malicious test data	7540	63649

However, directly deploying this model strains IoT devices. In order to mitigate this issue, we first leveraged ARM's NEON SIMD instructions to accommodate the high degree of parallelism inherent in Hawknet. Unfortunately, due to the high memory pressure in ANN computation for loading its weight values, utilizing NEON alone still falls short of making Hawknet efficient enough for IoT devices. Therefore, in addition, we capitalized on ANN weight quantization [15], compressing the vector values of Hawknet from 32-bit floating point numbers to 8-bit fixed point numbers. The compressed model of Hawknet, generated by employing the Tensorflowlite, only occupies 60KB. The learning rate was set to 0.001, which is a standard starting point for typical deep learning. The number of parameters in each layer of Hawknet is set as following: NBA's embedding layer, encoding layer, decoding layer and reconstruction layer each respectively have 297, 3840, 567 and 297 parameters, DBA's embedding layer, LSTM layer and softmax layer each have 3160, 840 and 3476 parameters and there are 210 parameters for CC.

Training Hawknet: In order to facilitate the faster training of Hawknet, we collected benign data on the Raspberry Pi and then trained the model offline on a more powerful server. To obtain benign data, we ran, on the device, various network-based processes installed by default on the Raspbian OS with different arguments (e.g., OpenSSH, web browser, update-manager, ping, etc.). Note that we leveraged MM to collect device behavior data alongside with network behavior data. From the collected data we randomly select 40% of the data to train the model while using the rest to validate the trained model. The exact number of network packets and system calls in our dataset is described in Table II.

Detection accuracy: We first evaluate the comprehensive detection accuracy of Hawkware against attacks commonly found in IoT devices. We test Hawkware against attack samples of three classes of malware gathered from VirusTotal[3]: DDoS botnets (DDoS) attacks, bitcoin miners (Miner) and backdoors (Backdoor). Though Hawkware would be capable of detecting any unusual method of sending these malware over the network, we simply run their code directly on the device as we assume the malware can be introduced into the system through legitimate channels by employing attack methods such as social engineering. This is done to evaluate whether, even in such a case, Hawkware would be capable of detecting the malware.

In Figure 3, to facilitate a systematic comparison, alongside the receiver operating characteristic (ROC) curves for Hawknet and Kitsune [1], a state-of-the-art NIDS for low-cost IoT devices, we also display ROC curves for several variants of Hawknet: Hawknet with quantization (Hawk-Q), NBA of Hawknet (NBA-only) and DBA of Hawknet (DBA-only). We train and evaluate Kitsune on the same data

[2]https://www.tensorflow.org
[3]https://www.virustotal.com

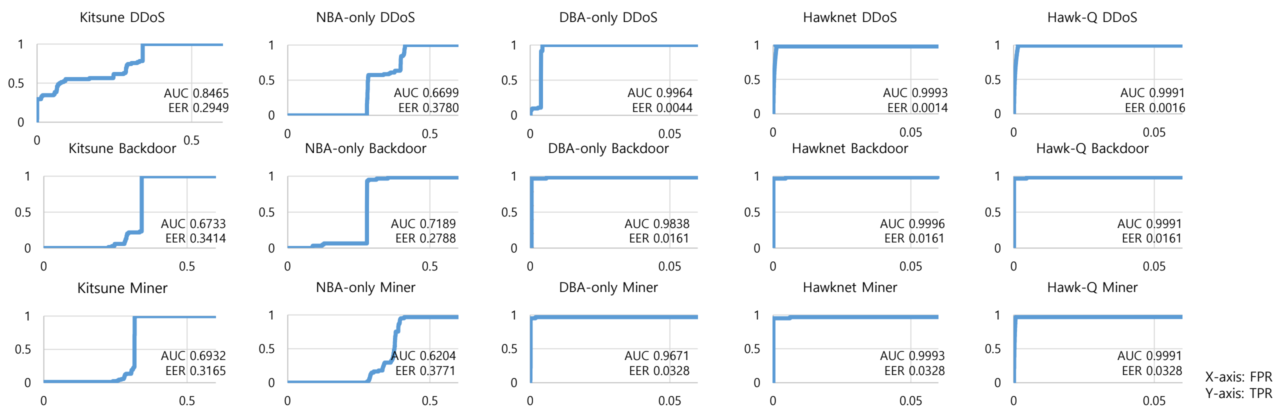

Fig. 3. Detection accuracy of Kitsune, Hawknet and its variations given as the receiver operating characteristic (ROC) curve alongside the area under curve (AUC) and equal error rate (EER)

that our models use. NBA-only and DBA-only are trained separately from Hawknet with only network behavior and NBA (NBA-only) or device behavior and DBA (DBA-only). The ROC curves plot the true positive rate (TPR) against the false positive rate (FPR) over various detection thresholds of the models. Instead of plotting the whole ROC curve, we magnify the left-side portion of the curve to better show the difference between the models. To provide a better understanding of the results, we also give the area under the ROC curve (AUC) and equal error rate (EER). EER is the false positive rate when the threshold is set to a value where the detection FPR is equal to the false negative rate (1-TPR) and therefore a lower EER value typically indicates better detection accuracy with lower false alarm rate. As Hawknet has two thresholds (one for NBA and another for DBA), to provide a clear comparison, we show the ROC curve for changing the NBA threshold while fixing the DBA threshold to the value when Hawknet shows optimal accuracy.

In Figure 4, we depict the degree of anomaly of each model with their mean value and standard error of the mean (SEM). Note that, though Hawknet has two degrees of anomaly, one for NBA and another for DBA, the NBA score is the same to that of NBA-only because Hawknet does not use additional information from the device in calculating network behavior anomaly. On the other hand, as DBA in Hawknet leverages network behavior context and therefore gives different values from that of DBA-only, we display the degree of anomaly of DBA for Hawknet and Hawk-Q. The mimicry attacks depicted in Figure 4 will be discussed below separately.

In Figure 3, NBA-only shows comparable accuracy to that of Kitsune as both examine similar network behavior data. Both perform poorly against *coinminer* and *backdoor* due to the fact that, as the main function of these malware run on the device and only the results are relayed via network, there is little visibility of the malware behavior over the network which makes it difficult for Kitsune and NBA to differentiate the behavior of the malware from around 30% of benign network-based processes. This is also supported by the SEM depicted in Figure 4 where a portion of benign and backdoor/miner would overlap in their degrees of anomaly. The high values of the average degree of anomaly for backdoor in NBA-only and DDoS in Kitsune are due to the fact that a portion of malware in these classes show high degrees of anomaly, which are reflected in the left-side portion of their respective ROC curves in Figure 3. On the other hand, DBA-only shows good accuracy against all three classes of malware. As these malware act as agents within the device and perform malicious activities on their own or as a response to messages

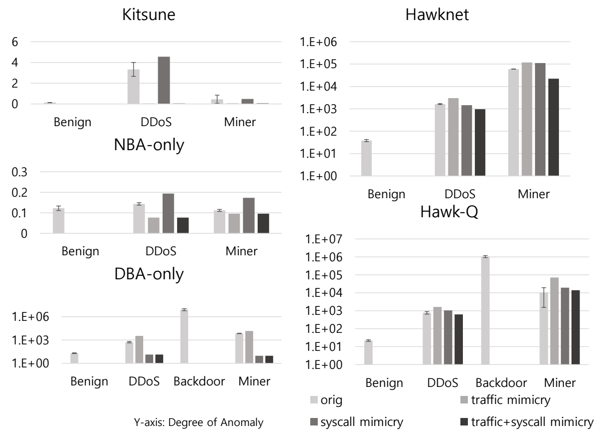

Fig. 4. Degrees of anomaly (RMSE for Kitsune and NBA-only, perplexity for the rest) of Kitsune, Hawknet and its variations

from external command & control servers, their behavior likely deviates from that of a typical network-based process. The figure also demonstrates the positive effect on the detection accuracy that Hawknet can obtain by leveraging both device behavior and network behavior in its detection task. Though the difference may seem little, when considering that recent works in NIDS strive to gain every little advantage in improving accuracy, the improved accuracy of Hawknet over DBA-only and NBA-only supports our initial assumption in that modeling device behavior is a good substitute for DPI in improving NIDS accuracy. It is also encouraging that even after quantization, Hawknet suffers little in its detection accuracy and thus would be deployable on IoT devices.

Against advanced attacks: The superiority of Hawkware comes from the fact that it can even detect advanced attacks adopting mimicry schemes. To justify the claim, we conducted another experiment using a *DDoS* and *Miner* malware which we modified to incorporate certain mimicry schemes. We generated three versions for each malware: (1) traffic mimicry [13], (2) system call mimicry [19] and (3) traffic and system call mimicry combined. The traffic mimicry scheme divides the adversarial packets into smaller sizes to mimic the size of normal network packets. The system call mimicry scheme pads the malware's runtime behavior with negligible system calls to mimic a system call sequence found in a normal device behavior.

As can be seen in Figure 4, the degree of anomaly calculated by Kitsune, NBA-only and DBA-only showed little difference from

TABLE III
AVERAGE PERFORMANCE OF EACH COMPONENT OF HAWKNET (IN CYCLES PER INPUT)

	NBA	CC	DBA	Hawknet
CPU	767,498	94,116.8	448,689	5,348,504.8
NEON	11,566.9	1,422.33	6,486.07	77,849.93
Quantization	203.034	108.962	372.120	4,033.196

those of benign activities, thus they are vulnerable to the mimicry attack schemes. On the other hand, Hawknet stayed resilient against the advanced attacks. Even when both mimicry attack schemes were applied, Hawknet can successfully detect the malware as the correlation between system call invocation and network behavior has not been mimicked. Though theoretically, it might be possible to craft a mimicry attack that can even bypass Hawkware, we believe Hawkware's resilience gives little leeway to adversaries in crafting such mimicry attacks and thus, makes it much more difficult to launch such attacks.

Runtime overhead: We compared the runtime overhead of Hawkware with Snort2 [2], one of the most popular NIDS using DPI, and Kitsune. To quantify the overheads, we ran sysbench[4] while executing each NIDS. We generated 300 network packets per second, which is a realistic settings for an IoT device. In this environment, Hawkware incurred 7.38% runtime overhead comparable to 5.03% of Kitsune. Snort2, showing 10.04 % overhead, required more computational load than the previous ones.

Detection Module performance: Table III shows the average number of CPU cycles required to process a single input for each component of Hawknet as well as the impact of each optimization applied in its implementation. Note that a single input of Hawknet is defined as an NBFV and 10 DBFVs. The results indicate that, for every network packet captured by PA, Hawknet would consume up to an average of 2.7 ms on a 1.4 GHz processor to fully process its related device behavior. Considering that the amount of network traffic directed towards a typical end-point IoT device is low, we believe that processing around 370 packets a second would be practical in the real world. To compare these results with Snort2 and Kitsune, we also measured their average number of CPU cycles required to process a single input. Snort2 and Kitsune takes 61,348,998.888 and 8,275,400 cycles respectively. Kitsune implemented in C++, not Python in which the open source of Kitsune is implemented, takes 82,754 cycles (the author claims 100x faster with C++ implementation) whereas Hawknet takes only 4,033.196 cycles.

V. CONCLUSION

Hawkware is an ANN-based NIDS that incorporates network and device behavior analysis for detecting attacks on IoT devices. Our evaluations have shown that Hawkware can successfully correlate network and device behaviors for network intrusion detection. Furthermore, we have shown that Hawkware is resilient against advanced attacks including techniques like traffic mimicry or system call mimicry via this correlation of behaviors. Our prototype on a Raspberry Pi has shown that, by capitalizing on ARM's NEON SIMD architecture and ANN weight quantization techniques, Hawkware's implementation can be efficient enough to be deployed on embedded devices in an IoT system while outperforming state-of-the-art lightweight NIDS for IoT [1] against network-based malware.

ACKNOWLEDGMENT

This work was partly supported by Institute of Information & communications Technology Planning & Evalution(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00230, Development on Autonomous Trust Enhancement Technology of IoT Device and Study on Adaptive IoT Security Open Architecture based on Global Standardization [TrusThingz Project]), SAMSUNG Research, Samsung Electronics Co.,Ltd and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478).

[4]https://launchpad.net/sysbench

REFERENCES

[1] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: an ensemble of autoencoders for online network intrusion detection," 2018 [25th Annual Network and Distributed System Security Symposium (NDSS'18)].
[2] J. Beale, A. R. Baker, B. Caswell, M. Poor, "Snort 2.1 Intrusion Detection," Syngress Publication, Rckland, MA, Second Edition, pp. 185-228, April 2004.
[3] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," computers & security, vol. 28, 1-2, pp. 18–28, 2009.
[4] A. L. Buczak, and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," IEEE Communications Surveys & Tutorials, vol. 18, 2, pp. 1153–1176, 2016.
[5] F. Tegeler, X. Fu, G. Vigna, and C. Kruegel, "Botfinder: Finding bots in network traffic without deep packet inspection," ACM, pp. 349–360, 2012 [Proceedings of the 8th international conference on Emerging networking experiments and technologies].
[6] T. H. Ptacek, and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," SECURE NETWORKS INC CALGARY ALBERTA, 1998.
[7] K. Wang, and S. J. Stolfo, "Anomalous payload-based network intrusion detection," Springer, pp. 203–222, 2004 [International Workshop on Recent Advances in Intrusion Detection].
[8] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," IEEE, pp. 44–51, 2003 [High performance interconnects, 2003. proceedings. 11th symposium on].
[9] T. AbuHmed, A. Mohaisen, and D. Nyang, "A survey on deep packet inspection for intrusion detection systems," arXiv preprint arXiv:0803.0037, 2008.
[10] A. A. Diro, and N. Chilamkurti, "Distributed attack detection scheme using deep learning approach for Internet of Things," Future Generation Computer Systems, vol. 82, pp. 761–768, 2018.
[11] A. Abeshu, and N. Chilamkurti, "Deep learning: the frontier for distributed attack detection in Fog-to-Things computing," IEEE Communications Magazine, vol. 56, 2, pp. 169–175, 2018.
[12] P. Narang, V. Khurana, and C. Hota, "Machine-learning approaches for P2P botnet detection using signal-processing techniques," ACM, pp. 338–341, 2014 [Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems].
[13] O. Kolesnikov, and W. Lee, "Advanced polymorphic worms: Evading ids by blending in with normal traffic," Georgia Institute of Technology, 2005.
[14] S. Hochreiter, and J. Schmidhuber, "Long short-term memory," Neural computation, vol. 9, 8, pp. 1735–1780, 1997.
[15] S. Shin, K. Hwang, and W. Sung, "Fixed-point performance analysis of recurrent neural networks," IEEE, pp. 976–980, 2016 [2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)].
[16] S. Forrest, S. Hofmeyr, and A. Somayaji, "The evolution of system-call monitoring," IEEE, pp. 418–430, 2008 [Computer Security Applications Conference, 2008. ACSAC 2008. Annual].
[17] N. Srivastava, E. Mansimov, and R. Salakhudinov, "Unsupervised learning of video representations using lstms," pp. 843–852, 2015 [International conference on machine learning].
[18] G. Creech, and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns," IEEE Transactions on Computers, vol. 63, 4, pp. 807–819, 2014.
[19] D. Wagner, and P. Soto, "Mimicry attacks on host-based intrusion detection systems," ACM, pp. 255–264, 2002 [Proceedings of the 9th ACM Conference on Computer and Communications Security].