Towards a General Model for Intrusion Detection: An Exploratory Study

Tommaso Zoppi(✉), Andrea Ceccarelli, and Andrea Bondavalli

Department of Mathematics and Informatics, University of Florence, Viale Morgagni 65, 50142 Florence, Italy
tommaso.zoppi@unifi.it

Abstract. Exercising Machine Learning (ML) algorithms to detect intrusions is nowadays the de-facto standard for data-driven detection tasks. This activity requires the expertise of the researchers, practitioners, or employees of companies that also have to gather labeled data to learn and evaluate the model that will then be deployed into a specific system. Reducing the expertise and time required to craft intrusion detectors is a tough challenge, which in turn will have an enormous beneficial impact in the domain. This paper conducts an exploratory study that aims at understanding to which extent it is possible to build an intrusion detector that is general enough to learn the model once and then be applied to different systems with minimal to no effort. Therefore, we recap the issues that may prevent building general detectors and propose software architectures that have the potential to overcome them. Then, we perform an experimental evaluation using several binary ML classifiers and a total of 16 feature learners on 4 public attack datasets. Results show that a model learned on a dataset or a system does not generalize well as is to other datasets or systems, showing poor detection performance. Instead, building a unique model that is then tailored to a specific dataset or system may achieve good classification performance, requiring less data and far less expertise from the final user.

Keywords: Intrusion detection · General model · Transferability · Machine learning · Feature learning

1 Introduction

"*Unfortunately, we cannot claim validity of our results beyond the system/datasets used in this study*". This statement appears quite frequently when discussing threats to validity or when remarking lessons learned from an experimental study. At a first glance, it may be seen as a defensive statement, which discourages the reader from applying the proposed technique in systems other than those considered in the study. However, generalizing the results of an experimental study is together one of the main goals and at the same time one of the most difficult achievements of those studies.

In the security domain, this aspect is extremely relevant as most of the mitigations, defenses, and detection mechanisms are tightly tailored to a specific system, domain or attack to defend against. More specifically, intrusion detectors are nowadays built [48] by feeding Machine Learning (ML) with performance indicators that are being continuously monitored and analyzed to spot anomalous behaviors due to ongoing attacks. Those ML algorithms are typically binary classifiers, which aim at distinguishing between normal and attack-related behavior by processing feature values. This has proven to be very effective for detecting a wide variety of attacks, and in the last two decades originated a huge amount of research papers and industrial applications [41–47] that have the potential to improve security attributes of ICT systems. However, researchers and practitioners have to craft intrusion detectors for specific systems, network interfaces and attack models, to name a few.

As a result, intrusion detectors that may have excellent detection performance for a given system will not have comparable detection performance when applied to different systems, network topologies or attack types. On the other hand, the availability of ML algorithms that are more robust, accurate and that orchestrate ensembles of ML algorithms themselves (i.e., meta-learners [14, 17]) may offer the opportunity to build intrusion detectors that generalize well to (slightly) different systems or domains.

Therefore, this paper conducts an exploratory study to understand to what extent, and under which assumptions, it is possible to craft intrusion detectors that have satisfying detection performance and generalize well to different systems. We start by listing the main threats to building general intrusion detectors according to the literature on the domain. This paves the way for proposing two software architectures that rely either on feature mapping or feature learning and that allow building intrusion detectors that are as general as possible, and can potentially be trained once and used in different systems with minimal effort. We then conduct an experimental campaign embracing 4 public attack datasets that have overlapping feature sets and that suit the evaluation of both feature mapping and feature learning architectures for intrusion detection. Results clearly show that it is not possible to build an intrusion detector that is general enough to be trained once and then be applied to other systems or datasets with no effort, achieving satisfying detection performance. Instead, it is possible to build a detector to be used as a baseline and then tailored to the specific system, requiring minimal expertise and less data with respect to creating a system/specific intrusion detector, but having comparable detection performance.

The paper is structured as follows. Section 2 summarizes related works and the main issues in building general intrusion detectors, letting Sect. 3 propose software architectures for building general intrusion detectors. Section 4 expands on our experimental campaign, whose results are elaborated in Sect. 5. Section 6 concludes the paper.

2 On Generalizing Intrusion Detectors

The traditional flow for deriving intrusion detectors [48] starts from identifying security problems and collecting data to be used for learning models. Their performance is then evaluated and compared against potential competitors, and then the detection system is deployed and put into operation. This is a consolidated flow that has been proven effective in many studies [41, 43–47].

2.1 Motivation and Novelty of this Study

However, the intrusion detector created at the end of this process is system-specific, meaning that it is meant to be effective only in a specific system, against specific attacks and under specific additional assumptions (if any).

This forces the security specialist to start almost from scratch whenever they have a new problem to deal with. Companies or research institutes often already have system monitors that can be used to gather the values of performance indicators of a system over a period of time; however, the collection process is time-consuming, and labeling monitored data has even an higher cost. That is why in recent years there were studies [2, 49] aimed at building intrusion detectors that are not system-specific and could generalize to other datasets, requiring far less data and knowledge for training, evaluating and deploying the detector. Both studies rely on two datasets with similar feature sets, learn the model (supervised in [2], unsupervised in [49]) using one dataset, and test their model on the other dataset. Authors agree that a model learned on a dataset cannot perform detection in another one with good detection performance.

In this paper, we are interested in conducting an exploratory study that spans across a wider range of software architectures that could potentially build general intrusion detectors. According to studies [2, 49], we do not expect models learned on a dataset to have excellent detection capabilities on other systems or datasets when used as they are. Instead, we explore the extent to which is it possible to tailor an existing model to a new dataset or system to perform intrusion detection satisfactorily, and the amount of knowledge that is required to perform such tailoring. Should this knowledge be small enough, this would require less expertise and save time (i.e., money) as it will allow building intrusion detectors starting from a general baseline instead of starting every time from scratch [48].

2.2 Issues in Generalizing Intrusion Detectors

Here we summarize the obstacles to building a general intrusion detector.

I1: Domain and Purpose of the System. It is widely acknowledged that modern ICT systems can be targeted by attackers [25, 26]. There is significant evidence on the risk of cyber-attacks, both in terms of the likelihood of being targeted and the cost and impact of a successful attack. The number of computer security incidents has been steadily growing over the past few years: in 2021, SonicWall [26] reported an average of 28 million cyber-attacks detected daily, with 140 000 of them being novel malware samples. Starting from 2020, the European Union Agency for Cybersecurity (ENISA) observed a spike in non-malicious incidents, most likely because the COVID-19 pandemic became a multiplier for human errors and system misconfigurations, and attributed them as the root cause for the majority of security breaches [25].

Consequently, virtually any system connected to a public (but also private) network should be willing to adopt an intrusion detector to ensure that appropriate security requirements are met. From a theoretical standpoint, a general intrusion detector should achieve satisfying detection performance when processing data from any domain, which is clearly unfeasible in practice. However, there could be small constraints to be applied and that allow building an intrusion detector with a wide (albeit not complete) range of applicability.

I2: Monitoring. Regardless of the purpose, type and domain of a system, it is not possible to conduct intrusion detection without monitoring specific attributes, areas, components or layers of the system itself. Monitoring activities collect the value of performance indicators of a system at given time-instants or when specific events occur: examples include memory usage [19, 20], the throughput of buses [20], and system calls [21]. Also, those indicators can be gathered at hardware or low-level [22], system-level [20, 21], input/sensor [24], environment [19], or even coding-level [23]. Specifically for intrusion detectors, indicators to be monitored are usually related to network usage: this reduces the uncertainty regarding where a specific indicator is going to be monitored. Unfortunately, different network monitors may provide different indicators, or similar indicators with different measure units or sampling process, which still complicates the data analysis process.

I3: Feature Extraction and Learning. The baseline upon which intrusion detectors learn how to assign the "normal" or "anomaly/attack" binary label to a data point depends on the features, which are defined as "*individual measurable properties or characteristics of a phenomenon being observed*" [17]. Feature values related to the state of the system at a given instant build a *data point*: collections of data points are typically in the form of tabular datasets. Each data point contains values for each feature engineered from monitored system indicators. Additional attributes, called meta-features, can be further extracted from the corresponding dataset during the process [18]. Not all features help in distinguishing between normal or anomalous data points, whereas some of them may just represent noise. The importance of a thorough understanding of the underlying data and their features, as well as the produced results, is stressed in [1].

Learning complex features from the training dataset is of utmost importance, especially in deep learners that exercise a backbone [16] composed of convolutional and pooling layers, and forward its outputs to the connected layers that learn how the value of those features is linked to either normal or anomalous behavior due to attacks.

I4: Availability and Quality of Data. It is of no surprise that the amount [7] and noise [9, 13] contained in training data heavily affect the model building and consequently the whole detection task. Relying on a small training data set may result in underfitting [8] the model: this means that the model was created using poor or insufficient knowledge and will not be accurate nor general. In addition, data pre-processing (or the ML algorithm itself) should minimize uncertainty due to noisy labels [9] or in the training set [13]: a noisy item should not have a major impact on the way an ML algorithm learns a model, or on the way the model is used to assign labels to novel instances.

I5: Learning Process. ML algorithms are trained using a training dataset [3], which contains data points and the associated labels describing the binary class of each point. When the model learned from the ML algorithm is not general, even small perturbations can cause classifiers with high accuracy to produce an incorrect prediction on novel samples [4]. Overfitting [8] happens when a classifier learns a model that corresponds too closely or exactly to a particular set of data, and may therefore fail to generalize to a different, albeit similar, input set.

Throughout years, ML algorithms and especially deep learners were made more and more robust to overfitting through techniques such as pruning [15], early stopping [12], batch normalization [5], dropout regularization [6], conjugate gradient [11], and weight decay [10]. Altogether, those techniques are necessary to build models which have satisfying generalization capabilities. Unfortunately, they are not sufficient, as "it is very difficult to make a detailed characterization of how well a specific hypothesis generated by a certain learning algorithm will generalize, in the absence of detailed information about the given problem instance" [14].

3 Architectures for Building Generalized Intrusion Detectors

All the issues above constitute severe obstacles in building a generalized Intrusion Detector. However, there are efforts that could be made to overcome some of them and mitigate the negative impact of other issues.

3.1 Dealing with Generalization Issues

The Domain and Purpose (I1) of the system will impact how the intrusion detector will work regardless of all the efforts we could put. Intrusions will be at least partially related to the target system: in this study, we limit the uncertainty on this aspect by assuming that an intrusion detector is a binary classifier, which raises an alert if notices something unexpected in the data flow. The data flow of performance indicators comes from Monitoring (I2) activities: each system has its own monitoring strategy we do not have control about. Whereas it is likely that network indicators will be monitored through state of the art (and usually open source) tools such as Wireshark, Nagios, Prometheus, Zabbix, CICFlowMeter or slight variations of them, we cannot reasonably assume to know how many and which indicators are going to be monitored for a given system. However, we can manage the way we Extract and Learn Features (I3) from those data, to provide the intrusion detector with a set of features of constant and predefined amount. This will require exercising a system-dependent activity that processes monitored performance indicators (PI) to extract a fixed amount of features to be fed into the intrusion detector which is therefore decoupled from the target system.

This way, it is possible to gather data from different systems or existing datasets, merge them and build training and test datasets for intrusion detection that contain far more data instances. This helps also with the issue of availability (I4) of data to make the intrusion detector learn how to distinguish between normal and attack-related data. This learning process (I5) is at this point may even be completely decoupled from the target system(s), providing the system architect with extreme freedom in choosing the binary classifier that has the best potential for building an accurate intrusion detector.

3.2 Feature Mapping and Feature Learning

Let us explore how we deal with I3 with the aid of Fig. 1. On top of the figure we find three different sample target systems, each running a monitoring strategy that gathers heterogeneous sets of performance indicators, whose cardinality may be different (size *a, b, c* in the figure). As a result, the intrusion detector cannot assume to know the contents and the size of the feature set. This requires crafting a *System-Dependent Feature Processing* layer that is in charge of processing performance indicators to build a feature set that contains a fixed amount of features and with known content, regardless of the size and the contents of the monitored indicators from the target system. We foresee two possible software architectures to implement this activity:

- Feature Mapping (on the left of Fig. 1): the first option creates a mapping function that processes the set of performance indicators and maps them into a pre-defined set of features of fixed length m $\{F_1, F_2, \ldots F_m\}$. Suppose you want to process performance indicators to build a set of 4 features ($m = 4$) $\{F_1 =$ protocol, $F_2 =$ packet size, $F_3 =$ packet length, $F_4 =$ header flags$\}$. The feature mapper should process performance indicators of a system or a dataset to extract those features: clearly, the mapper depends on the target system since it has to know details about performance indicators and then derive the mapping function to the defined feature set.
- Feature Learning (on the right of Fig. 1): differently, we can exercise an additional layer of ML that does not aim at classifying, but is instead directed to learn a fixed amount of features from the heterogeneous sets of performance indicators. Learned features will then be provided to the intrusion detector for the second level of learning: in other words, we are building a stacking meta-learner [27]. This approach employs a set of *k* ML algorithms that are trained using the specific set of performance indicators PI of a given system (and thus feature learning is system-dependent), whose output has a fixed cardinality, regardless of the size of the input indicators. The outputs of all the k ML algorithms are then assembled to build a feature set of n features $\{F_1, F_2, \ldots F_n\}$. For example, we could employ $k = 3$ ML algorithms: two binary classifiers BC1 and BC2 each of them outputting two probabilities pN (probability of data being normal) and pA (probability of data being an attack), and a deep learner DL we use as backbone, extracting the 4 features it generates after convolutional and pooling layers. This generates a set of $n = 8$ features {BC1_pN, BC1_pA, BC2_pN, BC2_pA, DL_F1, DL_F2, DL_F3, DL_F4}, which has constant size regardless of the input performance indicators.

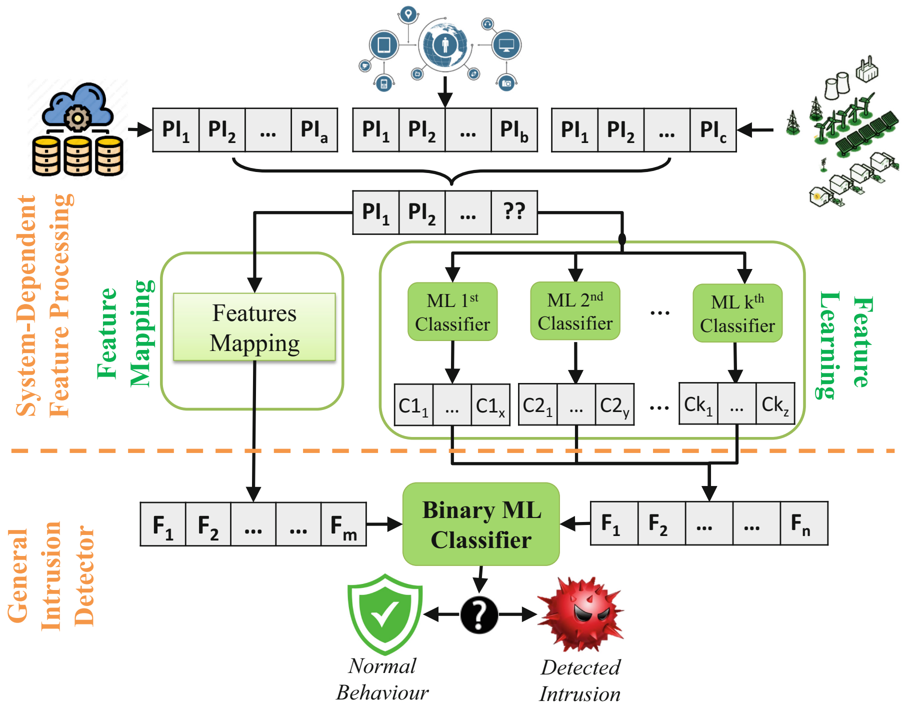

Fig. 1. Architectures for building a General Intrusion Detector. Regardless of the size of the feature set gathered from different systems (on top), exercising either Feature Mapping (on the left) or Feature Learning provides the Binary ML classifier (bottom of the figure) in charge of detecting intrusions with a feature set that has constant size.

3.3 Discussion, Advantages and Disadvantages

Those two software architectures have their strengths and weaknesses.

Feature mapping is clearly faster to execute and does not involve training ML algorithms (other than the binary classifier) which may be a time-consuming and also a complex task that involves optimizations, sensitivity analyses, and many more. On the downside, mapping performance indicators into a set of features may lead to loss of information, be very tricky and often unfeasible. For example, the NGIDS [50] dataset has only 3 features (process_id, syscall, event_id) as well as ADFANet [50] (packets, bytes, duration), whereas the CICIDS17, CICIDS18, AndMal17, and SDN20 share the same feature set of 77 network indicators. Finding a feature set that can convey most of the information contained in those datasets is not possible at all as there are no overlapping indicators between NGDIS and the other datasets. Even excluding NGDIS, ADFANet has far less indicators that other datasets and therefore it is very difficult to map all datasets into a unique feature set without losing information. It follows that this approach should be preferred whenever it is possible to tune the monitoring system to extract relevant indicators, while it is less feasible when detecting intrusions in existing datasets or in systems with non-customizable monitoring strategies.

Differently, Feature Learning is a complex process that abstracts from all those problems, which are masked by the learning process of ML algorithms used for feature learning. Moreover, it is a flexible approach since the amount and the type of feature learners can be tuned depending on the needs of the user e.g., the computational power available for intrusion detection.

In the rest of the paper we will build an experimental campaign to quantify the generalization capabilities of intrusion detectors that use either of those two architectures. Since our data baseline can be only composed of existing datasets, we will choose those that were built using the same monitoring strategy and tooling and have overlapping feature sets to make feature mapping feasible.

4 Experimental Campaign

4.1 Datasets Collection

There is a wide variety of tabular datasets related to intrusion detection, ranging from device data in Internet-of-Things (IoT) systems to network data for intrusion detection [28, 29]. Those often have heterogeneous feature sets which may not fit our exploratory study. Instead, AndMal17 [32], CICIDS17 [31], CICIDS18 [31], and SDN20 [30] were collected using the same network monitoring tool and as such fit our analysis. Table 1 summarizes the datasets considered in this study, reporting domain, name, publication year, number of data points, number of features, types, and percentages of attacks. Those datasets are quite recent (not older than 2017), and they are well-known reference datasets in the domain. Regarding the attacks logged in the datasets, (Distributed) Denial of Service and scanning attacks (e.g., probing, port scanning, reconnaissance) appear in all datasets but AndMal17. Other attacks such as malware (AndMal17, SDN20), web attacks (CICIDS17, CICIDS18), botnets (CICIDS18), spam (AndMal17) and phising (AndMal17) occur in a few datasets. Overall, these 4 datasets provide a view on most of the common attacks in the current threat landscape [25] and therefore we believe they provide a representative data baseline to experiment on. Also, the reader should notice that different datasets log system behavior under different attacks and therefore diversity among datasets is dual: both from the target system and the attack model standpoints.

Table 1. Selected datasets: name, reference, release year, size, number of features, number and percentage of attacks.

Dataset name	Ref	Year	# Data points used	# Features	# Attacks	% Attacks
AndMal17	[32]	2017	100 000	77	4	15.5
CICIDS17	[31]	2017	500 000	77	5	79.7
CICIDS18	[31]	2018	200 000	77	8	26.2
SDN20	[30]	2020	205 167	77	5	66.6

4.2 Binary Classifiers for Intrusion Detection

We then choose the candidate binary classifiers for implementing the intrusion detector. We do not aim at identifying a complete and broad set of classifiers: instead, we want to use those that were widely used in the literature and that were proven to be effective for tabular data. We ended up selecting Random Forests (RF, [35]), eXteme Gradient Boosting (XGB, [36]) and the deep learner FastAI (FAI, [37]), which has optimizations for tabular data. Random Forests are a well-known bagging ensemble of decision trees that saw a lot of applicability for intrusion detection in the last decade [33], while XGBoost has proven to outperform many classifiers including deep learners [34] for tabular data. Lastly, FastAI contains optimizations for processing tabular data and entity embedding of categorical features.

4.3 ML Algorithms to Be Used for Feature Learning

Feature learners to be used in this study can be essentially any ML algorithm: supervised, unsupervised, backbone deep learner, and so on and so forth. Since this is an exploratory study, we aim at exercising as many feature learners as possible: then, we may filter out those that learn weak features and keep only those that learn the strong ones. In our study, each feature learner learns two features, which are the probability of being a normal data point, or the probability of being an attack. Summarizing, this study employs 16 feature learners, that learn a total of 32 features (2 each):

- 10 unsupervised ML algorithms from the library PYOD [38], namely: ECOD, COPOD, FastABOD, HBOS, MCD, PCA, LOF, CBLOF, Isolation Forests, SUOD.
- 5 supervised ML algorithms from Scikit-Learn [39], different from those used for intrusion detection in the previous section: k-th Nearest Neighbors, ADABoost, Naïve Bayes, Logistic Regression, Linear Discriminant Analysis.
- A deep learner used as backbone for feature learning (FastAI), which as motivated before contains suitable optimizations to learn features from tabular data.

4.4 Experimental Setup and Methodology

Experiments are executed on a Dell Precision 5820 Tower with an Intel I9-9920X, GPU NVIDIA Quadro RTX6000 with 24 GB VRAM, 192 GB RAM, and Ubuntu 18.04, and they required approximately 6 weeks of 24 h execution.

The *Pyod*, *Scikit-Learn* and *xgboost* python packages contain all the code needed to exercise ML algorithms. We created a Python script to load datasets, orchestrate feature learners, train and evaluate intrusion detectors. The evaluation will mainly be carried out by means of evaluation metrics for binary classification i.e., confusion matrix [40] and especially using aggregated metrics as Accuracy and Matthews Correlation Coefficient (MCC). Additionally, we compute the importance that intrusion detectors assign to their features: those will help to break down the behavior of different intrusion detectors and provide insights on the way they build their models. We split each of the dataset in half (50–50 train-test split) and perform 5 series of experiments, which we explain below and partially depict in Fig. 2:

- RegularID: we exercise the ML algorithms Random Forests (RF), XGBoost (XGB), and FastAI (FAI) on all datasets separately using the 50–50 train-test split and collect metric scores. This is the usual way of training and evaluating an intrusion detector, which is entirely system-dependent (i.e., not general).
- FeatL: we exercise the 16 feature learners on the train portion of each dataset but the one used for testing, collecting their outputs. Those build a huge training set composed of data instances with homogeneous structure (i.e., each of those data points has 32 feature values), even if they come from different datasets. Those are used to train the intrusion detectors RF, XGB, FAI individually. For example, when testing the dataset AndMal17, we train the detector using the train partition of CICIDS17, CICID18 and SDN20 (i.e., without using AndMal17 at all). The resulting model is then used to detect intrusions using the test portion of AndMal17, which is completely unknown to the intrusion detector. This quantifies how well the detector generalizes to a different dataset.
- FeatL_TL: This is a process similar to FeatL, but it is not completely unrelated from the dataset used for testing. Particularly, we partially use the train partition of the dataset we want to evaluate to re-train the FeatL detector using transfer learning mechanics. This way, the binary ML classifier gets tailored using some key information about the system under test and is expected to have better classification performance than FeatL, at a cost of a less general model. We will use either 1000, 5000, 10000, 20000 data points for transfer learning, labeling the corresponding detector as FeatL_TL1, FeatL_TL5, FeatL_TL10, FeatL_TL20.
- Map: it is a process similar to FeatL, but does not execute Feature Learning. Instead, it maps directly features from different datasets to the same feature set, since the 4 datasets in this study all share the exact same feature set.
- Map_TL: it is a process similar to FeatL_TL, but does not execute Feature Learning. Instead, it maps directly features from different datasets to the same feature set, since the 4 datasets in this study all share the exact same feature set.

5 Results and Discussion

5.1 Regular, Feature Learning and Feature Mapping Intrusion Detectors

We start analyzing results with the aid of Table 2. The table reports the highest MCC achieved either by RF, XGB, or FAI for a given intrusion detector: RegularID, FeatL Map, FeatL_TL20, Map_TL20. We chose the TL20 variants of the FeatL_TL and Map_TL as they were delivering higher MCC than their counterparts which are using less data for transfer learning. It turns out evident how RF and XGB are the preferred ML algorithm for intrusion detection in most of the cases: they achieve the highest MCC on most configurations reported in the table. Also, the AndMal17 dataset is the hardest of the four to perform detection on: while for CICIDS17, CICIDS18 and SDN20 we have MCC scores over 0.90, for AndMal17 the MCC does not exceed 0.65, that corresponds to an accuracy of 92.9 and a Recall of 48.3 (i.e., more than half of the attacks, the 51.7%, are not detected by the intrusion detector).

Going into the detail of the 5 different intrusion detectors we instantiated in this study, we can observe that – as expected – RegularID has the highest MCC being specific of

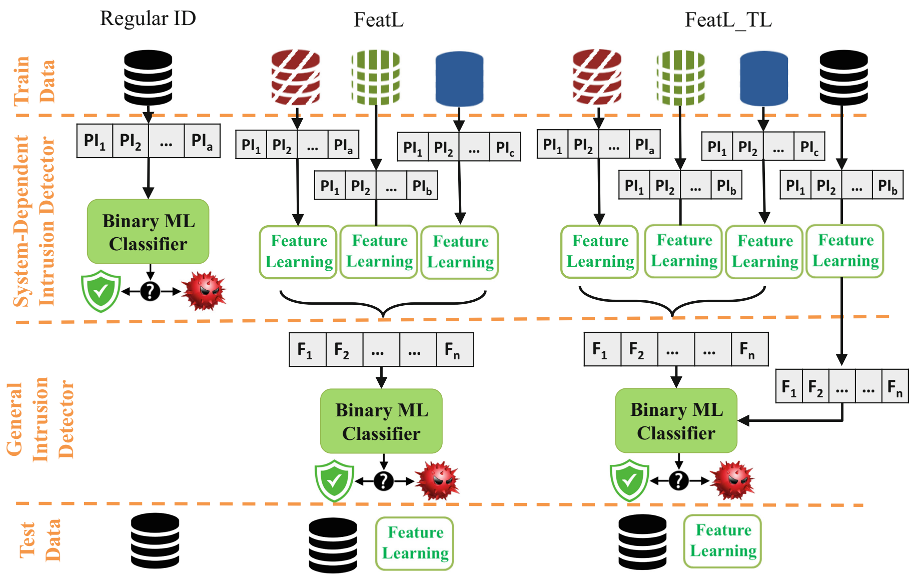

Fig. 2. Experiments for building a RegularID, FeatL and FeatL_TL intrusion detectors, separating the system-dependent from the general part of those detectors. The Map and Map_TL detectors work the same as the FeatL and FeatL_TL, but do not perform feature learning.

a dataset and with no generalization capabilities. Using only 3 datasets for training a unique model to be tested on another unknown dataset, either by mapping features (Map in Table 2) or by feature learning (FeatL in the table) generates MCC scores that are far lower than those of RegularID. Map scores are not even comparable with others, whereas FeatL scores are better than those of Map but still noticeably lower than RegularID in all datasets but SDN20, making those general detectors not applicable in a real setup due to an excessive amount of False Positives and/or False Negatives. Scores of Map_TL20 and FeatL_TL20 are clearly better than those of Map and FeatL, but still lower than those of RegularID: additionally, transfer learning limits the generalization capabilities of those detectors as it adds another system-specific training component.

Nevertheless, it is interesting to observe the impact transfer learning has on MCC scores. We discuss this aspect with the aid of Fig. 3, which also allows remarking the following important observations:

- Adopting transfer learning clearly improves capabilities of intrusion detectors: Map_TL1 has better MCC than Map, and the MCC grows the more data is used for transfer learning (i.e., Map_TL20 has better MCC than Map_TL10, which is better than Map_TL5, which outperforms Map_TL1). The same applies to FeatL and FeatL_TL.
- Transfer learning has an outstanding impact when using detectors relying on feature mapping. Map detectors have very poor scores, but improve dramatically even when only 1000 data points are used for transfer learning (Map_TL1). This can be observed in Fig. 3a and 3b looking at the two series of bars on the bottom of each bar chart.

Table 2. MCC scores of the best ML algorithm (FAI, RF, XGB) used as regular ID, FeatL, Map, FeatL_TL20, Map_TL20.

Dataset	Map		Map_TL20		FeatL		FeatL_TL20		RegularID	
AndMal17	0.023	XGB	0.251	XGB	0.313	FAI	0.453	XGB	0.647	RF
CICIDS17	0.626	XGB	0.993	XGB	0.975	FAI	0.987	RF	0.999	XGB
CICIDS18	0.260	FAI	0.853	XGB	0.890	RF	0.908	RF	0.928	XGB
SDN20	0.180	XGB	0.999	XGB	0.999	RF	0.999	RF	1.000	RF
Average MCC	**0.272**		**0.774**		**0.794**		**0.837**		**0.893**	

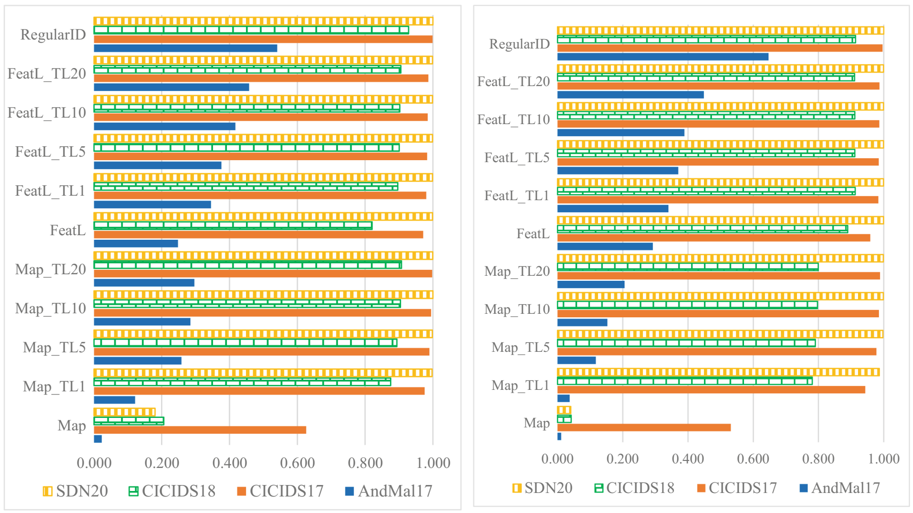

Fig. 3. a (left) and b (right). MCC scores for each of the four datasets (one bar series each). Each bar chart has 11 series of bars, one for each intrusion detector. Scores using XGB are on the left (Fig. 3a), while scores using RF are on the right (Fig. 3b).

- The FeatL detectors have overall better performance than Map and therefore their performance improvement with transfer learning is less evident than those of Map. Nevertheless, applying transfer learning brings FeatL_TL20 to achieve MCC scores that are very similar to those of RegularID scores. This is an important results because it shows how it is possible to achieve good detection performance tailoring an existing model rather than crafting an intrusion detection from scratch, saving key amount of time and thus money.

5.2 On the Contribution of Feature Learners

FeatL has better scores than Map: this is due to the feature learners, which are trained using a small portion of the novel system under test to extract features. We explain the contribution each feature learner has on the overall detection process with the aid of Table 3, which presents the importance of feature learners for FeatL and FeatL_TL20 using either RF or XGB on the CICIDS18 dataset. Importance in each row of the table sum up to 1, while each score ranges between 0 and 1: the higher, the most relevant features learned from a feature learner are for training the intrusion detector. Additionally, we report the difference in the importance of features between the FeatL_TL20 and the FeatL, which does not apply transfer learning. The importance using XGB or RF follow a similar path: the FeatL detector learns a model that is almost entirely built over features learned by KNN, which has 0.920 and 0.911 importance respectively for XGB and RF. Other feature learners have marginal to negligible contribution, making the FeatL detectors very dependent on the behavior of KNN features.

Table 3. Importance of feature learners in building the model for FeatL and FeatL_TL20 using either XGB or RF as ML algorithms for the CICIDS18 dataset.

		Unsupervised Feature Learn.										**Supervised Feature Learn.**					
ML Algorithm	**Intrusion Detector**	**COPOD**	**ABOD**	**HBOS**	**MCD**	**PCA**	**ECOD**	**LOF**	**CBLOF**	**IForest**	**SUOD**	**Naïve Bayes**	**LDA**	**Logistic Reg**	**KNN**	**FAI**	**ADABoost**
XGB	FeatL	.001	.030	.003	.002	.003	.001	.002	.003	.002	.002	.003	.002	.000	.920	.020	.003
	FeatL_TL20	.009	.012	.003	.001	.013	.007	.001	.005	.005	.007	.033	.088	.000	.455	.221	.139
	Diff	**.007**	**-.018**	**.000**	**-.002**	**.011**	**.005**	**-.002**	**.002**	**.004**	**.005**	**.030**	**.086**	**.000**	**-.465**	**.201**	**.136**
RF	FeatL	.001	.032	.004	.003	.003	.002	.002	.003	.002	.002	.003	.002	.000	.911	.021	.004
	FeatL_TL20	.010	.006	.004	.002	.011	.011	.001	.007	.007	.008	.027	.107	.000	.352	.327	.108
	Diff	**.009**	**-.026**	**.000**	**-.001**	**.008**	**.009**	**-.001**	**.004**	**.006**	**.006**	**.024**	**.105**	**.000**	**-.559**	**.306**	**.105**

Differently, the FeatL_TL20 models obtained using 20 000 data points of CICIDS18 (the system under test for building this table) for transfer learning do not rely entirely on KNN to detect intrusions. The importance of KNN features decreases a lot, favoring FAI, ADABoost and LDA features. Other feature learners, especially those unsupervised, still have very marginal contribution to the overall detection process.

Overall, it is safe to say that transfer learning makes XGB and RF learn a model that does not heavily depends on a single feature learner, but instead combines the output of different feature learners: this results in a more general model.

6 Conclusions and Future Works

In this study we proposed and experimentally evaluated two software architectures for building intrusion detectors that have good generalization capabilities. Briefly, we aimed at learning the model once and then apply it to as many datasets the user wants with minimal effort, still achieving satisfying detection performance.

Our experimental results are not fully encouraging: they tell us that no matter the intrusion detector, it will not generalize well to other datasets as is. Instead, it will be outperformed by system-specific intrusion detectors, confirming the studies [2, 49]. A non-zero amount of knowledge about the system under test is indeed required to make intrusion detectors able to detect intrusions in other datasets with satisfactory performance. Knowing only a few thousands of data points of the system under test allowed intrusion detectors reaching satisfying detection scores in our experiments, without outperforming traditional system-specific intrusion detectors. It follows that tailoring a baseline model through transfer learning has the potential to obtain satisfactorily (albeit not optimal) detection performance, requiring less data and minimal expertise from the user standpoint, which does not have to train multiple ML algorithms nor running complex performance evaluations.

Particularly, pre-processing datasets through different ML algorithms deployed as feature learners clearly builds an intrusion detector that potentially has generalization capabilities. Therefore, as future works we want to elaborate more on those detectors with respect to three dimensions of analysis, namely: i) carefully selecting feature learners to be used, ii) gathering training data from more datasets, hoping to build a detector which is more solid and as such has better generalization capabilities, and iii) performing sensitivity analyses aiming at clearly identifying the minimum amount of data which we have to gather from the system under test to train feature learners and tailor the detector to achieve satisfactory detection performance.

Acknowledgments. This work has been partially supported by the H2020 Marie Sklodowska-Curie g.a. 823788 (ADVANCE), by the Regione Toscana POR FESR 2014–2020 SPaCe, and by the NextGenerationEU programme, Italian DM737 – CUP B15F21005410003.

References

1. Sommer, R., Paxson, V.: Outside the closed world: on using machine learning for network intrusion detection. In: 2010 IEEE Symposium on Security and Privacy, pp. 305–316. IEEE, May 2010
2. Catillo, M., Del Vecchio, A., Pecchia, A., Villano, U.: Transferability of machine learning models learned from public intrusion detection datasets: the CICIDS2017 case study. Softw. Qual. J. **30**, 955–981 (2022). https://doi.org/10.1007/s11219-022-09587-0
3. Schmidhuber, J.: Deep learning in neural networks: an overview. Neural Netw. **61**, 85–117 (2015)
4. Schmidt, L., Santurkar, S., Tsipras, D., Talwar, K., Madry, A.: Adversarially robust generalization requires more data. In: Advances in Neural Information Processing Systems, vol. 31 (2018). Accessed 07 Apr 2022
5. Li, Y., Wang, N., Shi, J., Liu, J., Hou, X.: Revisiting batch normalization for practical domain adaptation, November 2016. http://arxiv.org/abs/1603.04779. Accessed 07 Apr 2022
6. Jindal, I., Nokleby, M., Chen, X.: Learning deep networks from noisy labels with dropout regularization. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 967–972, December 2016. https://doi.org/10.1109/ICDM.2016.0121
7. Chen, X.W., Lin, X.: Big data deep learning: challenges and perspectives. IEEE Access **2**, 514–525 (2014)
8. Lawrence, S., Giles, C.L.: Overfitting and neural networks: conjugate gradient and backpropagation. In: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks, IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium, vol. 1, pp. 114–119. IEEE, July 2000

9. Song, H., et al.: Learning from noisy labels with deep neural networks: a survey. IEEE Trans. Neural Netw. Learn. Syst. (2022, article in press). https://doi.org/10.1109/TNNLS.2022.3152527
10. Krogh, A., Hertz, J.: A simple weight decay can improve generalization. In: Advances in Neural Information Processing Systems, vol. 4 (1991)
11. Caruana, R., Lawrence, S., Giles, C.: Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. In: Advances in Neural Information Processing Systems, vol. 13 (2000)
12. Prechelt, L.: Early stopping - but when? In: Orr, G.B., Müller, K.-R. (eds.) Neural Networks: Tricks of the trade. LNCS, vol. 1524, pp. 55–69. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49430-8_3
13. Sietsma, J., Dow, R.J.: Creating artificial neural networks that generalize. Neural Netw. **4**(1), 67–79 (1991)
14. Kawaguchi, K., Kaelbling, L.P., Bengio, Y.: Generalization in deep learning. arXiv preprint arXiv:1710.05468 (2017)
15. Cestnik, B., Bratko, I.: On estimating probabilities in tree pruning. In: Kodratoff, Y. (ed.) Machine Learning — EWSL-91: European Working Session on Learning Porto, Portugal, March 6–8, 1991 Proceedings, pp. 138–150. Springer, Heidelberg (1991). https://doi.org/10.1007/BFb0017010
16. Gao, S.H., Cheng, M.M., Zhao, K., Zhang, X.Y., Yang, M.H., Torr, P.: Res2Net: a new multi-scale backbone architecture. IEEE Trans. Pattern Anal. Mach. Intell. **43**(2), 652–662 (2019)
17. Bishop, C.: Pattern Recognition and Machine Learning. Springer, Berlin (2006). ISBN: 0-387-31073-8
18. Rivolli, A., Garcia, L.P., Soares, C., Vanschoren, J., de Carvalho, A.C.: Meta-features for meta-learning. Knowl.-Based Sys. **240**, 108101 (2022)
19. Cotroneo, D., Natella, R., Rosiello, S.: A fault correlation approach to detect performance anomalies in Virtual Network Function chains. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), pp. 90–100. IEEE (2017)
20. Zoppi, T., Ceccarelli, A., Bondavalli, A.: MADneSs: a multi-layer anomaly detection framework for complex dynamic systems. IEEE Trans. Dependable Secure Comput. **18**(2), 796–809 (2019)
21. Murtaza, S.S., et al.: A host-based anomaly detection approach by representing system calls as states of kernel modules. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). IEEE (2013)
22. Wang, G., Zhang, L., Xu, W.: What can we learn from four years of data center hardware failures? In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 25–36. IEEE, June 2017
23. Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z.: SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. **19**(4), 2244–2258 (2022)
24. Robles-Velasco, A., Cortés, P., Muñuzuri, J., Onieva, L.: Prediction of pipe failures in water supply networks using logistic regression and support vector classification. Reliab. Eng. Syst. Saf. **196**, 106754 (2020)
25. Ardagna, C., Corbiaux, S., Sfakianakis, A., Douliger, C.: ENISA Threat Landscape 2021. https://www.enisa.europa.eu/topics/threat-risk-management/threats-and-trends. Accessed 6 May 2022
26. Connell, B.: 2022 SonicWall Threat Report. https://www.sonicwall.com/2022-cyber-threat-report/. Accessed 6 May 2022
27. Džeroski, S., Ženko, B.: Is combining classifiers with stacking better than selecting the best one? Mach. Learn. **54**(3), 255–273 (2004). https://doi.org/10.1023/B:MACH.0000015881.36452.6e

28. Khraisat, A., Gondal, I., Vamplew, P., Kamruzzaman, J.: Survey of intrusion detection systems: techniques, datasets, and challenges. Cybersecurity **2**(1) (2019). Article number: 20. https://doi.org/10.1186/s42400-019-0038-7
29. Ring, M., Wunderlich, S., Scheuring, D., Landes, D., Hotho, A.: A survey of network-based intrusion detection data sets. Comput. Secur. **86**, 147–167 (2019)
30. Elsayed, M.S., Le-Khac, N.A., Jurcut, A.D.: InSDN: a novel SDN intrusion dataset. IEEE Access **8**, 165263–165284 (2020)
31. Sharafaldin, I., et al.: Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: ICISSP, pp. 108–116, January 2018
32. Lashkari, A.H., et al.: Toward developing a systematic approach to generate benchmark Android malware datasets and classification. In: 2018 International Carnahan Conference on Security Technology (ICCST), pp. 1–7. IEEE, October 2018
33. Resende, P.A.A., Drummond, A.C.: A survey of random forest based methods for intrusion detection systems. ACM Comput. Surv. (CSUR) **51**(3), 1–36 (2018)
34. Shwartz-Ziv, R., Armon, A.: Tabular data: deep learning is not all you need. Inf. Fusion **81**, 84–90 (2022)
35. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
36. Chen, T., et al.: XGBoost: eXtreme gradient boosting. R Package Version 0.4-2, **1**(4), 1–4 (2015)
37. Howard, J., Gugger, S.: Fastai: a layered API for deep learning. Information **11**(2), 108 (2020)
38. Zhao, Y., Nasrullah, Z., Li, Z.: PyOD: a python toolbox for scalable outlier detection. arXiv preprint arXiv:1901.01588 (2019)
39. Buitinck, L., et al.: API design for machine learning software: experiences from the scikit-learn project. arXiv preprint arXiv:1309.0238 (2013)
40. Luque, A., et al.: The impact of class imbalance in classification performance metrics based on the binary confusion matrix. Pattern Recogn. **91**, 216–231 (2019)
41. Ucci, D., Aniello, L., Baldoni, R.: Survey of machine learning techniques for malware analysis. Comput. Secur. **81**, 123–147 (2019)
42. Demetrio, L., et al.: Adversarial exemples: a survey and experimental evaluation of practical attacks on machine learning for windows malware detection. ACM Trans. Priv. Secur. (TOPS) **24**(4), 1–31 (2021)
43. Zhauniarovich, Y., Khalil, I., Yu, T., Dacier, M.: A survey on malicious domains detection through DNS data analysis. ACM Comput. Surv. (CSUR) **51**(4), 1–36 (2018)
44. Oliveira, R.A., Raga, M.M., Laranjeiro, N., Vieira, M.: An approach for benchmarking the security of web service frameworks. Future Gener. Comput. Syst. **110**, 833–848 (2020)
45. Andresini, G., Appice, A., Malerba, D.: Autoencoder-based deep metric learning for network intrusion detection. Inf. Sci. **569**, 706–727 (2021)
46. Apruzzese, G., Colajanni, M., Ferretti, L., Guido, A., Marchetti, M.: On the effectiveness of machine and deep learning for cyber security. In: 2018 10th International Conference on Cyber Conflict (CyCon), pp. 371–390. IEEE, May 2018
47. Folino, F., et al.: On learning effective ensembles of deep neural networks for intrusion detection. Inf. Fusion **72**, 48–69 (2021)
48. Arp, D., et al.: Dos and don'ts of machine learning in computer security. In: Proceedings of the USENIX Security Symposium, August 2022
49. Verkerken, M., D'hooge, L., Wauters, T., Volckaert, B., De Turck, F.: Towards model generalization for intrusion detection: unsupervised machine learning techniques. J. Netw. Syst. Manag. **30** (2022). Article number: 12. https://doi.org/10.1007/s10922-021-09615-7
50. Haider, W., Hu, J., Slay, J., Turnbull, B.P., Xie, Y.: Generating realistic intrusion detection system dataset based on fuzzy qualitative modeling. J. Netw. Comput. Appl. **87**, 185–192 (2017)