Enhancing Automata Learning with Statistical Machine Learning: A Network Security Case Study

Negin Ayoughi, Shiva Nejati, Mehrdad Sabetzadeh
University of Ottawa, Canada
{negin.ayoughi,snejati,m.sabetzadeh}@uottawa.ca

Patricio Saavedra
RabbitRun Technologies Inc.
Toronto, Ontario, Canada
pat@rabbit.run

ABSTRACT

Intrusion detection systems are crucial for network security. Verification of these systems is complicated by various factors, including the heterogeneity of network platforms and the continuously changing landscape of cyber threats. In this paper, we use automata learning to derive state machines from network-traffic data with the objective of supporting behavioural verification of intrusion detection systems. The most innovative aspect of our work is addressing the inability to directly apply existing automata learning techniques to network-traffic data due to the numeric nature of such data. Specifically, we use interpretable machine learning (ML) to partition numeric ranges into intervals that strongly correlate with a system's decisions regarding intrusion detection. These intervals are subsequently used to abstract numeric ranges before automata learning. We apply our ML-enhanced automata learning approach to a commercial network intrusion detection system developed by our industry partner, RabbitRun Technologies. Our approach results in an average 67.5% reduction in the number of states and transitions of the learned state machines, while achieving an average 28% improvement in accuracy compared to using expertise-based numeric data abstraction. Furthermore, the resulting state machines help practitioners in verifying system-level security requirements and exploring previously unknown system behaviours through model checking and temporal query checking. We make our implementation and experimental data available online.

ACM Reference Format:
Negin Ayoughi, Shiva Nejati, Mehrdad Sabetzadeh and Patricio Saavedra. 2024. Enhancing Automata Learning with Statistical Machine Learning: A Network Security Case Study. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Our work stems from the needs of our industry partner, RabbitRun Technologies (RRT for short). RRT develops affordable network routers for small office and home office (SOHO) users – a customer base that has grown in importance and size during and post-pandemic. Network routers are complex devices that handle various protocols and services. RRT's router comes with a proprietary network intrusion detection system that monitors network traffic for suspicious activities to detect various types of attacks.

Understanding and characterizing system-level behaviours of a complex network intrusion detection system poses a challenge due to the heterogeneity of network platforms and the constantly evolving landscape of cyber threats. Developing such understanding and being able to precisely capture system-level behaviours are nonetheless paramount for our industry partner for several reasons, including (a) providing better support resources and guidelines for network administrators and operators, (b) improving the identification of security vulnerabilities, such as unexpected responses to certain types of network traffic, and (c) having an analyzable specification against which to test new router software versions. To facilitate the above, *we investigate the feasibility and effectiveness of automatically learning system-level behavioural models, more specifically system-level automata, to capture the behaviours of RRT's network intrusion detection system.*

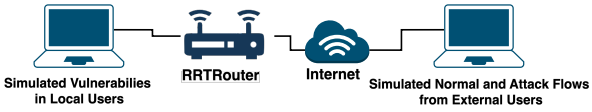
Automata learning derives state machines in a black-box manner [2, 13, 20, 21, 30]. This can be done in either an active or passive mode. Active learning involves algorithms interacting with the system under learning to generate data, whereas passive learning uses existing datasets like log files. Active automata learning has been used in real-world scenarios, including network protocols like BLE [24], MQTT [28], and TCP [11], where a clearly defined interface with the system under learning is available. Network intrusion detection systems, however, analyze the continuous and numeric properties of network-traffic flows over time to detect security attacks, such as denial of service [33]. That is, their inputs and outputs consist of time-series data involving numerical and continuous values. These systems cannot support an iterative query-and-response closed-loop, which is crucial for active learning. Passive learning, on the other hand, is a more suitable strategy for learning the behaviours of network intrusion systems. Nonetheless, passive learning cannot directly handle numeric time-series data.

In this paper, we propose the *MachinE Learning-enhanced passive Automata learning approach (MELA)* to derive state machines for network intrusion detection systems with numeric time-series inputs and outputs. Our approach generates network flows simulating both normal and attack scenarios, and captures the system outputs. We transform the time-series data obtained from the network flows and system outputs into a set of traces. This transformation involves partitioning raw, numerical ranges into a set of intervals. We use decision trees to optimize the correlation of the resulting intervals with a system's decisions regarding intrusion detection. We then use passive automata learning [14] to derive state machines that capture the behaviours of RRT's network intrusion detection system.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(a) Deployment of RRTRouter in the real-world



(b) Our testbed simulating the real-world deployment of RRTRouter

Figure 1: Our case study system: (a) Deployment of our industry partner's router (RRTRouter) in the real-world, and (b) our testbed simulating the real-world deployment.

Contributions. Our paper demonstrates the effective application of passive automata learning for building accurate and practically useful state machines for systems with numerical, time-series inputs and outputs. The key idea behind our approach is using decision-tree learners to abstract the numeric data in traces so as to improve the accuracy and precision of the learned state machines. We evaluate our approach, MELA, using an industrial network intrusion detection system developed by our partner, RRT. We compare the state machines learned using MELA with those developed using two baselines: one employing passive automata learning without data abstraction and the other using automata learning combined with expertise-based data abstraction. Our results indicate that, when given the same set of time-series data, MELA generates state machines with an average of 67.5% fewer transitions and states, while, on average, achieving 28% higher accuracy compared to the baselines. That is, combining automata learning with our ML-driven data abstraction leads to more concise state machines that can more accurately represent the behaviours of the system under learning. Further, we show how, using temporal model checking [9] and query checking [8], the learned state machines help our industry partner with verifying system-level requirements and identifying previously unknown system-level behaviours.

2 INTRUSION DETECTION CASE STUDY

Figure 1a illustrates the deployment of an RRT network router, which enables local users to connect to the Internet and external networks. The router is equipped with an intrusion detection system to identify denial of service (DoS) and distributed denial of service (DDoS) attacks originating from external networks. DoS attacks flood their target with excessive traffic from a single source, whereas DDoS attacks employ multiple sources to perform a more extensive attack [33]. RRT's intrusion detection system monitors the incoming traffic to the router for identifying attacks. If an attack is detected, the system moves to an alert state, triggering actions to block unauthorized traffic. We refer to an RRT router (together with the intrusion detection system running on it) as *RRTRouter*.

To test RRTRouter with different traffic flows, in collaboration with RRT, we have designed and implemented a testbed shown in Figure 1b. The testbed enables the simulation of a varying number of external users, located to the right of RRTRouter, sending network

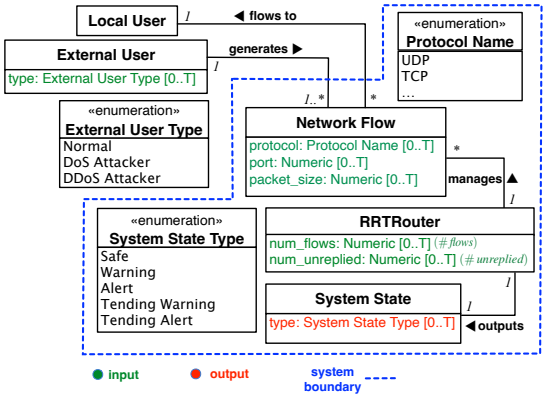


Figure 2: A conceptual model for our testbed.

flows to the local users, located to the left of RRTRouter. As we discuss in Section 4.1, the testbed uses a tool to simultaneously generate normal network flows alongside DoS and DDoS attacks from external users. It further uses a penetration testing framework to simulate vulnerabilities in local users. The network flows from external users pass through RRTRouter, allowing it to collect data, analyze this data, and identify DoS / DDoS attacks.

A conceptual model that represents the entities of our testbed, including its inputs and outputs – shaded green and red respectively – is provided in Figure 2. External users are either normal users or attackers (DoS / DDoS). External users generate network flows, which then go to local users. The inputs to RRTRouter are: (a) the attributes of individual network flows, i.e., `protocol`, `port`, and `packet_size`, and (b) aggregated flow attributes computed for all the flows passing through RRTRouter, i.e., `num_flows` and `num_unreplied`. Here, `protocol` specifies the communication protocol, e.g., `TCP` or `UDP`; `port` specifies the destination port number for the incoming traffic; `packet_size` specifies the size of incoming packets; `num_flows` denotes the number of all flows passing through RRTRouter; and `num_unreplied` indicates the number of flows that are not acknowledged by local users.

RRTRouter updates its state in response to the network flows passing through it, selecting one of the following: `Safe`, indicating no signs of an attack; `Warning` indicating the presence of unusual but not necessarily harmful network flows; and `Alert` indicating the detection of potentially harmful or malicious traffic. In addition, there are the `Tending Warning` and `Tending Alert` states used when RRTRouter is inclined to transition to `Warning` or `Alert`, respectively, but has not conclusively committed to escalation. As Figure 2 shows, the inputs and outputs of RRTRouter are time-series vectors. A time-series vector is a function $v : [0..T] \rightarrow D$, where the interval $[0..T]$ is a *time domain* with duration T, and D is the range of the time series.

In a real-world deployment, RRTRouter has access to all the inputs shown in Figure 2, i.e., the attributes highlighted green, *except for* the type of the external users: in such a deployment, one does not know whether the flows are attack flows or normal flows. Therefore, based on raw data from a real-world deployment, one cannot determine whether RRTRouter accurately identifies attacks and responds appropriately. To verify system behaviours, we need to associate time-series vectors from network flows with

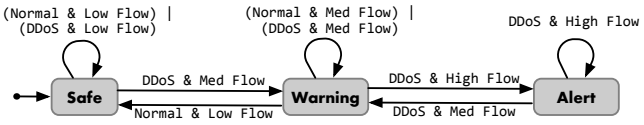


Figure 3: A *simplified* example of a state machine learned for RRTRouter by our approach (MELA).

the user types generating these flows. Our testbed establishes this association, enabling the learning of state machines that reliably link system behaviours to the external user types – attackers or normal users – generating input flows.

In the next section, we present MELA, our approach for deriving state machines from time-series data. A *simplified* example state machine generated for RRTRouter using MELA is depicted in Figure 3. This state machine shows how RRTRouter changes state in response to receiving a Low, Med(ium) or High number of flows from normal users or DDoS attackers. MELA uses a systematic process, guided by machine learning, to derive effective abstractions from numeric time-series data obtained from RRTRouter's inputs and outputs. In Sections 4 and 6, we discuss how these abstractions not only contribute to the conciseness and accuracy of the learned state machines, but also aid RRT engineers in verifying the system-level requirements of RRTRouter and discovering unknown behaviours such as those related to the `Tending Warning` and `Tending Alert` states (not included in Figure 3 due to space). In particular, our state machines enable engineers to (a) identify the ranges for the number of normal and attack flows and (b) refine the requirements of RRTRouter regarding how it should react to normal and attack scenarios with different flow sizes.

3 ML-ENHANCED AUTOMATA LEARNING

Algorithm 1 shows our approach for ML-enhanced automata learning (MELA). The input to the algorithm is a system, S, denoting the system under learning (SUL). MELA assumes that S accepts time-series data as input and generates time-series data as output. Examples of such systems include cyber-physical systems (CPS) and network systems [17, 18]. Our approach treats S as a black box and does not make any assumptions about its internals. In addition to S, MELA requires four input parameters. These parameters, as we will discuss later, are used in the routines responsible for generating traces and abstracting numeric values in time-series data (lines 7–8 of Algorithm 1). The output of MELA is a state machine derived based on the time-series data obtained from S.

Following the formalization of signal-based test inputs and outputs for CPS [12], we denote a test input for S as $\overline{\mathrm{i}} = (i_1, i_2 \ldots i_m)$ and a test output for S as $\overline{\mathrm{o}} = (o_1, o_2 \ldots o_n)$ where m is the number of system inputs, n is the number of system outputs, and each i_j and each o_j is a time-series vector for some input and some output of S, respectively. Given a time-series vector $v : [0..T] \rightarrow D$, the time-series range D can be either discrete, i.e., $D \subseteq \mathbb{N}$, or continuous, i.e., $D \subseteq \mathbb{R}$. Discrete time-series ranges can be enumerate such as $D = \{\mathtt{UDP}, \mathtt{TCP}\}$, or numeric such as $D = \{1, \ldots, 6000\}$. As discussed in Section 1, automata learning algorithms are not able to abstract and generalize numeric data ranges, whether continuous or discrete. Figure 4 illustrates time-series vectors over the time domain [0..30 min] for the inputs of RRTRouter, i.e., the attributes

Algorithm 1 ML-enhanced automata learning (MELA) for systems with time-series inputs and outputs.

Input S: System under learning
Param δ: Sampling rate
Param `Max_Depth`: Maximum depth of decision trees
Param `Sup_Th`: Support threshold for range abstraction
Param `Purity_Th`: Confidence threshold for range abstraction
Output `Aut`: An automaton abstracting the behaviour of S

```
1:  TimeSeriesData = ∅;
2:  do //Data Generation Loop
3:    Input = GenerateInput(S);
4:    Output = Execute(Input, S);
5:    TimeSeriesData = TimeSeriesData ∪ (Input·Output);
6:  until (state coverage is not improving)
7:  Traces = CreateTraces(TimeSeriesData, δ);
8:  Traces' = AbstractTraces(Traces, Max_Depth, Sup_Th, Conf_Th);
9:  Aut = LearnAutomata(Traces');
10: return Aut;
```

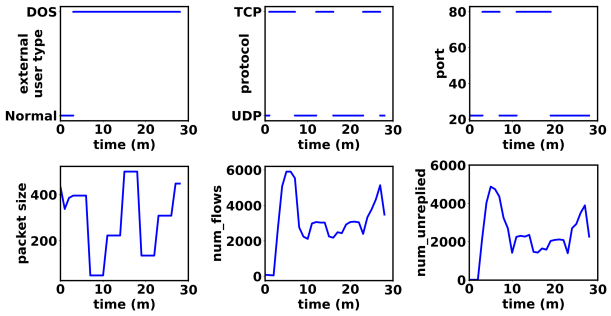


Figure 4: A test input consisting of time-series vectors corresponding to six inputs of RRTRouter.

shaded green in Figure 2: external user type, protocol, port, packet size, and RRTRouter's number of flows and number of unreplied requests. The external user type and protocol have an enumerated range, while the other four inputs are numeric.

We assume that among the outputs of system S, there is one output that captures the system state and has a discrete, enumerable range. For example, as shown in Figure 2, RRTRouter has a specific system-state output. A system-state output is often present in CPS and network systems, which require continuous monitoring and control [5, 12, 18]. This output offers feedback on the system's operational status and helps with decision making and control.

Data Generation. Algorithm 1 begins with a data generation loop (lines 2–6), where it iteratively generates test inputs for S and executes system S to produce test outputs. The purpose of the data generation loop is to produce time-series data to be used for automata learning. In this loop, the test inputs are randomly generated using existing parameterized time-series data generation techniques [4, 29]. To ensure that the learned automata effectively capture the behaviours of SUL, we need to generate test inputs that exercise SUL for different scenarios and yield test outputs that adequately capture SUL's behaviours. To increase the adequacy of the generated data, we employ established black-box test coverage criteria for software testing based on system-state coverage [3]. Specifically, the data generation loop of Algorithm 1 terminates

(a) Before trace abstraction

```
([80, TCP, 173, 82, Normal], Safe), ([80, TCP, 173, 82, Normal, 80, TCP, 122,
      767, DoS], Warning), ([80, TCP, 173, 82, Normal, 80, TCP, 122, 767, DoS,
      80, TCP, 178, 4331, DoS], Alert) . . .
```

(b) After trace abstraction

```
([Normal, Low], Safe), ([Normal, Low, DoS, Med], Warning), ([Normal, Low, DoS,
      Med, DoS, High], Alert) . . .
```

Figure 5: Traces for RRTRouter: (a) an example of an actual trace and (b) the same trace after trace abstraction.

when there is no further improvement in state coverage. This occurs either when the generated outputs cover all system states or when, after several consecutive iterations, the outputs do not cover any new states, indicating that our test generation is unlikely to yield further improvements in state coverage.

Trace Creation. Algorithm 1 uses the CreateTraces routine (line 7) to convert time-series data vectors into traces to be used for autamata learning. Each time-series data vector $v : [0..T] \rightarrow D$ is converted into a sequence $v^0, v^1, \ldots, v^k$ of values using the sampling rate parameter δ, which is an input to Algorithm 1. Specifically, $v^0 = v(0), v^1 = v(\delta), v^2 = v(2 \cdot \delta), \ldots, v^k = v(k \cdot \delta)$, with $k \cdot \delta = T$.

Let $\overline{\mathtt{i}} = (i_1, i_2 \ldots i_m)$ be a test input of S, and $\overline{\mathtt{o}} = (o_1, o_2 \ldots o_n)$ be a test output of S. An input/output trace corresponding to each pair of test input and output of S is defined as follows:

$$(i_1^0, \ldots, i_m^0, o_1^0, \ldots, o_n^0), (i_1^0, \ldots, i_m^0, i_1^1, \ldots, i_m^1, o_1^1, \ldots, o_n^1), \ldots,$$
$$(i_1^0, \ldots, i_m^0, \ldots, i_1^k, \ldots, i_m^k, o_1^k, \ldots, o_n^k)$$

where k is the number of steps with time-step size δ in time domain $[0..T]$, and for every l, i_l^j is the jth sampled value from input vector i_l, and o_l^j is the jth sampled value from output vector o_l.

For example, Figure 5(a) shows a small excerpt from a trace generated for RRTRouter. The values enclosed within "[" and "]" are, respectively, sampled from the following test input vectors of RRTRouter: `port`, `protocol`, `packet size`, `num_flows` and `External user type`. The last value in each tuple is sampled from the test output vector of RRTRouter: `State`.

Trace Abstraction. Automata learning approaches assume that traces consist of abstract values only [20]. Hence, raw numerical values should be replaced by categorical or interval-based representations before applying automata learning. To obtain traces consisting of abstract values, we use the AbstractTraces routine (line 8), which is a novel contribution of our work. This routine uses statistical machine learning to refine traces consisting of raw numerical values into a more abstract form. The AbstractTraces routine consists of two steps: *First,* among all the inputs and outputs of S, we select those that are non-redundant and most correlated with the system state. *Second*, we abstract raw, numeric ranges of the inputs and outputs of S into discrete categories. Below, we describe these two steps; we refer to the first step as *variable selection* and to the second step as *range abstraction.*

Variable selection. We use the information gain [32] to calculate the importance score of each input and each output of S with respect to predicting the system state. To do so, we create a table using the data from our traces such that each column lists the values appearing in the traces for an individual input or output, and the last column lists the corresponding values for the system state. We then compute the information gain of all the columns in this table with respect to the last column. The information gain quantifies the predictive relevance of each column, which represents a system input or output, towards the system state. We rank these columns based on their importance score and eliminate the lowest-ranked ones (i.e., select the highest-ranked ones). We then refine the traces obtained from the trace creation routine (line 7 of Algorithm 1) by removing the values related to the eliminated inputs and outputs.

For example, by applying variable selection to RRTRouter, we remove the variables `num_unreplied`, `port`, `protocol`, and `packet size`, and retain only `num_flows` and `External User type`. This modification follows the computation of importance scores, where `num_flows` and `External User type` are the top-ranked with a large distance from the other variables in terms of importance.

Range abstraction. We use decision-tree learners to abstract the ranges of numeric inputs and outputs included in our traces after the variable selection step. For each numeric input or output variable u, we create a two-column table where the first column is the values of u appearing in the traces, and the second column is the corresponding values for the system state. We then develop a decision tree with inputs as the values of u and outputs as the values of the system state which is a categorical variable. The tree's construction is controlled by a stopping criterion specified by the input parameter `Max_Depth` of Algorithm 1. This parameter determines the maximum depth to which the tree can grow. Each tree leaf represents the following information: (1) the count of samples that are clustered in that leaf (support), and (2) the purity of the leaf (confidence), which indicates the homogeneity of the samples within the leaf. A higher purity indicates a greater concentration of one class; in our context, each class corresponds to a system state. Every leaf is linked to its immediate parent node through a condition such as $u < c$, where u is the variable and c is a constant in the range of u. Among all the tree leaves, we select the ones that have a number of samples and a purity percentage, respectively, higher than the `Sup_Th` and `Purity_Th` thresholds, which are input parameters of Algorithm 1. We consider the conditions $u < c$ linking the selected leaves to their immediate parent nodes. Let $c_1, \ldots, c_k$ denote the constants in ascending order that appear in these conditions. We partition the range of u into the following intervals: $[0, c_1), [c_1, c_2) \ldots [c_k, \infty)$. Then in our traces, we replace the numeric values of u with the categories representing these intervals.

For example, Figure 6 shows a decision tree used to abstract the numeric `num_flows` variable. As shown in Figure 5(b), our traces for RRTRouter include tuples relating values of `num_flows` and other input variables of RRTRouter to system-state values. The decision tree in Figure 6 determines, based on the `num_flows` and `state` values extracted from traces, how well `num_flows` predicts the system state. For this example, we assume that `Max_Depth` is set to 3, and the `Sup_Th` and `Purity_Th` thresholds are set to 20% of the total data and to 70%, respectively. In Figure 6, we show the generated tree leaves and their respective number of samples and purity level. We select all three leaves in the figure, i.e., Node 2, Node 4 and Node 5, since for all these leaves, both the sample count and purity are greater than their respective thresholds. Based on the conditions linking these leaf nodes to their parent nodes, we partition the range of `num_flows` into the following intervals: $[0, 454)$, $[454, 3500)$, $[3500, \infty)$. The number of intervals is not fixed a priori and depends on the decision tree. For `num_flows`, as we have three intervals, we

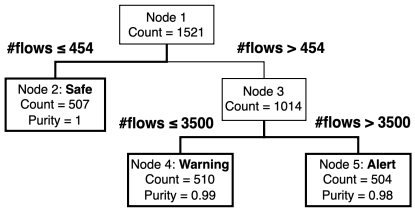


Figure 6: Illustrating how a decision tree is used to abstract the numeric range of the `num_flows` input attribute. The numeric range of `num_flows` is abstracted to the enumerated range [Low, Med, High] such that "Low"=$[0, 454)$, "Med"=$[454, 3500)$, and "High"=$[3500, \infty)$.

designate them as "Low", "Med", and "High", respectively, for better readability. Specifically: "Low"=$[0, 454)$, "Med"=$[454, 3500)$, and "High"=$[3500, \infty)$. After performing variable selection and range abstraction for RRTRouter, we convert the trace in Figure 5(a) to that in Figure 5(b).

Automata Learning. Algorithm 1 uses the LearnAutomata routine to build an automaton (line 9) from abstract traces. We realize this routine using the well-known Regular Positive Negative Inference (RPNI) [2, 10, 14] algorithm which is for passive learning. While the original RPNI algorithm has been designed to generate DFAs, there are variants of RPNI that can learn Mealy and Moore machines [20], which are more expressive models of computation wherein outputs depend not only on the current state but also on inputs. For our case-study system, RRTRouter, we choose to generate *Moore machines*, as this system's traces include both inputs and outputs. Furthermore, Moore machines map outputs directly to states, making them a suitable representation for RRTRouter, as the output of RRTRouter reflects the system's state. In the rest of this paper, depending on the context, we interchangeably refer to the outputs of our approach – Moore machines – as either state machines or automata.

For example, Figure 3 shows a simplified state machine learned for RRTRouter, illustrating how it transitions between different states – `Safe`, `Warning`, and `Alert` – based on the number of flows – Low, Med(ium), or High – and different external user types – Normal or DDoS. Briefly, RRTRouter starts in the `Safe` state and maintains this state while there is no attack, i.e., Normal users sending flows, or when an attack involves low numbers of flows. The system transitions to the `Warning` state under a medium-flow DoS attack, escalating to the `Alert` state under a high-flow DoS attack.

4 EVALUATION

We evaluate MELA by (1) assessing the complexity and accuracy of the generated state machines in representing the behaviours of our case-study system (RRTRouter), and (2) investigating how the learned state machines help with verifying this system against its requirements and exploring unknown behaviours. Specifically, our evaluation aims to answer the following research questions (RQs):

RQ1 (Complexity and Conformance). *How effective is the trace abstraction component of MELA in reducing the complexity of the generated state machines while maintaining a high-level of accuracy?* RQ1 assesses how the trace abstraction component of MELA, specifically line 8 of Algorithm 1, impacts the complexity and accuracy of the generated state machines. The primary goal of the MELA is to generate state machines that are understandable, abstract, yet highly accurate, ensuring high conformance to SUL. We consider two baselines for RQ1: (1) An approach that learns state machines from the traces generated by RRTRouter without performing any trace abstraction, and (2) An approach that incorporates a trace abstraction component similar to MELA, but rather than relying on statistical machine learning, it uses manually defined abstractions based on expert judgment. We refer to the former baseline as Passive since it is the same as the state-of-the-art passive automata learning, and to the latter as Manual since it uses manually defined abstractions and human judgment.

RQ2 (Verification). *Do the state machines learned using MELA help determine whether the system meets its requirements and explore its unknown behaviours?* In collaboration with RRT, we elaborate the system-level requirements of RRTRouter into detailed temporal properties. As discussed in Section 2, the behaviours of RRTRouter is not fully known, particularly for the circumstances under which the system may enter the `Tending Warning` and `Tending Alert` states. To explore unknown system behaviours, in addition to temporal properties, we develop *temporal queries* which are temporal properties with placeholders [8]. Temporal queries yield predicates such that, when these predicates replace the placeholder in the query, they form a property that holds over the state machine. We report on the evaluation of our temporal properties and queries against the state machines learned in RQ1.

4.1 Testbed Implmenetation

We generate the time-series input data for MELA using the testbed presented in Figure 1b. Our testbed consists of three virtual machines (VMs) created using VirtualBox [22] and deployed on three separate laptop computers. The left VM, i.e., VM-left, simulates local users, the centre VM, i.e., VM-centre, hosts the RRTRouter from RRT, and the third VM, i.e., VM-right, simulates external users. Both attackers and normal users are simulated using VM-right, while VM-left is designed to capture potential vulnerabilities in local users. VM-right generates two types of data flows: normal traffic and DoS/DDoS packets, using the *hping3* tool [16] – a network tool designed for testing firewall rules and network performance. After passing through RRTRouter (VM-centre), these flows are directed to VM-left, running Metasploitable [26] – an intentionally vulnerable Linux VM for security and penetration testing. VM-left and VM-right are connected to RRTRouter using two unmanaged NETGEAR GS308v3 Gigabit Ethernet switches. The time-series values for all input and output variables of the system, except for the values of the `external user type` variable, are gathered from VM-centre. The values for `external user type`, which are not accessible through VM-centre, are collected from VM-right.

4.2 RQ1: Complexity and Conformance

We discuss the experiment design and the results obtained for RQ1.

4.2.1 Baselines. We compare MELA with the Passive and Manual baselines (introduced earlier). These baselines obtain traces similarly to MELA, using the same data generation loop and trace creation steps, i.e., lines 1–7 of Algorithm 1. However, the baselines differ from MELA on line 8. Specifically, Passive skips line 8 and

proceeds directly to automata learning, i.e., line 9 of Algorithm 1, after the traces are created. The Manual baseline, on the other hand, uses a two-step data abstraction process, similar to that of MELA, which encompasses variable selection and range abstraction. This baseline nonetheless relies on expert knowledge rather than machine learning. Specifically, for variable selection, a domain expert manually selects the most relevant input and output variables of RRTRouter for identifying DoS and DDoS attacks. In the range abstraction step, the expert abstracts the numeric ranges by partitioning them into enumerated categories of his choice.

4.2.2 Experiment Design. To compare MELA with the two baselines, we use identical sets of traces as inputs for these techniques by applying the data generation and trace creation steps of Algorithm 1 over the testbed described in Section 4.1. We refer to the generated trace sets, which are used to learn automata, as *learning sets*. We use time-series vectors with a time domain $[0..256min]$ for the input generation routine, i.e., line 3 of Algorithm 1, and the sampling rate $\delta = 10sec$ to convert the time-series data vectors into traces by the trace creation routine, i.e., line 7 in Algorithm 1. The choice of sampling rate is based on the minimum refresh rate that our testbed supports. A long duration in the time domain was recommended by the domain expert to enable us to obtain traces achieving higher state coverage.

As discussed in Section 2, RRTRouter's inputs consist of normal traffic combined with either DoS or DDoS attacks. We do not alternate between DoS and DDoS attacks during an individual testing campaign, as each type of attack requires a different experimental setup. Hence, we develop distinct learning sets for DoS and DDoS attacks, resulting in separate state machines capturing the behaviours of RRTRouter for each attack type. To account for randomness in the generation of the learning sets, we rerun the data generation loop of Algorithm 1 five times for DoS and five times for DDoS, obtaining five different learning sets for each attack type. For DoS, three of these learning sets covered only three system states (`Safe`, `Warning`, and `Alert` in Figure 2), while the other two sets covered all five system states in Figure 2. For DDoS, all generated learning sets could only cover three out of the five system states, i.e., `Safe`, `Warning`, and `Alert`. This inability to achieve full state coverage led to further investigation, as we will explain in Section 4.3. This investigation revealed, as confirmed by our domain expert, that the two states, `Tending Warning` and `Tending Alert`, are unreachable under DDoS attacks.

In view of the above, we use the following learning sets for our experiments: `DoS3`, the union of the three learning sets for DoS with three-state coverage; `DoS5`, the union of the two learning sets for DoS with five-state coverage; and `DDoS`, the union of the five learning sets for DDoS. We note that one could obtain a single learning set for DoS by combining all five learning sets. However, as indicated in our online material [1], the state machines obtained by unioning all five sets and those obtained by combining the two five-state coverage sets are identical. Hence, in the paper, we present the results for two separate DoS learning sets – one for three-state and another for five-state coverage – to facilitate direct comparison with the DDoS results, which only achieve three-state coverage. Table 1(a) shows the average length of traces in `DoS3`, `DoS5`, and `DDoS`, as well as the total execution time (in minutes) required to generate the traces in these learning sets.

Table 1: Parameters for our experiments: (a) parameters of the learning sets used by MELA and the baselines; (b) parameters of the trace abstraction step of MELA; and (c) information about trace abstraction in the Manual baseline.

a. Trace Generation for MELA, Manual and Passive

Learning Set	Avg Trace Length	Execution Time (m)
`DoS3`: 3-state coverage	1530	870
`DoS5`: 5-state coverage	1402	474
`DDoS`: 3-state coverage	1523	1420

b. Trace Abstraction for MELA

Variables Selected by Information Gain	Range Abstraction by Decision Tree
`1st: num_flows`	`Max_Depth`: 3
`2nd: num_unreplied`	`Purity_Th`: 0.7
`Top-2: num_flows and num_unreplied`	`Sup_Th`: $0.2 \times n$; n is total data

c. Trace Abstraction for the Manual baseline

Variables Selected by Domain Expert	Range Abstraction by Domain Expert
	Let d be the max range of the numeric variable:
`1st: num_flows`	`Low`: $[0...0.33 \times d]$
`2nd: num_unreplied`	`Med`: $[0.33 \times d...0.66 \times d]$
`Top-2: num_flows and num_unreplied`	`High`: $[0.66 \times d...d]$

For the trace abstraction step of MELA, i.e., line 8 of Algorithm 1, we perform variable selection and range abstraction. For variable selection, we rank RRTRouter's input variables, i.e., variables shaded green in Figure 2, based on their information gain relative to the system state. In our experiments, there is a significant information-gain gap between the second- and third-ranked variables. Therefore, we consider three alternative cases involving the selection of the two top-ranked variables of the system: (1) selecting the variable with the highest information gain, i.e., `1st: num_flows`; (2) selecting the variable with the second-highest information gain, i.e., `2nd: num_unreplied`; and (3) selecting the top two variables with the highest information gains, i.e., `Top-2: num_flows and num_unreplied`. For range abstraction, we use a decision tree to convert the numeric ranges of the RRTRouter's variables into enumerated ranges. We set the tree's maximum-depth parameter in Algorithm 1 to three (i.e., `Max_Depth` = 3); this helps avoid overfitting and prevents our trees from generating too many leaves, which may result in partitioning numeric ranges into several fine-grained intervals and having overly detailed enumerated ranges. Further, we set the `Sup_Th` parameter to 20% of the total data count used for building the decision tree, and the `Purity_Th` parameter to 70%. This ensures that the tree leaves selected for defining partitions contain a sufficient number of elements corresponding to a specific system state. The details related to the trace abstraction component of MELA for our experiments are presented in Table 1(b).

Since the Passive baseline does not perform any trace abstraction, it has no parameters related to trace abstraction. For the Manual baseline, we use the domain expert's judgment for variable selection and range abstraction. Specifically, for variable selection, we requested the domain expert to select the two variables that, in his opinion, most significantly impact the state of RRTRouter. The variables selected by the expert matched those identified by our approach. For range abstraction, the expert suggested dividing the ranges of each numeric variable into three equal intervals of `Low`, `Med` and `High`. The information related to the trace abstraction for the Manual baseline is shown in Table 1(c). Finally, for automata

learning, MELA and the two baselines use the RPNI passive learning algorithm implemented by the AALpy [20] tool, a Python library that provides a range of advanced automata learning algorithms.

4.2.3 Metrics. To assess the complexity of the generated state machines, we report the number of states, the number of transitions, and the size of the input alphabet. These three size-based metrics are commonly used in the literature to evaluate the complexity of state machines [15]. To measure the accuracy of the learned models, we follow the established practice in the automata learning literature [2] and compare the models against the ground truth, i.e., traces generated by the RRTRouter's testbed in our context. We note that the learned automata in passive learning are only as good as the traces in the learning sets. Since the learning sets might be incomplete, there could be an accuracy gap between the behaviours of the learned automata and those of the actual system.

To measure the accuracy of MELA and the two baselines, we generate five randomly generated sets of traces and determine the percentage of these traces that the automaton generated by each technique can accept. We refer to these sets as *test sets*. We use the data generation loop of Algorithm 1 to create these test sets. To ensure that we cover a range of trace lengths, we generate five test sets referred to as *very small*, *small*, *medium*, *large*, and *very large*. Each test set has 100 traces, leading to a total of 500 traces for assessing the accuracy of the learned automata. The average trace lengths in these test sets are as follows: very small at 70, small at 138, medium at 207, large at 276, and very large at 345. The lengths of the traces in the very small, small, medium, large, and very large test sets are chosen to be approximately 10%, 20%, 30%, 40%, and 50% of the trace length in the learning set, respectively. It took approximately 600 hours to generate the traces in these test sets using our testbed discussed in Section 4.1. Given the high cost of trace generation, we decided to cap the max length of traces in test sets at 50% of the size of traces in our learning sets.

4.2.4 Results. To answer RQ1, we applied MELAas well as the Manual and Passive baselines to the learning sets DoS3, DoS5 and DDoS. However, using the Passive baseline, we could not generate any automaton. Recall from Section 4.2.1 that Passive does not perform any abstraction and uses the traces with raw numeric values. Provided with such traces, the AALpy tool failed to generate any result since the traces contained too many distinct numeric values. Hence, we only compare the results of MELA and Manual with respect to our complexity and accuracy metrics discussed in Section 4.2.3. Table 2 reports the number of states and transitions and the alphabet size for the generated automata by MELA and the Manual baseline. Specifically, we obtain 18 automata by applying the two approaches to the three different learning sets and considering three different abstraction options of 1st, 2nd, and Top-2. Figure 7 shows the accuracy results for the learned automata by MELA and Manual. The accuracy values are computed by applying each of the 18 learned automata to the 500 test traces described in Section 4.2.3. Each plot in Figure 7 shows the accuracy distribution of each learned automaton with respect to our 500 test traces.

As indicated by Table 2 and Figure 7, all the 18 generated automata by MELA have fewer states and transitions than the corresponding autoamta generated by Manual. Through the ML-based trace abstraction in MELA, we obtain automata that, on average, have 69.61% fewer states and 65.41% fewer transitions than automata derived using expertise-based abstraction. Further, the alphabet size of the automata generated by MELA is the same as or smaller than that of the automata generated by Manual. The accuracy results in Figure 7 show that MELA not only substantially reduces the size of the learned automata but also results in significantly more accurate automata. On average, the automata learned by MELA are 28.75% more accurate than those learned by Manual.

Table 2: Comparing the number of states, number of transitions, and the alphabet size for the state machines learned by MELA versus by the Manual baseline.

Learning set	Configuration	MELA			Manual		
		# States	# Transitions	|Alphabet|	# States	# Transitions	|Alphabet|
DoS3	Top-2	3	11	7	23	65	8
	1st	3	9	5	290	325	5
	2nd	19	51	5	29	68	5
DDoS	Top-2	3	11	7	20	52	8
	1st	3	9	5	284	319	5
	2nd	17	46	5	24	62	5
DoS5	Top-2	46	83	8	167	253	9
	1st	47	78	5	410	499	5
	2nd	121	187	5	178	267	5

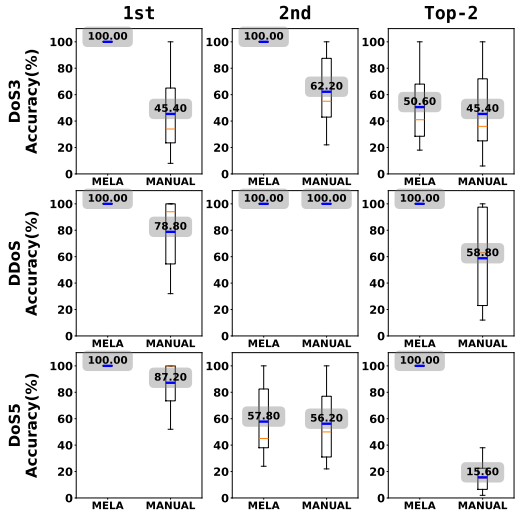


Figure 7: Comparing the accuracy of the state machines learned by MELA versus by the Manual baseline for different learning sets (DoS3, DoS5 and DDoS), and different configurations (1st, 2nd, and Top-2 as defined in Table 1).

RQ1: Our approach (MELA) leads to an average reduction of 67.5% in the number of states and transitions of the learned automata, while improving accuracy by an average of 28% compared to using expertise-based abstractions for automata learning.

4.3 RQ2: Verification

To answer RQ2, we use six state machines from RQ1: (a) three developed using the DoS5 learning set (the learning set with the highest coverage for DoS attacks), corresponding to the 1st, 2nd, and Top-2 configurations; and (b) three developed using the DDoS learning set for DDoS attacks, corresponding to the same configurations.

Table 3: Model-checking results for the temporal properties derived from RRTRouter requirements R_1 and R_2

(a) Results for temporal properties derived from R_1

	Learning set	1st			2nd			Top-2		
		Low	Med	High	Low	Med	High	Low	Med	High
G(Attack ∧ S ⟹ X(W ∨ TW))	DoS5	×	✓	v	×	✓	v	×	✓	v
	DDoS	×	✓	v	×	✓	v	×	✓	v
G(Attack ∧ W ⟹ X(A ∨ TA))	DoS5	v	×	✓	v	✓	✓	v	×	✓
	DDoS	v	×	✓	v	×	✓	v	×	✓
G(Attack ∧ A ⟹ X(A ∨ TA))	DoS5	v	×	✓	v	✓	✓	v	×	✓
	DDoS	v	×	✓	v	×	✓	v	×	✓

(b) Results for temporal properties derived from R_2

	Learning set	1st			2nd			Top-2		
		Low	Med	High	Low	Med	High	Low	Med	High
G(Normal ∧ S ⟹ X(S))	DoS5	✓	v	v	✓	v	v	✓	v	v
	DDoS	✓	v	v	✓	v	v	✓	v	v
G(Normal ∧ W ⟹ X(S))	DoS5	✓	×	v	✓	×	v	✓	×	v
	DDoS	v	×	v	✓	×	v	✓	×	v
G(Normal ∧ A ⟹ X (W ∨ TW))	DoS5	v	v	v	v	v	v	v	v	v
	DDoS	v	v	v	v	v	v	v	v	v

We identify two high-level requirements for RRTRouter related to its intrusion detection function: R_1 *When attacks happen, the system shall change state in a staged manner from safe to warning and from warning to alert.* R_2 *When attacks are stopped, the system shall restore in a staged manner its state from alert to warning, and from warning to safe.* In collaboration with RRT, we derived temporal properties from R_1 and R_2, shown in the leftmost columns of Tables 3(a) and 3(b), respectively. These properties are expressed as Linear Temporal Logic (LTL) formulas [9, 25], where "G" is the globally operator and "X" is the next state operator. For succinctness, we use abbreviated names for states: "S" for `Safe`, "W" for `Warning`, "A" for `Alert`, "TW" for `Tending Warning`, and "TA" for `Tending Alert`. For example, the property G (Attack ∧ S ⟹ X (W ∨ TW)) specifies that if an attack occurs at state S, the system must transition to W or TW in the next state.

The properties in Table 3(a), derived from R_1, indicate that the system must transition to a higher criticality level in the next state in response to DoS or DDoS attacks. Here, state S represents the lowest criticality state, while TA and A denote the highest. Note that when RRTRouter is in the TA or A states and an ongoing attack persists, the system shall remain in the TA or A state. In a dual manner, the properties in Table 3(b), derived from R_2, indicate that the system must transition to a lower criticality level under normal traffic conditions and when there is no ongoing attack.

As discussed in Section 2, there is uncertainty regarding the system's subsequent state after attack scenarios or under normal traffic conditions for the TW and TA states. Hence, we cannot develop temporal properties having TW and TA as their antecedent similar to those in Tables 3(a) and 3(b). Instead of temporal properties, we develop temporal queries with placeholders shown in the leftmost column of Table 4. The queries in Table 4 enable us to explore the subsequent state of RRTRouter at the TW and TA states under attack and normal traffic. For example, the temporal query G (Attack ∧ TA ⟹ X (?)) uses a placeholder for the "X" operator.

4.3.1 Model-checking. Tables 3(a) and 3(b) show the results of verifying the state machines obtained from the `DoS5` and `DDoS` learning sets against the temporal properties for R_1 and R_2, respectively. For each state machine, a temporal property is marked as pass (✓) if it holds, as fail (×) if it is violated, or as vacuous (v) if the property's antecedent is never met, resulting in vacuous satisfaction [6]. Note

Table 4: Exploring the behaviours of RRTRouter at the TW and TA states using temporal queries.

	Learning set	1st			2nd			Top-2		
		Low	Med	High	Low	Med	High	Low	Med	High
G(Attack ∧ TW ⟹ X(?))	DoS5	v	TW ∨ TA	A∨ TA	v	TW	TW ∨ TA	v	TW	TA
	DDoS	v	v	v	v	v	v	v	v	v
G(Attack ∧ TA ⟹ X(?))	DoS5	v	W	A ∨ TA	v	W	A ∨ TA	v	W	A ∨ TA
	DDoS	v	v	v	v	v	v	v	v	v
G(Normal ∧ TW ⟹ X(?))	DoS5	v	v	v	v	v	v	v	v	v
	DDoS	v	v	v	v	v	v	v	v	v
G(Normal ∧ TA ⟹ X(?))	DoS5	v	v	v	v	v	v	v	v	v
	DDoS	v	v	v	v	v	v	v	v	v

that the antecedent refers to the subformulas of the properties in Tables 3(a) and 3(b) that appear before the ⟹ operator. When assessing the temporal properties, we observed that the number of attack and normal flows has a significant impact on the results. Therefore, Tables 3(a) and 3(b) present the results for low, medium, and high numbers of flows separately.

For example, the property G (Attack ∧ S ⟹ X (W ∨ TW)) holds for the state machines learned for both `DoS5` and `DDoS` and for the `1st`, `2nd`, and `Top-2` configurations when the number of flows is medium. The property fails when the number of flows is low and is vacuous when the number of flows is high. This indicates that when the system is in its `Safe` state, requirement R_1 is met only when the number of attack flows is medium; a low-flow attack goes undetected, and a high-flow attack never happens. Specifically, the property G (Attack ∧ S ⟹ X (W ∨ TW)) holding vacuously for high-flow traffic implies that the antecedent, Attack ∧ S, never holds when the number of attack flows is high.

4.3.2 Query-checking. Table 4 shows the results of evaluating the state machines obtained from the `DoS5` and `DDoS` learning sets against our temporal queries for R_1 and R_2. Evaluating a temporal query on a state machine either yields a predicate or results in vacuity (v) if the query's antecedent is never met. When the outcome is a predicate, the property obtained by replacing the placeholder in the query with the predicate will hold over the state machine. For example, consider query G (Attack ∧ TA ⟹ X (?)) in Table 4. The results indicate that RRTRouter never experiences a low-flow attack when in state TA. This is because, for all the state machines, this query is vacuous for low-flow traffic. However, for medium-flow DoS attacks, the query yields W, meaning that when subjected to such attacks in the TA state, the system transitions to the W state.

4.3.3 Analysis. The results presented in Tables 3(a), 3(b), and 4 indicate that some temporal properties are violated and several properties and queries are vacuous. Below, we discuss the reasons for these violations and vacuities and assess whether the results obtained from different state machines are consistent.

Do violations (×) imply defects? The pass and fail results in Tables 3(a) and 3(b) show that RRTRouter reacts to DoS/DDoS attacks with only medium or high number of flows, ignoring low-flow attacks. Medium-flow attacks prompt RRTRouter to transition from state S to TW or W, but not to its most critical states, TA or A, which occur only with high-flow attacks. Dually, RRTRouter returns to state S only with low-flow normal traffic. In addition, the combination of normal traffic with a high number of flows does not appear in our state machines. This matched the domain expert's intuition, who noted that only attackers (and not normal users) may generate such high numbers of flows. We emphasize that the range for a

"high" number of flows is *learned* by our trace abstraction approach and is not a-priori known at the time of trace generation. The fact that the combination of normal traffic with a high number of flows is never generated indicates that our trace abstraction step is able to effectively differentiate between normal users and obvious attackers generating a high number of flows. Furthermore, RRTRouter remains in the TW and W states for medium-flow normal traffic, indicating that it considers this traffic suspicious.

After discussing these findings, our domain expert confirmed that the system's behaviours are acceptable. In particular, since low-flow DoS and DDoS attacks do not degrade the quality of service for clients, there is no need for the system to react. When recovering from a recent attack, however, it is acceptable for the system to flag medium-sized normal traffic as suspicious. This is because in a real-world deployment, the system does not have an oracle to inform it whether it is still under attack, and it is challenging to distinguish medium-flow normal traffic from DoS and DDoS attacks.

> The property violations identified in Tables 3(a) and 3(b) do not indicate defects in RRTRouter. Rather, they indicate that requirements R_1 and R_2 need to be refined to explicitly specify the thresholds for the number of attack and normal flows that the system should treat as suspicious.

Do vacuities (v) show gaps? The vacuous cases in Tables 3(a) and 3(b) are due to two factors: (1) Not all numbers of flows are observable in every system state. This is expected since DoS and DDoS attacks typically start at a low number of flows and gradually escalate. It is typical for a high-flow DoS or DDoS attack to be preceded by a medium-flow phase, which itself follows a low-flow stage. Hence, since a medium-flow attack shifts the state from S to W or TW, a high-flow attack is not observed at state S. (2) As discussed in Section 4.2.2, under a DDoS attack, RRTRouter does not transition to TW and TA states, leading to vacuous outcomes for properties involving these states. Our domain expert confirmed that DDoS attacks involve larger numbers of flows than DoS attacks, causing RRTRouter to bypass the TW and TA states.

> The observed vacuities are not due to gaps (incompleteness) in the learning sets obtained from our testbed. Rather, vacuity occurs because the physical characteristics of network flows impose certain constraints, such as attacks not starting immediately with a high number of flows but needing to grow over time from a lower number. This indicates that, along with the requirements, the environmental assumptions of RRTRouter should also be made explicit to help with refining the temporal properties of Tables 3(a) and 3(b) to avoid vacuity.

Are the results obtained from different state machines consistent? Our state machines – learned based on different configurations, i.e., `1st`, `2nd`, and `Top-2` – yield highly consistent model-checking and query-checking results. In particular, only 8% (3 out of 36 cases) of the model-checking results in Tables 3(a) and 3(b) are inconsistent. Inconsistency means that, for a given property, learning set, and range for the number of flows, we obtain different results for the state machines built using the `1st`, `2nd`, and `Top-2` configurations. In all inconsistent cases, out of the three alternative state machines (`1st`, `2nd`, and `Top-2`), two are in agreement, indicating that the inconsistencies can be resolved through a majority voting between the state machines. Similarly, 12.5% (3 out of 24 cases) of the query-checking results in Tables 4 are inconsistent. Nevertheless, in all of these inconsistent cases, the inconsistency can be resolved by weakening the learned predicates through taking their disjunction. The reason this strategy works is that the predicates replace the consequents of their respective temporal queries, and a disjunction (weakened predicate) still ensures the satisfaction of the query. For example, the query G (Attack $\wedge$ TW $\implies$ X (?)) with medium-flow attack, respectively derives the predicates TW $\vee$ TA, TW, and TW for configurations `1st`, `2nd`, and `Top-2`. Here, the disjunction of the predicates would be: TW $\vee$ TA.

> State machines obtained by different configurations of MELA yield highly consistent model-checking and query-checking results, showing the robustness of our trace abstraction approach. Furthermore, inconsistencies can be resolved by using a majority vote for model checking and by taking the disjunction of predicates for query checking.

4.4 Validity Considerations

Internal validity. To improve internal validity, we implemented measures to minimize the impact of extraneous factors. Specifically, (1) we controlled the trace generation process to prevent external traffic not initiated by our simulations from reaching the local users or the router; (2) we monitored the network during the experiments to ensure the absence of anomalies that might have arisen due to events beyond our control; and (3) to construct each dataset, we repeated the trace generation process five times and combined the results, thereby mitigating the effects of random variability.

External validity. While we believe our approach should generalize to other types of systems with numeric time-series inputs and outputs, we note that our evaluation was conducted on a single system in the domain of network intrusion detection. To more conclusively examine the generalizability of our approach and improve external validity, further experimentation with other systems, such as cyber-physical systems, would be necessary.

5 RELATED WORK

In this section, we compare our work with relevant strands in three areas: (1) automata learning and verification for network protocols, (2) model mining for intrusion detection systems, and (3) supervised rule mining for numeric systems.

Automata learning and verification for network protocols. Muskardin et al. [2] employ AALpy – the same tool we use in this paper for automata learning – to compare passive and active learning for network protocols [23]. They observe that active learning: (1) is time-consuming, especially when the system under learning needs resetting for each new input/output trace; (2) requires a fault-tolerant learning setup for interaction, which can be complex and limit practicality; and (3) involves significant interaction with the

system under learning, incurring high costs due to the potential for losses or delays. Based on these observations, Muskardin et al. argue that passive learning is a more efficient alternative for real-world problems if one can have a diverse and yet sparse dataset of system behaviours. Our work follows the same rationale for choosing passive learning over active learning. To improve learning-set diversity and coverage, we used randomization by rerunning the data generation loop of Algorithm 1 multiple times, further ensuring that we captured all known system states. The resulting learning sets are relatively small (with an average trace length of 1500), thus remaining amenable to effective passive learning.

Fiterau et al. [11] employ automata learning to derive behavioural models for TCP protocol components and apply model checking to verify their conformity with Request for Comments (RFCs). Our approach to learning differs from theirs in two main respects: (1) they do not address numeric systems, making their abstractions unsuitable for our context, and (2) they use active learning, whereas we use passive learning. Both our work and Fiterau et al.'s employ model checking to verify the resulting models. However, the nature of the properties of interest differs: while they examine interactions in TCP network protocol components, we assess the behaviours of an entire system (a network router) in its deployment environment.

Model mining for intrusion detection systems. ML technologies have become crucial for enhancing automated intrusion detection systems. However, as observed by Shahraki et al. [27], ML techniques used for network-traffic monitoring and analysis (including for our use case in this paper) have thus far been heavyweight and primarily tailored to enterprise networks. Little attention has been given to lighter-weight techniques that would be suitable for the needs of small networks or resource-parsimonious networking platforms, such as those prevalent in the market where our industry partner operates. Another issue related to the application of ML for intrusion detection is the lack of interpretability, as humans often struggle to comprehend the deep operational layers of ML-based intrusion detection systems [31].

We are not the first to explore the application of passive learning to mine interpretable models for network intrusion detection systems. Cao et al. [7] use passive learning to construct probabilistic state machines for detecting network anomalies based on features such as protocol, bytes sent, and flow duration. We differ from Cao et al. in two key aspects. First, our work focuses on router firewall behaviours, emphasizing state transitions (Safe, Warning, Alert) based on network flows, whereas Cao et al. target anomaly detection in Kubernetes clusters. Our use case differs from theirs, and the solutions are not interchangeable. Second, whereas Cao et al. use clustering for numeric-range partitioning, we employ decision-tree learners. Noting that our experimental setup allows for control over attack and non-attack scenarios without manual effort or compromising accuracy in labelling, decision trees are advantageous over clustering due to their better interpretability and resilience to outliers and dataset imbalances.

Supervised rule mining for numeric systems. Our research relies on supervised rule mining to improve the abstraction of behavioural models for numeric systems. While we are not aware of prior research that employs supervised rule mining for a similar application, recent work on rule mining for numeric systems has inspired our approach. Notably, Jodat et al. [17, 18] propose a method for combining machine learning and adaptive random testing to identify test inputs leading to non-robust and potentially failing system behaviours. Our current research was conducted with the same industry partner as Jodat et al. (namely, RabbitRun Technologies). However, this prior work focuses on a different aspect of the partner's system, specifically controlling the flow of network traffic (traffic shaping). In short, this earlier research neither concerns learning behavioural models nor addresses intrusion detection.

6 LESSONS LEARNED AND FUTURE WORK

Below, we reflect on the lessons learned from the development of MELA. We believe our lessons would be most relevant for (1) researchers interested in behavioural model mining and model-based verification of cyber-physical and network systems, and (2) practitioners applying model-based analysis for cybersecurity.

(1) For systems with time-series inputs and outputs, automata learning produces effective, interpretable behavioural models. While interpretable statistical learning is effective at deriving static abstractions from data, it is not as effective at capturing temporal behaviours encoded in time-series data. Interpretable ML methods [19], such as decision trees and decision rules, are effective in identifying predicates that explain the relationship between system inputs and system states. However, they are inadequate in capturing temporal relationships between states and in understanding how system inputs might trigger change of states. Our research shows that: (1) automata learning, due to its ability to capture temporal behaviours effectively, proves useful for mitigating the shortcomings of statistical learning, and (2) to overcome the limitations of automata learning in abstracting data, one can increase the level of abstraction in traces first before attempting to learn automata.

(2) Automata learning is useful for analyzing and gaining a better understanding of cyber-intrusion detection systems. There is a lack of industrial case studies on the automated derivation of behavioural models for cyber-intrusion detection systems. Our work highlights some important contextual factors related to the construction of behavioural models for such systems, notably the numeric and time-series nature of the inputs and outputs of these systems, as well as the lack of amenability to active learning techniques due to the difficulty of building query-and-response loops. An important lesson learned from our work is the feasibility of automata learning for cyber-intrusion detection systems through an explicit treatment of time-series numeric data and the use of passive learning.

In future work, we plan to further explore model-based construction and verification of cyber-intrusion detection systems. This includes developing domain-specific languages for this purpose, implementing federated learning of state machines based on numerous instances of routers used by our industry partner's clients, and generating synthetic adversarial attacks to iteratively improve the quality of the learned state machines.

7 DATA AVAILABILITY

We publicly share our input generation scripts, trace creation and abstraction routines, along with all experimental data, including datasets for training and testing, and the resulting state machines [1].

REFERENCES

[1] 2024. MELA: Machine Learning-Enhanced Automata Learning. GitHub repository. Artifacts and Supplementary Material [Online]. Available: https://github.com/neayoughi/MELA.git.

[2] Bernhard K. Aichernig, Edi Muskardin, and Andrea Pferscher. 2022. Active vs. Passive: A Comparison of Automata Learning Paradigms for Network Protocols. In *Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022, and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE)Berlin, Germany, 26th and 27th of September 2022 (EPTCS, Vol. 371)*, Matt Luckcuck and Marie Farrell (Eds.). 1–19.

[3] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing.* Cambridge University Press.

[4] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. 2019. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology* 114 (2019), 137–154.

[5] Foozhan Ataiefard, Mohammad Jafar Mashhadi, Hadi Hemmati, and Neil Walkinshaw. 2022. Deep State Inference: Toward Behavioral Model Inference of Black-Box Software Systems. *IEEE Trans. Software Eng.* 48, 12 (2022), 4857–4872.

[6] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. 1997. Efficient Detection of Vacuity in ACTL Formulaas. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1254)*, Orna Grumberg (Ed.). Springer, 279–290.

[7] Clinton Cao, Agathe Blaise, Sicco Verwer, and Filippo Rebecchi. 2022. Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster. *CoRR* abs/2207.12087 (2022).

[8] William Chan. 2000. Temporal-Locig Queries. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1855)*, E. Allen Emerson and A. Prasad Sistla (Eds.). Springer, 450–463.

[9] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. 2018. *Model checking, 2nd Edition.* MIT Press.

[10] Colin De la Higuera. 2010. *Grammatical inference: learning automata and grammars.* Cambridge University Press.

[11] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 454–471.

[12] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C. Briand, and David Wolfe. 2020. Mining assumptions for software components using machine learning. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 159–171.

[13] Bharat Garhewal and Carlos Diego Nascimento Damasceno. 2023. An Experimental Evaluation of Conformance Testing Techniques in Active Automata Learning. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023.* IEEE, 217–227.

[14] Antonio Cano Gómez. 2010. Inferring Regular Trace Languages from Positive and Negative Samples. In *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6339)*, José M. Sempere and Pedro García (Eds.). Springer, 11–23.

[15] Mathew Hall. 2011. Complexity Metrics for Hierarchical State Machines. In *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6956)*, Myra B. Cohen and Mel Ó Cinnéide (Eds.). Springer, 76–81.

[16] hping3 Authors. 2024. hping3 - Active Network Security Tool. https://www.kali.org/tools/hping3/ Accessed: 2023-02-01.

[17] Baharin Aliashrafi Jodat, Abhishek Chandar, Shiva Nejati, and Mehrdad Sabetzadeh. 2023. Test Generation Strategies for Building Failure Models and Explaining Spurious Failures. *CoRR* abs/2312.05631 (2023).

[18] Baharin Aliashrafi Jodat, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. 2023. Learning Non-robustness using Simulation-based Testing: a Network Traffic-shaping Case Study. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023.* IEEE, 386–397.

[19] Christoph Molnar. 2020. *Interpretable machine learning.* Lulu. com.

[20] Edi Muskardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. 2021. AALpy: An Active Automata Learning Library. In *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12971)*, Zhe Hou and Vijay Ganesh (Eds.). Springer, 67–73.

[21] Daniel Neider, Rick Smetsers, Frits W. Vaandrager, and Harco Kuppens. 2018. Benchmarks for Automata Learning and Conformance Testing. In *Models, Mindsets, Meta: The What, the How, and the Why Not? - Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 11200)*, Tiziana Margaria, Susanne Graf, and Kim G. Larsen (Eds.). Springer, 390–416.

[22] Oracle Corporation. 2024. *Oracle VM VirtualBox.* https://www.virtualbox.org/

[23] Andrea Pferscher and Bernhard K. Aichernig. 2021. Fingerprinting Bluetooth Low Energy Devices via Active Automata Learning. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 524–542.

[24] Andrea Pferscher and Bernhard K. Aichernig. 2022. Fingerprinting and analysis of Bluetooth devices with automata learning. *Formal Methods Syst. Des.* 61, 1 (2022), 35–62.

[25] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 46–57.

[26] Rapid7. 2024. Metasploit Framework. https://www.metasploit.com/. Accessed: 2023-02-01.

[27] Amin Shahraki, Amir Taherkordi, and Øystein Haugen. 2021. TONTA: Trend-based Online Network Traffic Analysis in ad-hoc IoT networks. *Comput. Networks* 194 (2021), 108125.

[28] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. 2017. Model-based testing IoT communication via active automata learning. In *2017 IEEE International conference on software testing, verification and validation (ICST).* IEEE, 276–287.

[29] Cumhur Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. 2019. Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles* 5, 2 (2019), 265–280.

[30] Frits W. Vaandrager. 2017. Model learning. *Commun. ACM* 60, 2 (2017), 86–95.

[31] Maonan Wang, Kangfeng Zheng, Yanqing Yang, and Xiujuan Wang. 2020. An Explainable Machine Learning Framework for Intrusion Detection Systems. *IEEE Access* 8 (2020), 73127–73141.

[32] Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data mining: practical machine learning tools and techniques, 3rd Edition.* Morgan Kaufmann, Elsevier.

[33] Saman Taghavi Zargar, James Joshi, and David Tipper. 2013. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Commun. Surv. Tutorials* 15, 4 (2013), 2046–2069.