In Search of netUnicorn: A Data-Collection Platform to Develop Generalizable ML Models for Network Security Problems

Roman Beltiukov
rbeltiukov@ucsb.edu
UC Santa Barbara
California, USA

Wenbo Guo
henrygwb@purdue.edu
Purdue University
Indiana, USA

Arpit Gupta
agupta@ucsb.edu
UC Santa Barbara
California, USA

Walter Willinger
wwillinger@niksun.com
NIKSUN, Inc.
New Jersey, USA

ABSTRACT

The remarkable success of the use of machine learning-based solutions for network security problems has been impeded by the developed ML models' inability to maintain efficacy when used in different network environments exhibiting different network behaviors. This issue is commonly referred to as the generalizability problem of ML models. The community has recognized the critical role that training datasets play in this context and has developed various techniques to improve dataset curation to overcome this problem. Unfortunately, these methods are generally ill-suited or even counterproductive in the network security domain, where they often result in unrealistic or poor-quality datasets.

To address this issue, we propose a new closed-loop ML pipeline that leverages explainable ML tools to guide the network data collection in an iterative fashion. To ensure the data's realism and quality, we require that the new datasets should be endogenously collected in this iterative process, thus advocating for a gradual removal of data-related problems to improve model generalizability. To realize this capability, we develop a data-collection platform, netUnicorn, that takes inspiration from the classic "hourglass" model and is implemented as its "thin waist" to simplify data collection for different learning problems from diverse network environments. The proposed system decouples data-collection intents from the deployment mechanisms and disaggregates these high-level intents into smaller reusable, self-contained tasks. We demonstrate how netUnicorn simplifies collecting data for different learning problems from multiple network environments and how the proposed iterative data collection improves a model's generalizability.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Security and privacy**; • **Networks** → **Network measurement**;

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623075

KEYWORDS

Network Security, Machine Learning, Artificial Intelligence, Data Collection, Generalizability

ACM Reference Format:
Roman Beltiukov, Wenbo Guo, Arpit Gupta, and Walter Willinger. 2023. In Search of netUnicorn: A Data-Collection Platform to Develop Generalizable ML Models for Network Security Problems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623075

1 INTRODUCTION

Machine learning-based methods have outperformed existing rule-based approaches for addressing different network security problems, such as detecting DDoS attacks [68], malwares [2, 10], network intrusions [37], etc. However, their excellent performance typically relies on the assumption that the training and testing data are independent and identically distributed. Unfortunately, due to the highly diverse and adversarial nature of real-world network environments, this assumption does not hold for most network security problems. For instance, an intrusion detection model trained and tested with data from a specific environment cannot be expected to be effective when deployed in a different environment, where attack and even benign behaviors may differ significantly due to the nature of the environment. This inability of existing ML models to perform as expected in different deployment settings is known as *generalizability problem* [32], poses serious issues with respect to maintaining the models' effectiveness after deployment, and is a major reason why security practitioners are reluctant to deploy them in their production networks in the first place.

Recent studies (e.g., [7]) have shown that the quality of the training data plays a crucial role in determining the generalizability of ML models. In particular, in popular application domains of ML such as computer vision and natural language processing [100, 107], researchers have proposed several data augmentation and data collection techniques that are intended to improve the generalizability of trained models by enhancing the diversity and quality of training data [51]. For example, in the context of image processing, these techniques include adding random noise, blurring, and linear interpolation. Other research efforts leverage open-sourced datasets collected by various third parties to improve the generalizability of text and image classifiers.

Unfortunately, these and similar existing efforts are not directly applicable to network security problems. For one, since the semantic constraints inherent in real-world network data are drastically different from those in text or image data, simply applying existing augmentation techniques that have been designed for text or image data is likely to result in unrealistic and semantically incoherent network data. Moreover, utilizing open-sourced data for the network security domain poses significant challenges, including the encrypted nature of increasing portions of the overall traffic and the fact that without detailed knowledge of the underlying network configuration, it is, in general, impossible to label additional data correctly. Finally, due to the high diversity in network environments and a myriad of different networking conditions, randomly using existing data or collecting additional data without understanding the inherent limitations of the available training data may even reduce data quality. As a result, there is an urgent need for novel data curation techniques that are specifically designed for the networking domain and aid the development of generalizable ML models for network security problems.

To address this need, we propose a new closed-loop ML pipeline (workflow) that focuses on training generalizable ML models for networking problems. Our proposed pipeline is a major departure from the widely-used standard ML pipeline [32] in two major ways. First, instead of obscuring the role that the training data plays in developing and evaluating ML models, the new pipeline elucidates the role of the training data. Second, instead of being indifferent to the black-box nature of the trained ML model, our proposed pipeline deliberately focuses on developing explainable ML models. To realize our new ML pipeline, we designed it using a closed-loop approach that leverages a novel data collection platform (called netUnicorn[1]) in conjunction with state-of-the-art explainable AI (XAI) tools so as to be able to iteratively collect new training data for the purpose of enhancing the ability of the trained models to generalize. Here, during each iteration, the insights obtained from applying the employed explainability tools to the current version of the trained model are used to synthesize new policies for exactly what kind of new data to collect in the next iteration so as to combat generalizability issues affecting the current model.

In designing and implementing netUnicorn, the novel data collection platform that our proposed ML pipeline relies on, we leveraged state-of-the-art programmable data-plane targets, programmable network infrastructures, and different virtualization tools to enable flexible data collection at scale from disparate network environments and for different learning problems without network operators having to worry about the details of implementing their desired data collection efforts. This platform can be envisioned as representing the "thin waist" of the classic hourglass model [11], where the different learning problems comprise the top layer and the different network environments constitute the bottom layer. To realize this "thin waist" analog, netUnicorn supports a new programming abstraction that (i) decouples the data-collection intents or policies (i.e., answering what data to collect and from where) from the mechanisms (i.e., answering how to collect the desired data on a given platform); and (ii) disaggregates the high-level intents into self-contained and reusable subtasks.

[1] https://netunicorn.cs.ucsb.edu

In effect, our newly proposed ML pipeline advances the current state-of-the-art in ML model development by (1) augmenting the standard ML pipeline with an explainability step that impacts how we evaluate ML models' suitability for deployments, (2) leveraging existing explainable AI (XAI) tools to identify issues with the utilized training data that may affect a trained model's generalizability, and (3) utilizing insights from (2) to direct the netUnicorn-driven iterative data collection, thus gradually improving the models' generalizability using these new datasets. Unlike the standard "open-loop" ML pipelines, which often rely on synthetic data to bolster model generalizability or lack the means to gather data from varying network settings or distinct learning scenarios, our innovative closed-loop ML pipeline can consistently collect the "right" training data from diverse network conditions for any specified learning problem. This paper shows that the proposed ML pipeline's ability to iteratively collect the "right" training data from disparate network environments and for any given learning problem paves the way for the development of generalizable ML models for networking problems.

Contributions. This paper makes the following contributions:

- **An alternative ML pipeline.** We propose a novel closed-loop ML pipeline that leverages a new data-collection platform in conjunction with state-of-the-art explainability (XAI) tools to enable iterative and informed data collection to gradually improve the quality of the data used for model training and thus boost the trained models' generalizability (Section 2).
- **A new data-collection platform.** We justify (Section 3) and present the design and implementation (Section 4) of netUnicorn, the new data-collection platform that is key to performing iterative and informed data collection for any given learning problem and from any network environment as part of our newly proposed closed-loop ML pipeline in practice. We made several design choices in netUnicorn to tackle the research challenges of realizing the "thin waist" abstraction.
- **An extensive evaluation.** We demonstrate the capabilities of netUnicorn and the effectiveness of our newly proposed ML pipeline by (i) considering various learning models for network security problems that have been studied in the existing literature and (ii) evaluating them with respect to their ability to generalize (Section 5 and Section 6).
- **Artifacts.** We make the full source code of the system, as well as the datasets used in this paper, publicly available. Specifically, we have released three repositories: full source code of netUnicorn [73], a repository of all discussed tasks and data-collection pipelines [74], and other supplemental materials [75] (see [13], Appendix I).

We consider the proposed ML pipeline and data-collection platform a promising foundation for creating generalizable ML-based network security solutions more likely to see practical deployment. Still, significant work lies ahead. To ensure model generalizability, we must thoroughly assess the network infrastructure for data collection and the nature of traffic in production settings.

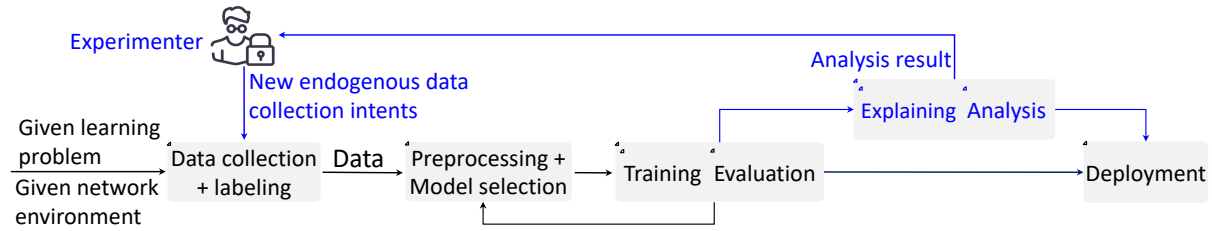


Figure 1: Overview of the existing (standard) and the newly-proposed (closed-loop) ML pipelines. The components marked in blue are our proposed augmentations to the standard ML pipeline.

2 BACKGROUND AND PROBLEM SCOPE

2.1 Existing ML Pipeline for Network Security

Key components. The standard ML pipeline (see Figure 1) defines a workflow for developing ML artifacts and is widely used in many application domains, including network security. To solve a learning problem (e.g., detecting DDoS attack traffic), the first step is to collect (or choose) labeled data, select a model design or architecture (e.g., random forest classifier), extract related features, and then perform model training using the training dataset. An independent and identically distributed (iid) evaluation procedure is then used to assess the resulting model by measuring its expected predictive performance on test data drawn from the training distribution. The final step involves selecting the highest-performing model from a group of similarly trained models based on one or more performance metrics (e.g., F1-score). The selected model is then considered the ML-based solution for the task at hand and is recommended for deployment and being used or tested in production settings.

Data collection mechanisms. As in other application areas of ML, the collection of appropriate training data is of paramount importance for developing effective ML-based network security solutions. In network security, the standard ML pipeline integrates two basic data collection mechanisms: *real-world network data* collection and *emulation-based network data* collection.

In the case of real-world network data collection, data such as traffic-specific aspects are extracted directly (and usually passively) from a real-world target network environment. While this method can provide datasets that reflect pertinent attributes of the target environment, issues such as encrypted network traffic and user privacy considerations pose significant challenges to understanding the context and correctly labeling the data. Despite an increasing tendency towards traffic encryption [23], this approach still captures real-world networking conditions but often restricts the quality and diversity of the resulting datasets.

Regarding emulation-based network data collection, the approach involves using an existing or building one's own emulated environment of the target network and generating (usually actively) various types of attack and benign traffic in this environment to collect data. Since the data collector has full control over the environment, it is, in general, easy to obtain ground truth labels for the collected data. While created in an emulated environment, the resulting traffic is usually produced by existing real-world tools. Many widely used network datasets, including the still-used DARPA1998 dataset [33] and the more recent CIC-IDS intrusion detection datasets [28] have been collected using this mechanism.

2.2 Model Generalizability Issues

Although existing emulation-based mechanisms have the benefit of providing datasets with correct labels, the training data is often riddled with problems that prevent trained models from generalizing, thus making them ill-suited for real-world deployment.

There are three main reasons why these problems can arise. First, network data is inherently complex and heterogeneous, making it challenging to produce datasets that do not contain inductive biases. Second, emulated environments typically differ from the target environment – without full knowledge of the target environment's configurations, it is difficult to accurately mimic it. The result is datasets that do not fully represent all the target environment's attributes. Third, shifting attack (or even benign) behavior is the norm, resulting in training datasets that become less representative of newly created testing data after the model is deployed.

These observations motivate considering the following concrete issues concerning the generalizability of ML-based network security solutions but note that there is no clear delineation between notions such as *credible*, *trustworthy* or *robust* ML models and that the existing literature tends to blur the line between these (and other) notions and what we refer to as model generalizability.

Shortcut learning. As discussed in [7], ML-based security solutions often suffer from shortcuts. Here, shortcuts refer to encoded/inductive biases in a trained model that stem from false or non-causal associations in the training dataset [42]. These biases can lead to a model not performing as desired in deployment scenarios, mainly because the test datasets from these scenarios are unlikely to contain the same false associations. Shortcuts are often attributable to data-collection issues, including how the data was collected (intent) or from where it was collected (environment). Recent studies have shown that shortcut learning is a common problem for ML models trained with datasets collected from emulated networking environments. For example, [56] found that the reported high F1-score for the VPN vs. non-VPN classification problem in [36] was due to a specific artifact of how this dataset was curated.

Out-of-distribution issues. Due to unavoidable differences between a real-world target environment and its emulated counterpart or subtle changes in attack and/or benign behaviors, out-of-distribution (ood) data is another critical factor that limits model generalizability. The standard ML pipeline's evaluation procedure results in models that may appear to be well-performing, but their excellent performance can often be attributed to the models' innate ability for "rote

learning", where the models cannot transfer learned knowledge to new situations. To assess such models' ability to learn beyond iid data, purposefully curated ood datasets can be used.

For network security problems, ood datasets of interest can represent different real-world network conditions (e.g., different user populations, protocols, applications, network technologies, architectures, or topologies) or different network situations (also referred to as distribution shift [83] or concept drift [63]). For determining whether or not a trained model generalizes to different scenarios, it is important to select ood datasets that accurately reflect the different conditions that can prevail in those scenarios.

2.3 Existing Approaches

We can divide the existing approaches to improving a model's generalizability into two broad categories: (1) Efforts for improving model selection, training, and testing algorithms; and (2) Efforts for improving the training datasets. The first category focuses mainly on the later steps in the standard ML pipeline (see Figure 1) that deal with the model's structure, the algorithm used for training, and the evaluation process. The second category is concerned with improving the quality of datasets used during model training and focuses on the early steps in the standard ML pipeline.

Improving model selection, training, and evaluation. The focal point of most existing efforts is either the model's structure (e.g., domain adaption [40, 92] and multi-task learning [88, 108]), or the training algorithm (e.g., few-shot learning [46, 87]), or the evaluation process (e.g., ood detection [58, 106]). However, they neglect the training dataset, mainly because it is in general assumed to be fixed and already given. While these efforts provide insights into improving model generalizability, studying the problem without the ability to actively and flexibly change the training dataset is difficult, especially when the given training dataset turns out to exhibit inductive biases, be noisy or of low quality, or simply be non-informative for the problem at hand [51]. See Section 8 for a more detailed discussion about existing model-based efforts and how they differ from our proposed approach described below.

Improving the training dataset. Data augmentation is a passive method for synthesizing new or modifying existing training datasets and is widely used in the ML community to improve models' generalizability. Technically, data augmentation methods leverage different operations (e.g., adding random noise [100], using linear interpolations [107] or more complex techniques) to synthesize new training samples for different types of data such as images [95, 100], text [107], or tabular data [24, 59]. However, using such passive data-generation methods for the network security domain is inappropriate or counterproductive because they often result in unrealistic or even semantically meaningless datasets [43]. For example, since network protocols usually adhere to agreed-upon standards, they constrain various network data in ways that such data-generation methods cannot ensure without specifically incorporating domain knowledge. Furthermore, various network environments can induce significant differences in observed communication patterns, even when using the same tools or considering the same scenarios [38], by influencing data characteristics (e.g., packet interarrival times, packet sizes, or header information) and introducing unique network conditions or patterns.

2.4 Limitations of Existing Approaches

From a network security domain perspective, these existing approaches miss out on two aspects that are intimately related to improving a model's ability to generalize: (1) Leveraging insights from model explainability tools, and (2) ensuring the realism of collected training datasets.

Using explainable ML techniques. To better scrutinize an ML model's weaknesses and understand model errors, we argue that an additional explainability step that relies on recent advances in explainable ML should be added to the standard ML pipeline to improve the ML workflow for network security problems [50, 56, 81, 94]. The idea behind adding such a step is that it enables taking the output of the standard ML pipeline, extracting and examining a carefully-constructed white-box model in the form of a decision tree, and then scrutinizing it for signs of blind spots in the output of the standard ML pipeline. If such blind spots are found, the decision tree and an associated summary report can be consulted to trace their root causes to aspects of the training dataset and/or model specification that led the output to encode inductive biases.

Ensuring realism in collected training datasets. To beneficially study model generalizability from the training dataset perspective, we posit that for the network security domain, the collection of training datasets should be done *endogenously* or *in vivo*; that is, performed or taking place within the network environment of interest. Given that network-related datasets are typically the result of intricate interactions between different protocols and their various embedded closed control loops, accurately reflecting these complexities associated with particular deployment settings or traffic conditions requires collecting the datasets from within the network.

2.5 Our Approach in a Nutshell

We take a first step towards a more systematic treatment of the model generalizability problem and propose an approach that (1) uses a new closed-loop ML pipeline and (2) calls for running this pipeline in its entirety multiple times, each time with a possibly different model specification but always with *a different training dataset* compared to the original one. Here, we use a newly-proposed closed-loop ML pipeline (Figure 1) that differs from the standard pipeline by including an explanation step. Also, each new training dataset used as part of a new run of the closed-loop ML pipeline is assumed to be endogenously collected and not exogenously manipulated.

The collection of each new training dataset is informed by a root cause analysis of identified inductive bias(es) in the trained model. This analysis leverages existing explainability tools that researchers have at their disposal as part of the closed-loop pipeline's explainability step. In effect, such an informed data-collection effort promises to enhance the quality of the given training datasets by gradually reducing the presence of inductive biases that are identified by our approach, thus resulting in trained models that are more likely to generalize. Note, however, that our proposed approach *does not guarantee* model generalizability. Instead, by eliminating identified inductive biases in the form of shortcuts and ood data, our approach enhances a model's generalizability capabilities. Also, note that our focus in this paper is not on designing novel model explainability methods but rather on applying available techniques from the existing literature. In fact, while we are agnostic about

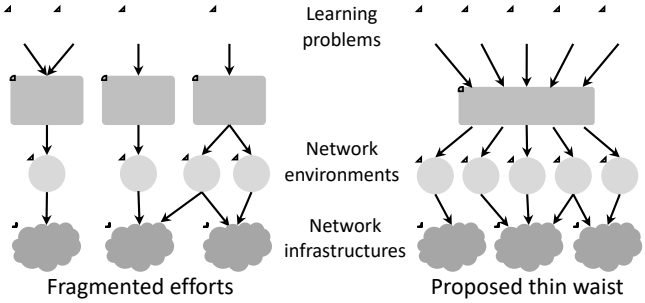


Figure 2: netUnicorn vs. existing data collection efforts.

which explainability tools to use for this step, we recommend the application of global explainability tools such as Trustee [56] over local explainability techniques (e.g., [50, 65, 85, 101, 103]), mainly because the former are in general more powerful and informative with respect to faithfully detecting and identifying root causes of inductive biases compared to the latter. However, as shown in Section 5 below, either of these two types of methods can shed light on the nature of a trained model's inductive biases.

Our proposed approach differs from existing approaches in several ways. First, it reduces the burden on the user or domain expert to select the "right" training dataset apriori. Second, it calls for the collection of training datasets that are endogenously generated and where explainability tools guide the decision-making about what "better" data to collect. Third, it proposes using multiple training datasets, collected iteratively (in a fail-fast manner), to combat the underspecification of the trained models and thus enhance model generalizability. In particular, it recognizes that an "ideal" training dataset may not be readily available in the beginning and argues strongly against attaining it through exogenous means.

3 ON "IN VIVO" DATA-COLLECTION

In this section, we discuss some of the main issues with existing data-collection efforts and describe our proposed approach to overcome their shortcomings.

3.1 Existing Approaches

Data collection operations. We refer to collecting data for a learning problem from a specific network environment (or domain) as a data-collection *experiment*. We divide such a data-collection experiment into three distinct operations. (1) *Specification:* expressing the intents that specify what data to collect or generate for the experiment. (2) *Deployment:* bootstrapping the experiment by translating the high-level intents into target-specific commands and configurations across the physical or virtual data-collection infrastructure and implementing them. (3) *Execution:* orchestrating the experiment to collect the specified data while handling different runtime events (e.g., node failure, connectivity issues, etc.). Here, the first operation is concerned with "what to collect," and the latter operations deal with "how to collect" this data.

The "fragmentation" issue. Existing data-collection efforts are inherently *fragmented*, i.e., they only work for a specific learning problem and network environment, emulated using one or more network infrastructures (Figure 2). Extending them to collect data for a new learning problem or from a new network environment is challenging. For example, consider the data-collection effort for the video fingerprinting problem [90], where the goal is to fingerprint different videos for video streaming applications (e.g., YouTube) using a stream of encrypted network packets as input. Here, the data-collection intent is to start a video streaming session and collect the related packet traces from multiple end hosts that comprise a specific target environment. The deployment operation entails developing scripts that automate setting up the computing environment (e.g., installing the required selenium package) at the different end hosts. The execution operation requires developing a runtime system to start/stop the experiments and handle runtime events such as node failure, connectivity issues, etc.

Lack of modularity. In addition to being one-off in nature, existing approaches to collecting data for a given learning problem are also monolithic. That is, being highly problem-specific, there is, in general, no clear separation between experiment specification and mechanisms. An experimenter must write scripts that realize the data-collection intents (e.g., start/stop video streaming sessions, collect pcaps, etc.), deploy these scripts to one or more network infrastructures, and execute them to collect the required data. Given this monolithic structure, existing data collection approaches [90] cannot easily be extended so that they can be used for a different learning problem, such as inferring QoE [17, 48, 52] or for a different network environment, such as congested environments (e.g., hotspots in a campus network) or high-latency networks (e.g., networks that use GEO satellites as access link).

Disparity between virtual and physical infrastructures. While a number of different network emulators and simulators are currently available to researchers [62, 72, 77, 105], it is, in general, difficult or impossible to write experiments that can be seamlessly transferred from a virtual to a physical infrastructure and back. This capability is particularly appealing in view of the fact that virtual infrastructures provide the ability to quickly iterate on data collection and test various network conditions, including conditions that are complex in nature and, in general, difficult to achieve in physical infrastructures. Due to the lack of this capability, experimenters often end up writing experiments for only one of these infrastructures, creating different (typically simplified) experiment versions for physical test beds, or completely rewriting the experiments to account for real-world conditions and problems (e.g., node and link failures, network synchronization)

Missed opportunity. Together, these observations highlight a missed opportunity for researchers who now have access to different network infrastructures. The list includes NSF-supported research infrastructures, such as EdgeNet [39], ChiEdge [22], Fabric [8], PAWR [80], etc., as well as on-demand infrastructure offered by different cloud services providers, such as AWS [18], Azure [19], Digital Ocean [20], GCP [21], etc. This rich set of network infrastructures can aid in emulating diverse and representative network environments for data collection.

3.2 An "Hourglass" Design to the Rescue

The observed fragmented, one-off, and monolithic nature of how training datasets for network security-related ML problems are currently collected motivates a new and more principled approach that aims at lowering the threshold for researchers wanting to collect high-quality network data. Here, we say a training dataset is of

high quality if the model trained using this dataset is not obviously prone to inductive biases and, therefore, likely to generalize.

Our hourglass model. Our proposed approach takes inspiration from the classic "hourglass" model [11], a layered systems architecture that, in our case, consists of designing and implementing a "thin waist" that enables collecting data for different learning problems (hourglass' top layer) from a diverse set of possible network environments (hourglass' bottom layer). In effect, we want to design the thin waist of our hourglass model in such a way that it accomplishes three goals: (1) allows us to collect a specified training dataset for a given learning problem from network environments emulated using one or more supported network infrastructures, (2) ensures that we can collect a specified training set for each of the considered learning problems for a given network environment, and (3) facilitates experiment reproducibility and shareability.

Requirements for a "thin waist". Realizing this hourglass model's thin waste requires developing a flexible and modular data-collection platform that supports two main functionalities: (1) *decoupling* data-collection intents (i.e., expressing **what** to collect and from **where**) from mechanisms (i.e., **how** to realize these intents); and (2) *disaggregating* intents into independent and reusable tasks.

The required first functionality allows the experimenter to focus on the experiment's intent without worrying about how to implement it. As a result, expressing a data-collection experiment does not require re-doing tasks related to deployment and execution in different network environments. For instance, to ensure that the learning model for video fingerprinting is not overfitted to a specific network environment, collecting data from different environments, such as congested campus networks or cable- and satellite-based home networks, is important. Not requiring the experimenter to specify the implementation details simplifies this process.

Providing support for the second functionality allows the experimenter to reuse common data-collection intents and mechanisms for different learning problems. For instance, while the goal for QoE inference and video fingerprinting may differ, both require starting and stopping video streaming sessions on an end host.

Ensuring these two required functionalities makes it easier for an experimenter to iteratively improve the data collection intent, addressing apparent or suspected inductive biases that a model may have encoded and may affect the model's ability to generalize.

4 REALIZING THE "THIN WAIST" IDEA

To achieve the desired "thin waist" of the proposed hourglass model, we develop a new data-collection platform, netUnicorn. We identify two distinct stakeholders for this platform: (1) *experimenters* who express data-collection intents, and (2) *developers* who develop different modules to realize these intents. In Section 4.1, we describe the programming abstractions that netUnicorn considers to satisfy the "thin" waist requirements, and in Section 4.2, we show how netUnicorn realizes these abstractions while ensuring fidelity, scalability, and extensibility.

4.1 Programming Abstractions

To satisfy the second requirement (*disaggregation*), netUnicorn allows experimenters to disaggregate their intents into distinct pipelines and tasks. Specifically, netUnicorn offers experimenters `Task` and `Pipeline` abstractions. Experimenters can structure data collection experiments by utilizing multiple independent pipelines. Each pipeline can be divided into several processing stages, where each stage conducts self-contained and reusable tasks. In each stage, the experimenter can specify one or more tasks that netUnicorn will execute concurrently. Tasks in the next stage will only be executed once all tasks in the previous stage have been completed.

To satisfy the first requirement, netUnicorn offers a unified interface for all tasks. To this end, it relies on abstractions that concern specifics of the computing environment (e.g., containers, shell access, etc.) and executing target (e.g., ARM-based Raspberry Pis, AMD64-based computers, OpenWRT routers, etc.) and allows for flexible and universal task implementation.

To further decouple intents from mechanisms, netUnicorn's API exposes the `Nodes` object to the experimenters. This object abstracts the underlying physical or virtual infrastructure as a pool of data-collection nodes. Here, each node can have different static and dynamic attributes, such as type (e.g., Linux host, PISA switch), location (e.g., room, building), resources (e.g., memory, storage, CPU), etc. An experimenter can use the `filter` operator to select a subset of nodes based on their attributes for data collection. Each node can support one or more compute environments, where each environment can be a shell (command-line interpreter), a Linux container (e.g., Docker [34]), a virtual machine, etc. netUnicorn allows users to map pipelines to these nodes using the `Experiment` object and `map` operator. Then, experimenters can deploy and execute their experiments using the `Client` object. For details about the key components of netUnicorn's API, see [13], Table 7.

Illustrative example. To illustrate with an example how an experimenter can use netUnicorn's API to express the data-collection experiment for a learning problem, we consider the bruteforce attack detection problem. For this problem, we need to realize three pipelines, where the different pipelines perform the key tasks of running an HTTPS server, sending attacks to the server, and sending benign traffic to the server, respectively. The first pipeline also needs to collect packet traces from the HTTPS server.

Listing 1 shows how we express this experiment using netUnicorn. Lines 1-6 show how we select a host to represent a target server, start the HTTPS server, perform PCAP capture, and notify all other hosts that the server is ready. Lines 8-16 show how we can take hosts from different environments that will wait for the target server to be ready and then launch a bruteforce attack on this node. Lines 18-26 show how we select hosts that represent benign users of the HTTPS server. Finally, lines 28-32 show how we combine pipelines and hosts into a single experiment, deploy it to all participating infrastructure nodes, and start execution.

Note that in Listing 1 we omitted task definitions and instantiation, package imports, client authorization, and other details to simplify the exposition of the system.

4.2 System Design

The netUnicorn compiles high-level intents, expressed using the proposed programming abstraction, into target-specific programs. It then deploys and executes these programs on different data-collection nodes to complete an experiment. netUnicorn is designed to realize the high-level intents with *fidelity*, minimize the inherent

```
# Target server
h1 = Nodes.filter('location', 'azure').take(1)
p1 = Pipeline()
     .then(start_http_server)
     .then(start_pcap)
     .then(set_readiness_flag)

# Malicious hosts
h2 = [
  Nodes.filter('location', 'campus').take(40),
  Nodes.filter('location', 'aws').take(40),
  Nodes.filter('location', 'digitalocean').take(40),
]
p2 = Pipeline()
     .then(wait_for_readiness_flag)
     .then(patator_attack)

# Benign hosts
h3 = [
  Nodes.filter('location', 'campus').take(40),
  Nodes.filter('location', 'aws').take(40),
  Nodes.filter('location', 'digitalocean').take(40),
]
p3 = Pipeline()
     .then(wait_for_readiness_flag)
     .then(benign_traffic)

e = Experiment()
    .map(p1, h1)
    .map(p2, h2)
    .map(p3, h3)
Client().deploy(e).execute(e)
```

Listing 1: Data collection experiment example for the HTTPS bruteforce attack detection problem. We have omitted task instantiations and imports to simplify the exposition.

computing and communication overheads (*scalability*), and simplify supporting new data-collection tasks and infrastructures for developers (*extensibility*).

Ensuring high fidelity. netUnicorn is responsible for compiling a high-level experiment into a sequence of target-specific programs. We divide these programs into two broad categories for each task: deployment and execution. The deployment definitions help configure the computing environment to enable the successful execution of a task. For example, executing the `YouTubeWatcher` task requires installing a Chromium browser and related extensions. Since successful execution of each specified task is critical for satisfying the fidelity requirement, netUnicorn must ensure that the computing environment at the nodes is set up for a task before execution.

Addressing the scalability issues. To execute a given pipeline, a system can control deployment and execution either at the task- or pipeline-level granularity. The first option entails the deployment and execution of the *task* and then reporting results back to the system before executing the next *task*. It ensures fidelity at the task granularity and allows the execution of pipelines even with tasks with contradicting requirements (e.g., different library versions). However, since such an approach requires communication with core system services, it slows the completion time and incurs additional computing and network communication overheads.

Our system implements the second option, running all the setup programs before marking a *pipeline* ready for execution and then offloading the task flow control to a node-based executor that reports results only at the end of the pipeline. It allows for optimization of environment preparation (e.g., configure a single docker image for distribution) and time overhead between tasks, and also reduces

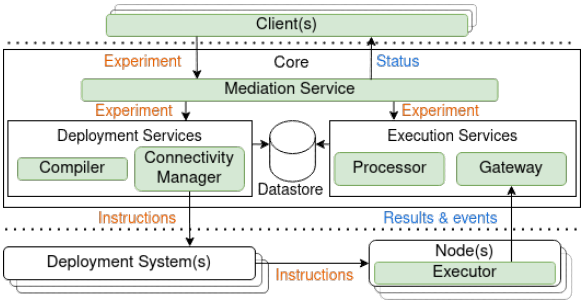


Figure 3: Architecture of the proposed system. Green-shaded boxes show all the implemented services.

network communication while offering only "best-effort" fidelity for pipelines.

Enabling extensibility. Enabling extensibility calls for simplifying how a developer can add a new task, update an existing task for a new target, or add a new physical or virtual infrastructure. Note that the netUnicorn's extensibility requirement targets developers and not experimenters.

Simplify adding and updating tasks. An experimenter specifies a task to be executed in a pipeline. The netUnicorn chooses a specific implementation of this task. This may require customizing the computing environment, which can vary depending on the target (e.g., container vs shell of OpenWRT router). For example, a Chromium browser and specific software must be installed to start a video streaming session on a remote host without a display. The commands to do so may differ for different targets. The system provides a base class that includes all necessary methods for a task.

Developers can extend this base class by providing their custom subclasses with the target-specific `run` method to specify how to execute the task for different types of targets. This allows for easy extensibility because creating a new task subclass is all that is needed to adapt the task to a new computing environment.

Simplify adding new infrastructures. To deploy new data-collection pipelines, send commands, and send/receive different events and data to/from multiple nodes in the underlying infrastructure, netUnicorn requires an underlying deployment system.

One option is to bind netUnicorn to one of the existing deployment (orchestration) systems, such as Kubernetes [60], SaltStack [89], Ansible [4], or others for all infrastructures. However, requiring a physical infrastructure to support a specific deployment system is disruptive in practice. Network operators managing a physical infrastructure are often not amenable to changing their deployment system as it would affect other supported services.

Another option is to support multiple deployment systems. However, we need to ensure that supporting a new deployment system does not require a major refactoring of netUnicorn's existing modules. To this end, netUnicorn introduces a separate connectivity module that abstracts away all the connectivity issues from the netUnicorn's other modules (e.g., runtime), offering seamless connectivity to infrastructures using multiple deployment systems. Each time developers want to add a new infrastructure that uses an unsupported deployment system, they only need to update the connectivity manager — simplifying extensibility.

4.3 Prototype Implementation

Our implementation of netUnicorn is shown in Figure 3. Our implementation embraces a service-oriented architecture [86] and has three key components: *client(s)*, *core*, and *executor(s)*. Experimenters use local instances of netUnicorn's *client* to express their data-collection experiments. Then, netUnicorn's *core* is responsible for all the operations related to the compilation, deployment, and execution of an experiment. For each experiment, netUnicorn's core deploys a target-specific *executor* on all related data-collection nodes for running and reporting the status of all the programs provided by netUnicorn's core.

The netUnicorn's core offer three main service groups: mediation, deployment, and execution services. Upon receiving an experiment specification from the client, the `mediation service` requests the `compiler` to extract the set of setup configurations for each distinct (pipeline, node-type) pair, which it uploads to the local PostgreSQL database. After compilation, the `mediation service` requests the connectivity manager to ship this configuration to the appropriate data-collection nodes and verify the computing environment. In the case of docker-based infrastructures, this step is performed locally, and the configured docker image is uploaded to a local docker repository. The `connectivity-manager` uses an infrastructure-specific deployment system (e.g., SaltStack [89]) to communicate with the data-collection nodes.

After deploying all the required instructions, the `mediation service` requests the connectivity manager to instantiate a target-specific `executor` for all data-collection nodes. The `executor` uses the instructions shipped in the previous stage to execute a data-collection pipeline. It reports the status and results to netUnicorn's `gateway` and then adds them to the related table in the SQL database via the `processor`. The `mediation service` retrieves the status information from the database to provide status updates to the experimenter(s). Finally, at the end of an experiment, the `mediation service` sends cleanup scripts (via `connectivity-manager`) to each node—ensuring the reusability of the data-collection infrastructure across different experiments.

5 EVALUATION: CLOSED-LOOP ML PIPELINE

In this section, we demonstrate how our proposed closed-loop ML pipeline helps to improve model generalizability. Specifically, we seek to answer the following questions: ❶ Does the proposed pipeline help in identifying and removing shortcuts? ❷ How do models trained using the proposed pipeline perform compare to models trained with existing exogenous data augmentation methods? ❸ Does the proposed pipeline help with combating ood issues?

5.1 Experimental Setup

To illustrate our approach and answer these questions, we consider the bruteforce example mentioned in Section 4.1 and first describe the different choices we made with respect to the ML pipeline and the iterative data-collection methodology.

Network environments. We consider three distinct network environments for data collection: a `UCSB` network, a hybrid `UCSB-cloud` setting, and a `multi-cloud` environment.

The `UCSB` network environment is emulated using a programmable data-collection infrastructure PINOT [12]. This infrastructure is deployed at a campus network and consists of multiple (40+) single-board computers (such as Raspberry Pis) connected to the Internet via wired and/or wireless access links. These computers are strategically located in different areas across the campus, including the library, dormitories, and cafeteria. In this setup, all three types of nodes (i.e., target server, benign hosts, and malicious hosts) are selected from end hosts on the campus network. The `UCSB-cloud` environment is a hybrid network that combines programmable end hosts at the campus network with one of three cloud service providers: AWS, Azure, or Digital Ocean.[2] In this setup, we deploy the target server in the cloud while running the benign and malicious hosts on the campus network. Lastly, the `multi-cloud` environment is emulated using all three cloud service providers with multiple regions. We deploy the target server on Azure and the benign and malicious hosts on all three cloud service providers.

Data collection experiment. The data-collection experiment involves three pipelines, namely target, benign, and malicious. Each of these pipelines is assigned to different sets of nodes depending on the considered network environment. The target pipeline is responsible for deploying a public HTTPS endpoint with a real-world API that requires authentication for access. Additionally, this pipeline utilizes *tcpdump* to capture all incoming and outgoing network traffic. The benign pipeline emulates valid usage of the API with correct credentials, while the malicious pipeline attempts to obtain the service's data by brute-forcing the API using the Patator [79] tool and a predefined list of commonly used credentials [91].

Data pre-processing and feature engineering. We used CICFlowMeter [29] to transform raw packets into a feature vector of 84 dimensions for each unique connection (flow). These features represent flow-level summary statistics (e.g., average packet length, inter-arrival time, etc.) and are widely used in the network security community [30, 36, 93, 109].

Learning models. We train four different learning models. Two of them are traditional ML models, i.e., Gradient Boosting (GB) [71], Random Forest (RF) [16]. The other two are deep learning-based methods: Multi-layer Perceptron (MLP) [46], and attention-based TabNet model (TN) [6]. These models are commonly used for handling tabular data such as CICFlowMeter features [49, 96].

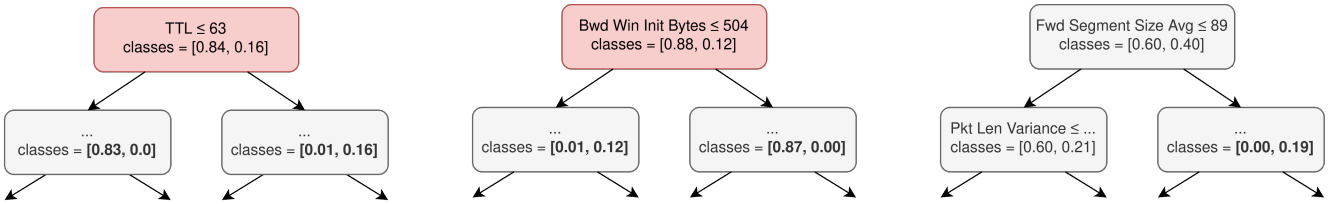
Explainability tools. To examine a model trained with a given training dataset for the possible presence of inductive biases such as shortcuts or ood issues, our newly proposed ML pipeline requires an explainability step that consists of applying existing model explainability techniques, be they global or local in nature, but what technique to use is left to the discretion of the user.

We illustrate this step by first applying a global explainability method. In particular, our method-of-choice is the recently developed tool Trustee [56], but other global model explainability techniques could be used as well, including PDP plots [41], ALE plots [5], and others [70, 76]. Our reasoning for using the Trustee tool is that for any trained black-box model, it extracts a high-fidelity and low-complexity decision tree that provides a detailed explanation of the trained model's decision-making process. Together with a summary report that the tool provides, this decision tree is an ideal means for scrutinizing the given trained model for possible problems such as shortcuts or ood issues.

[2] Unless specified otherwise, we host the target server on Azure for this environment.

Table 1: Number of LLoC changes, data points, and F1 scores across different environments and iterations.

	Iteration #0 (initial setup)		Iteration 1		Iteration 2	
LLoCs	**80**		**+10**		**+20**	
	UCSB-0 (train)	multi-cloud (test)	UCSB-1 (train)	multi-cloud (test)	UCSB-2 (train)	multi-cloud (test)
MLP	1.0	0.56	0.97 (-0.03)	0.62 (+0.06)	0.88 (-0.09)	**0.94 (+0.38)**
GB	1.0	0.61	1.0 (+0.00)	0.61 (+0.00)	0.92 (-0.08)	**0.92 (+0.31)**
RF	1.0	0.58	1.0 (+0.00)	0.69 (+0.11)	0.97 (-0.03)	**0.93 (+0.35)**
TN	1.0	0.66	0.97 (-0.03)	0.78 (+0.12)	0.92 (-0.05)	**0.95 (+0.29)**



(a) Iteration #0: top branch is a shortcut. **(b) Iteration #1: top branch is a shortcut.** **(c) Iteration #2: no obvious shortcut.**

Figure 4: Decision trees generated using Trustee [56] across the three iterations. We highlight the nodes that are indicators for shortcuts in the trained model.

To compare, we also apply local explainability tools to perform the explainability step. More specifically, we consider the two well-known techniques, LIME [85] and SHAP [65]. These methods are designed to explain a model's decision for individual input samples and thus require analyzing the explanations of multiple inputs to make conclusions about the presence or absence of model blind spots such as shortcuts or ood issues. While users are free to replace LIME or SHAP with more recently developed tools such as xNIDS [103] or their own preferred methods, they have to be mindful of the efforts each method requires to draw sound conclusions about certain non-local properties of a given trained model (e.g., shortcut learning).

5.2 Identifying and Removing Shortcuts

To answer ❶, we consider a setup where a researcher curates training datasets from the UCSB environment and aims at developing a model that generalizes to the multi-cloud environment (i.e., unseen domain).

Initial setup (iteration #0). We denote the training data generated from this experiment as UCSB-0. Table 1 shows that while all three models have a perfect training performance, they all have low testing performance (errors are mainly false positives). We first used our global explanation method-of-choice, Trustee, to extract the decision tree of the trained models. As shown in Figure 4, the top node is labeled with the separation rule ($TTL \leq 63$) and the balance between the benign and malicious samples in the data ("classes"). Subsequent nodes only show the class balance after the split.

From Figure 4a, we conclude that all four models use almost exclusively the TTL (time-to-live) feature to discriminate between benign and malicious flows, which is an obvious shortcut. Note that the top parts of Trustee-extracted decision trees were identical for all four models. When applying the local explanation tools LIME and SHAP to explain 100 randomly selected input samples, we found that these explanations identified TTL as the most important feature in all 100 samples. While consistent with our Trustee-derived conclusion, these LIME- or SHAP-based observations are necessary but not sufficient to conclusively decide whether or not the trained models learned a TTL-based shortcut strategy and further efforts would be required to make that decision.

To understand the root cause of this shortcut, we checked the UCSB infrastructure and noticed that almost all nodes used for benign traffic generation have the exact same TTL value due to a flat structure of the UCSB network. This observation also explains why most errors are false positives, i.e., the model treats a flow as malicious if it has a different TTL from the benign flows in the training set. Existing domain knowledge suggests that this behavior is unlikely to materialize in more realistic settings such as the multi-cloud environment. Consequently, we observe that models trained using the UCSB-0 dataset perform poorly on the unseen domain; i.e., they generalize poorly.

Removing shortcuts (iteration #1). To fix this issue, we modified the data-collection experiment to use a more diverse mix of nodes for generating benign and malicious traffic and collected a new dataset, UCSB-1. However, this change only marginally improved the testing performance for all three models (Table 1). Inspection of the corresponding decision trees shows that all the models use the "Bwd Init Win Bytes" feature for discrimination, which appears to be yet another shortcut. Again, we observed that all trees generated by Trustee from different black-box models have identical top nodes. Similar, our local explanation results obtained by LIME and SHAP also point to this feature as being the most important one across the analyzed samples.

More precisely, this feature quantifies the TCP window size for the first packet in the backward direction, i.e., from the attacked server to the client. It acts as a flow control and reacts to whether the receiver (i.e., HTTP endpoint) is overloaded with incoming data. Although it could be one indicator of whether the endpoint is being brute-force attacked, it should only be weakly correlated with whether a flow is malicious or benign. Given this reasoning

Table 2: F1 score of models trained using our approach (i.e., leveraging netUnicorn) vs. models trained with datasets collected from the UCSB network by exogenous methods (i.e., without using netUnicorn).

	Iteration #0				Iteration #1				**Iteration #2**			
	MLP	GB	RF	TN	MLP	GB	RF	TN	MLP	GB	RF	TN
Naive Aug.	0.51	0.57	0.56	0.53	0.73	0.67	0.71	0.82	-	-	-	-
Noise Aug.	0.66	0.68	0.67	0.66	0.72	0.83	0.76	0.82	-	-	-	-
Feature Drop	0.74	0.55	0.72	**0.87**	**0.91**	0.58	0.63	**0.89**	-	-	-	-
SYMPROD	0.66	0.71	0.67	0.41	0.69	0.66	0.75	0.67	**0.94**	**0.93**	**0.95**	**0.96**
Our approach									**0.94**	**0.92**	**0.95**	**0.95**

and the poor generalizability of the models, we consider the use of this feature to be a shortcut.

Removing shortcuts (iteration #2). To remove this newly identified shortcut, we refined the data-collection experiment. First, we created a new task that changes the workflow for the Patator tool. This new version uses a separate TCP connection for each bruteforce attempt and has the effect of slowing down the bruteforce process. Second, we increased the number of flows for benign traffic and the diversity of benign tasks. Using these changes, we collected a new dataset, UCSB-2.

Table 1 shows that the change in data-collection policy significantly improved the testing performance for all models. We no longer observe any obvious shortcuts in the corresponding decision tree. Moreover, domain knowledge suggests that the top three features (i.e., "Fwd Segment Size Average", "Packet Length Variance", and "Fwd Packet Length Std") are meaningful and their use can be expected to accurately differentiate benign traffic from repetitive brute force requests. Applying the local explanation methods LIME and SHAP also did not provide any indications of obvious additional shortcuts. Note that although the models appear to be shortcut-free, we cannot guarantee that the models trained with these diligently curated datasets do not suffer from other possible encoded inductive biases. Further improvements of these curated datasets might be possible but will require more careful scrutiny of the obtained decision trees and possibly more iterations.

5.3 Comparison with Exogeneous Methods

To answer ❷, we compare the performance of the model trained using UCSB-2 (i.e., the dataset curated after two rounds of iterations) with that of models trained with datasets modified by means of existing exogenous methods. Specifically, we consider the following methods:

(1) **Naive augmentation.** We use a naive data collection strategy that does not apply the extra explanation step that our newly proposed ML pipeline includes to identify training data-related issues. The strategy simply collects more data using the initial data-collection policy. It is an ablation study demonstrating the benefits of including the explanation step in our new pipeline. Here, for each successive iteration, we double the size of the training dataset.

(2) **Noise augmentation.** This popular data augmentation technique consists of adding suitable chosen random uniform noise [66] to the identified skewed features in each iteration. Here, for iteration #0, we use integer-valued uniformly-distributed random samples from the interval $[-1; +1]$ for TTL noise augmentation, and for iteration #1, we similarly use integer-valued uniformly-distributed samples from the interval $[-5; +5]$ for noise augmentation of the feature "Bwd Init Win Bytes".

(3) **Feature drop.** This method simply drops a specified skewed feature from the dataset in each iteration. In our case, we drop the identified skewed feature for all training samples in each training dataset.

(4) **SYMPROD.** SMOTE [24] is a popular augmentation method for tabular data that applies interpolation techniques to synthesize data points to balance the data across different classes. Here we utilize a recently considered version of this method called SYMPROD [61] and augment each training set by adding the number of rows necessary for restoring class balance ($proportion = 1$).

We apply these methods to the three training datasets curated from the campus network in the previous experiment. For UCSB-0 and UCSB-1, we use the two identified skewed features for adding noise or dropping features altogether.

Note that since we did not identify any skewed features in the last iteration, we did not apply any noise augmentation and feature drop techniques in this iteration and did not collect more data for the naive data augmentation method.

As shown in Table 2, the models trained using these exogenous methods perform poorly in all iterations when compared to our approach. This highlights the main benefit we gain from applying our proposed closed-loop ML pipeline for iterative data collection and model training. In particular, it demonstrates that the explanation step in our proposed pipeline adds value. While doing nothing (i.e., naive data augmentation) is clearly not a worthwhile strategy, applying either noise augmentation or SYMPROD can potentially compromise the semantic integrity of the training data, making them ill-suited for addressing model generalizability issues for network security problems.

Table 3: The testing F1 score of the models before and after retraining with malicious traffic generated by Hydra.

	MLP	GB	RF	TN	Avg
Before retraining	0.87	0.81	0.86	0.83	0.84
After retraining	**0.93**	**0.96**	**0.91**	**0.91**	**0.93**

Table 4: The F1 score of models trained using only UCSB data or data from UCSB and UCSB-cloud infrastructures.

	UCSB		**UCSB-cloud**	
	Training	Test	Training	Test
MLP	0.88	0.94	**0.95 (+0.07)**	**0.95 (+0.01)**
GB	0.92	0.92	**0.96 (+0.04)**	**0.95 (+0.03)**
RF	0.97	0.93	**0.96 (-0.01)**	**0.97 (+0.04)**
TN	0.83	0.95	**0.84 (+0.01)**	**0.96 (+0.01)**

5.4 Combating ood-specific Issues

To answer ❸, we consider two different scenarios: *attack adaptation* and *environment adaptation*.

Attack adaptation. We consider a setup where an attacker changes the tool used for the bruteforce attack, i.e., uses Hydra [55] instead

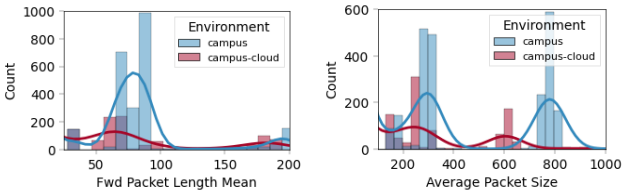


Figure 5: Distributions of several features across two different environments: UCSB and UCSB-cloud

of Patator. To this end, we use netUnicorn to generate a new testing dataset from the UCSB infrastructure with Hydra as the bruteforce attack. Table 3 shows that the model's testing performance drops significantly (to 0.85 on average). We observe that this drop is because of the model's reduced ability to identify malicious flows, which indicates that changing the attack generation tool introduces oods, although they belong to the same attack type.

To address this problem, we modified the data generation experiment to collect attack traffic from both Hydra and Patator in equal proportions. This change in the data-collection experiment only required **6** LLoC. We retrain the models on this dataset and observe significant improvements in the model's performance on the same test dataset after retraining (see Table 3).

Note that we only test one type of oods where the evolved attack still has the same goal and functionality. However, an attack can also evolve into another attack with a different goal, resulting in ood samples with new labels. Here, we leverage ensemble models and human analysis to identify the ood case. While it may be possible to identify ood issues using more automated methods that are motivated by findings obtained from applying global explainability tools, we plan to revisit this problem in our future work.

Environment adaptation. We consider testing the model we developed in the UCSB environment in the unseen `multi-cloud` environment as a different instance of an ood issue that is due to possible feature distribution differences. To address this issue, we use the `UCSB-cloud` environment for data collection. As expected, we observe differences in the distributions for some of the features across the two environments (see Figure 5). Table 4 shows the performance of the models trained using only the data from the UCSB environment compared to the ones that use data from both the UCSB and `UCSB-cloud` environments. Notably, as `UCSB-cloud` is more similar to the `multi-cloud` environment than the UCSB environment, the models trained with the `UCSB-cloud` data show improvements in their performance under the test settings.

6 EVALUATION: NETUNICORN

We answer if netUnicorn lowers the threshold for data collection for: ❹ different learning problems for a given network environment? ❺ a given learning problem from different environments, emulated using one or more network infrastructures? and ❻ iteratively calibrating the data collection intents for a given learning problem and environment? We also demonstrate ❼ how well does netUnicorn scale for larger data-collection infrastructures, especially the ones equipped with relatively low-end devices, such as RPis?

6.1 Experimental Setup

Learning problems. Besides the *HTTP bruteforce attack detection* problem, we explore two more learning problems for this experiment, namely *video fingerprinting* and *advanced persistent threats detection* (APTs). In the case of the first additional example, the learning problem is to fingerprint videos for web-based streaming services, such as YouTube, that adopt variable bitrates [90]. Previous work [90] did not evaluate the proposed learning model under realistic network conditions. Thus, to collect meaningful data for this problem, we use a network of end hosts in the UCSB infrastructure to collect a training dataset for five different YouTube videos.[3] Specifically, our data-collection intent is specified by the following sequence of tasks: start packet capture, watch a YouTube video in headless mode for 30 seconds, and stop packet capture. We repeat this sequence ten times for each video in a shuffled order and combine it into a single pipeline, where at the end, we upload the collected data to our server.

Regarding the second additional example, the learning problem, in this case, is to identify the hosts that some APTs have compromised. To generate data for this learning problem, we write an experiment that mimics the behavior of a compromised host. Specifically, our data-collection intent is as follows: find active hosts using `Ping`, check if port 443 is opened for active hosts (identified in the previous stage) with `PortScan`, and then for each host with open 443 port launch four different attacks in parallel: `CVE20140160` (Heartbleed), `CVE202141773` (Apache 2.4.49 Path), `CVE202144228` (Log4J), and `Patator` (HTTP admin endpoint bruteforce using the Patator tool). The ML pipeline creates a "semi-realistic" training dataset by combining actively generated attack traffic with passively collected packet traces from a border router of a production network, such as the UCSB network.[4] We then use this dataset for model training. Note, here we assume that we know the attacker's playbook; that is, the goal, in this case, is not to demonstrate a realistic attack playbook but to demonstrate that netUnicorn simplifies generating attack traffic for a given APT attack playbook.

Network environments. netUnicorn enables emulating network environments for data collection using one or more physical/virtual infrastructures. Previously, we used a SaltStack-based infrastructure at UCSB and multiple clouds to emulate various network environments: UCSB, `UCSB-cloud`, and `multi-cloud`. In this experiment, we implement a connector to another infrastructure, Azure Container Instances (ACI) to expand cloud-based environments with serverless Docker containers. During the experiments, containers were dynamically created in multiple regions and used for pipeline execution. Overall, netUnicorn currently supports six different deployment system connectors (see [13], Table 8, for more details).

Baseline. To the best of our knowledge, none of the existing platforms/systems offer the desired extensibility, scalability, and fidelity for data collection (see Section 8 for more details). To illustrate how netUnicorn simplifies data collection efforts, we consider baselines that directly configure three different deployment/orchestration

[3] Each video is identified with a unique URL.

[4] Note, in theory, we could use netUnicorn to actively collect the benign traffic for this learning problem in addition to the attack traffic. However, generating representative benign traffic for a large and complex enterprise network will require a more complex data-collection infrastructure than the one we use for evaluation. Section 7 discusses this issue in greater detail.

Table 5: LLoCs to implement different problems using netUnicorn and other deployment systems. Here, the three learning problems are (1) Bruteforce detection, (2) video fingerprinting, and (3) APT detection.

Learning Problems	netUnicorn Experiment (Tasks)	Other Deployment Systems Kubernetes	SaltStack	ACI
1	**21** (18)	74	113	61
2	**35** (115)	161	237	179
3	**17** (120)	151	232	176
LLoC Ratio for Experiments + Tasks		$1-2\times$	$2-3\times$	$1-2\times$
LLoC Ratio for Experiments		$3-9\times$	$5-13\times$	$3-10\times$

systems. Specifically, we consider the following deployment systems as baselines: **Kubernetes**, **SaltStack**, and **Azure Container Instances (ACI)**. For each data-collection experiment, we explicitly compose different tasks to realize different data-collection pipelines, create pipeline-specific docker images, and use existing tools (e.g., *kubectl*) to map and deploy these pipelines to different nodes.

6.2 Simplifying Data Collection Effort

We now demonstrate how netUnicorn simplifies data collection for:
Different learning problems for a given network environment (❹). Table 5 reports the effort in expressing the data-collection experiments for the three learning problems for the UCSB network. We observe that netUnicorn only requires 17-35 LLoCs to express the data-collection intent. The UCSB network infrastructure uses SaltStack as the deployment system, and we observe that it takes 113-237 LLoC (around 5-13 × more effort) to express and realize the same data-collection intents without netUnicorn.

The key enabler here is the set of self-contained tasks that realize different data-collection activities. For each learning problem, Table 5 quantifies the overhead of specifying new tasks unique to the problem at hand. Even taking the overheads of expressing these tasks into consideration, collecting the same data from UCSB network without netUnicorn requires around 2-3 × more effort.

Overall, we implemented around twenty different tasks to bootstrap netUnicorn (see the extended version of the paper [13] for more details). The total development effort for the bootstrapping was around **900 LLoCs** . Though this bootstrapping effort is not insignificant, we posit that this effort amortizes over time as this repository of reusable and self-contained tasks will facilitate expressing increasingly disparate data-collection experiments.
Given learning problem from multiple network environments (❺). As we discussed before, netUnicorn is inherently extensible, i.e., it can use different sets of network infrastructures to emulate disparate network environments for data collection. With netUnicorn, changing an existing data-collection experiment to collect data from a new set of network infrastructure(s) requires changing only a few LLoCs (2-3 for the examples in Table 5). In contrast, collecting the data for the HTTP Bruteforce detection problem from a cloud infrastructure (ACI) and a Kubernetes cluster requires writing additional 61 and 74 LLoCs, respectively. This effort is even more intense for video fingerprinting and APT detection problems.

The key enabler for simplifying data collection across one or more network infrastructures is netUnicorn's extensible connectivity manager that can interface with multiple deployment systems via a system of connectors. In [13], we provide a table where we enumerate all the implemented connectors and corresponding logical lines of code (LLoC) for each implementation. Note that this bootstrapping is a one-time effort, and these connectors can be reused across multiple physical infrastructures that are managed using either of the supported deployment systems (e.g., SaltStack, Kubernetes, etc.).
Iterative data collection (❻). To iteratively modify data collection intents, the system should allow flexibility in both pipeline modifications and environment changes. We implemented the experiment, described in Section 5, using netUnicorn, for all three environments (`UCSB`, `UCSB-cloud`, and `multicloud`). We report the combined LLoCs for experiment definitions and tasks implementations in Table 1. As we reused previously implemented connectors, we do not report their LLoC in the table.

The table shows that the overhead for iterative updates is minimal. While this overhead may also be minimal for more conventional (platform- and problem-specific) solutions, netUnicorn's abstractions allow for seamless integration of many other platforms, thus providing a means to increase the diversity of the collected datasets further and, in turn, a model's generalizability capabilities.

6.3 Scaling Data Collection

To quantify the computing and memory overheads of netUnicorn's core and executors (❼), we measure the wall time or elapsed time as a proxy for CPU cycles and use a Python-based memory profiler [67], respectively. Our results show that the executor running on a low-end node such as a Raspberry Pi incurs a computing overhead of approximately **1 second per stage** and **0.13 seconds per task** while consuming less than **21 MB of memory**. Meanwhile, netUnicorn's core incurs a computing overhead of around five seconds for deployment and 20 seconds for execution in a 20-node infrastructure while consuming less than **417 MB of memory**. These experiments are described in more detail in [13], Appendix F.

7 DISCUSSION

More learning problems. While not implemented in this paper, we envision that the netUnicorn platform can be used for a wide range of different network security problems, such as network censorship [3, 14, 53], website fingerprinting [27, 98], Tor traffic analysis [35], and others. Many of these problems involve an active measurement component for data collection, labeling, or communication and would benefit from netUnicorn-provided capabilities such as (i) running experiments that require the simultaneous use of different infrastructures and (ii) facilitating the reproducibility and shareability of experiments. To demonstrate this benefit, we used netUnicorn to implement a multi-vantage point validation of the Let's Encrypt ACME challenge [15]. We provide additional evidence for the practicability and versatility of netUnicorn and its use as part of our newly-proposed ML pipeline by describing in the extended version of this paper [13] (Appendices A and B) the application of our approach to the Let's Encrypt example and two additional real-world security problems, namely Heartbleed detection and OS fingerprinting.

Usability and Realism. First, a critical step in our proposed method is that we require domain experts to articulate data collection intents. As demonstrated in Section 5, it is often possible to generate appropriate intents with the help of explainable ML models. Our platform design further simplifies the process of translating intents into action, ensuring the usability of our proposed method. Second, our data collection follows an emulation-based mechanism that enables accurate labeling. With our proposed iterative approach, we can eliminate biases from the collected data. Additionally, our platform significantly lowers the threshold for gathering data from multiple environments, enhancing the diversity of the data collected. As demonstrated in Section 5, the data we collected is realistic and representative and can improve the generalizability of trained models in various environments.

Limitations of the proposed approach.

Active data collection. Our approach uses endogenously generated (labeled) network data from actual network environments. We note that it may also be possible to improve a model's generalizability by means of carefully selected and exogenously generated (passive) data from a production network, but such an approach is beyond the scope of this paper.

Feature pre-processing. Curating training datasets entails both data collection and pre-processing. Since data pre-processing remains the same for different versions of the collected data that result from our iterative approach, it poses no problems for the desired "thin waist" of netUnicorn's design. In this paper, we utilized the CICFlowmeter for pre-processing, which worked well for all considered learning problems. While we readily acknowledge that there is more to data pre-processing than CICFlowmeter, we leave the exploration of alternative pre-processing (as well as model selection and optimization) techniques for future work.

Decomposing pipelines. We assume that it is possible to decompose a data-collection pipeline into self-contained tasks. However, such a decomposition may be cumbersome for complex learning problems like Puffer [104] that require closer service integration.

Decoupling pipelines from infrastructures. We assume that it is possible to decouple the data-collection intents from actual infrastructure-specific mechanisms. However, realizing this may be difficult, especially for experiments where the data-collection tasks are heavily intertwined with a specific attribute of the data-collection node. For example, some IoT security experiments [99] require running the data-collection pipeline on specific devices with integrated firmware and pre-defined implementations of closed-source services, which cannot be easily supported by netUnicorn.

Programming overheads. Our approach requires experimenters to express new data-collection tasks that are not yet presented in netUnicorn's library. Though this effort will amortize over time, it will only materialize if we succeed in building and incentivizing a broad user community for the proposed platform. Here, we take a first step and make a case for a holistic communal effort to address the data quality and model generalizability issues that have impeded the use of ML-based network security solutions in practice to date.

Limitations of the prototype implementation.

Data-collection nodes. Our current prototype only supports Linux- or Windows-based nodes, optionally with Docker support to enable full platform capabilities (such as Docker container environments). This restriction is reasonable because of the widespread support for Docker-based containers in current data-collection infrastructures [22, 39] and a growing trend to manage Docker-based infrastructures [9, 60]. In future work, we plan to extend support to other computing environments, such as OpenWRT routers and PISA switches, which do not natively support Python or Docker. Currently, such extensions are possible using the sidecar model [97], which allows the configuration of nodes without Python support through Python-based APIs, such as P4-runtime [78].

Potential subjectivity and biases. Applying our proposed closed-loop ML pipeline involves the use of domain experts who themselves can be a source of possible biases or can make subjective decisions. One immediate solution to address this problem is to rely on multiple experts for cross-validation of explanations and decisions regarding data collection. For a more long-term solution, we envision the development of quantitative methods (e.g., metrics for evaluating explanation fidelity [50]) that will facilitate the detection of possible shortcuts or other types of inductive biases.

As far as other bias-related issues are concerned, we are already using a validation set for parameter selection to reduce parameter bias, and our method naturally helps avoid data snooping because it supports collecting data for different tasks and from different network environments at different times and allows for periodically examining and (if necessary) updating trained models.

Manual effort. A concerning side effect of using domain experts as part of our closed-loop ML pipeline is the manual effort it entails. While this makes the current version of our new pipeline inherently semi-automatic, future development of quantitative methods for detecting and possibly eliminating different types of inductive biases promises to reduce the manual effort required and make the pipeline more automatic. The development of such methods could potentially also benefit from advances in how AI can be utilized for examining model explanations and making model modification suggestions, but such issues are beyond the scope of this paper.

8 RELATED WORK

Alternative approaches for our designs. In principle, it is possible to use existing tools and frameworks to realize the "thin waist" we implemented for data collection, but doing so while achieving netUnicorn's level of abstraction, extensibility, fidelity, and scalability poses significant challenges (see [13], Appendix H and Appendix G, for details). For example, one possibility is to disaggregate pipelines into tasks with existing *workflow-management platforms*, such as Airflow [1] or others [31, 64, 69]. However, there is often no explicit support to map these pipelines to specific data-collection nodes and instantiate multiple copies of tasks – limiting data-collection experiments' flexibility. Existing *CI/CD systems* (e.g., Jenkins [57] and others [44, 45] allow explicit mapping of pipelines to nodes but typically require specific infrastructure access and configuration, limiting the desired extensibility and fidelity. Besides, they do not optimize inter-task execution time, limiting their ability

to scale the data collection scenarios. Finally, one can also use different *configuration* (e.g., SaltStack [89]) or *orchestration platforms* (e.g., Kubernetes [60]), and others [4, 25, 82, 102]. However, these systems lack the desired extensibility and flexibility because, being tailor-made for orchestration, they only work for specific types of infrastructures and do not provide explicit support for the proposed pipelines and stages abstraction, limiting tasks and experiments' reusability.

Passive data augmentation. In computer vision, researchers synthesize novel training data by adding random Gaussian noise to training images [95, 100] or blurring, rotating, and flipping them. However, these methods are specific to images and can only rarely be applied beyond vision data. Recent studies propose more application-domain independent methods, such as mixup [107] and SMOTE [24, 59], which can be applied to networking data. However, as demonstrated in Section 5, these methods have limited efficacy in networking applications due to the correctness of the augmented data. They also generate samples that are typically very similar to the given training data, thus limiting the examination of model generalizability. Another line of data augmentation methods generates adversarial samples by adding carefully crafted perturbations to training samples (e.g., [26, 47, 84]). Since these perturbations are just noises with a Non-Gaussian distribution, they suffer from similar limitations as adding Gaussian noise.

Model-side efforts. Various model-side efforts have also been considered to improve model generalizability. In particular, (reinforcement learning-based) domain adaptation methods (e.g.,[40, 92]) maintain an ML model's efficacy across multiple domains. To generalize across different learning problems, existing research proposed multi-task learning [88, 108]) and few-shot learning methods [46, 87]. Researchers have also developed advanced models to combat shortcuts [42] or out-of-distribution (ood) issues [54], such as detecting oods with contrastive learning [106]. All the model-side efforts assume that the training data is fixed and already given. These techniques are orthogonal and complementary to our method, which focuses on improving datasets.

9 CONCLUSION

In this paper, we present a novel closed-loop ML pipeline to curate high-quality datasets for developing generalizable ML-based solutions for network security problems. Our approach is based on a new data-collection method that leverages advances in explainable ML and emphasizes the need for a flexible "in vivo" collection of training datasets. It takes inspiration from the classic "hourglass" abstraction, where the different learning problems make up the hourglass' top layer, and the different network environments constitute its bottom layer. We realize the "thin waist" of this hourglass abstraction with a new data-collection platform, netUnicorn. In effect, for each learning problem, netUnicorn enables data collection in multiple network environments, and for each network environment, it facilitates data collection for multiple learning problems. Through extensive experiments that involve different network security problems and consider multiple network infrastructures, we demonstrate how netUnicorn, in conjunction with the use of explainable ML tools, simplifies data collection for different learning problems from diverse network environments, enables iterative data collection for advancing the development of generalizable ML models, and improves the reproducibility, reusability, and shareability of network security experiments.

ACKNOWLEDGMENTS

We thank the ACM CCS reviewers for their constructive feedback. NSF Awards CNS-2003257, OAC-2126327, and OAC-2126281 supported this work.

REFERENCES

[1] Apache airflow. https://airflow.apache.org.
[2] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu. Atlas: A sequence-based learning approach for attack investigation. In *USENIX Security*, 2021.
[3] Anonymous, A. A. Niaki, N. P. Hoang, P. Gill, and A. Houmansadr. Triplet censors: Demystifying great Firewall's DNS censorship behavior. In *FOCI*, 2020.
[4] Ansible automation platform. https://www.ansible.com/.
[5] D. W. Apley and J. Zhu. Visualizing the effects of predictor variables in black box supervised learning models, 2019.
[6] S. O. Arik and T. Pfister. Tabnet: Attentive interpretable tabular learning, 2020.
[7] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. Dos and don'ts of machine learning in computer security. In *USENIX Security*, 2022.
[8] I. Baldin, A. Nikolich, J. Griffioen, I. I. S. Monga, K.-C. Wang, T. Lehman, and P. Ruth. Fabric: A national-scale programmable experimental network infrastructure. *IEEE Internet Computing*, 2019.
[9] balena - the complete iot management platform. https://www.balena.io/.
[10] K. Bartos, M. Sofka, and V. Franc. Optimized invariant representation of network traffic for detecting unseen malware variants. In *USENIX Security*, 2016.
[11] M. Beck. On the hourglass model. *Commun. ACM*, 62(7):48–57, jun 2019.
[12] R. Beltiukov, S. Chandrasekaran, A. Gupta, and W. Willinger. Pinot: Programmable infrastructure for networking. In *ANRW*, 2023.
[13] R. Beltiukov, W. Guo, A. Gupta, and W. Willinger. In Search of netUnicorn: A Data-Collection Platform to Develop Generalizable ML Models for Network Security Problems (extended version). https://arxiv.org/abs/2306.08853, 2023.
[14] A. Bhaskar and P. Pearce. Many roads lead to rome: How packet headers influence DNS censorship measurement. In *USENIX Security*, 2022.
[15] H. Birge-Lee, L. Wang, D. McCarney, R. Shoemaker, J. Rexford, and P. Mittal. Experiences deploying Multi-Vantage-Point domain validation at let's encrypt. In *USENIX Security*, 2021.
[16] L. Breiman. Random forests. *Machine learning*, 45:5–32, 2001.
[17] F. Bronzino, P. Schmitt, S. Ayoubi, G. Martins, R. Teixeira, and N. Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *POMACS*, 2019.
[18] Cloud computing services - amazon web services. https://aws.amazon.com/.
[19] Cloud computing services - microsoft azure. https://azure.microsoft.com/.
[20] Cloud computing services - digitalocean. https://www.digitalocean.com/.
[21] Cloud computing services - google cloud. https://cloud.google.com/.
[22] Chi@edge. https://chameleoncloud.org/experiment/chiedge/.
[23] E. Chatzoglou, V. Kouliaridis, G. Karopoulos, and G. Kambourakis. Revisiting quic attacks: A comprehensive review on quic security and a hands-on study. *International Journal of Information Security*, 2022.
[24] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *JAIR*, 2002.
[25] Chef infra. http://www.chef.io/chef/.
[26] Z. Chen, Q. Li, and Z. Zhang. Towards robust neural networks via close-loop control. *arXiv preprint arXiv:2102.01862*, 2021.
[27] G. Cherubin, R. Jansen, and C. Troncoso. Online website fingerprinting: Evaluating website fingerprinting attacks on tor in the real world. In *USENIX Security*, 2022.
[28] Canadian institute for cybersecurity datasets. https://www.unb.ca/cic/datasets/index.html.
[29] Cicflowmeter-v4.0. https://github.com/ahlashkari/CICFlowMeter.
[30] A. Cuzzocrea, F. Martinelli, F. Mercaldo, and G. Vercelli. Tor traffic analysis and detection via machine learning techniques. In *Big Data*, 2017.
[31] Dagster. https://dagster.io/.
[32] A. D'Amour, K. Heller, D. Moldovan, B. Adlam, B. Alipanahi, A. Beutel, C. Chen, et al. Underspecification presents challenges for credibility in modern machine learning. *Journal of Machine Learning Research*, 2022.
[33] 1998 darpa intrusion detection evaluation dataset. https://www.ll.mit.edu/r-d/datasets/1998-darpa-intrusion-detection-evaluation-dataset.
[34] Docker. https://www.docker.com/.
[35] P. Dodia, M. AlSabah, O. Alrawi, and T. Wang. Exposing the rat in the tunnel: Using traffic analysis for tor-based malware detection. In *CCS*, 2022.

[36] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani. Characterization of encrypted and vpn traffic using time-related features. In *ICISSP*, 2016.
[37] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*, 2017.
[38] L. D'hooge, T. Wauters, B. Volckaert, and F. De Turck. Inter-dataset generalization strength of supervised machine learning methods for intrusion detection. *Journal of Information Security and Applications*, 54:102564, 2020.
[39] Edgenet. https://www.edge-net.org/.
[40] A. Farahani, S. Voghoei, K. Rasheed, and H. R. Arabnia. A brief review of domain adaptation. In *Advances in Data Science and Information Engineering*, 2021.
[41] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 2001.
[42] R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann. Shortcut learning in deep neural networks. *Nature Machine Intelligence*, 2020.
[43] A. Gepperth and S. Rieger. A survey of machine learning applied to computer networks. In *ESANN*, 2020.
[44] Github actions. https://docs.github.com/en/actions.
[45] Gitlab ci/cd. https://docs.gitlab.com/ee/ci/.
[46] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.
[47] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
[48] M. Gouel, K. Vermeulen, M. Mouchet, J. P. Rohrer, O. Fourmaux, and T. Friedman. Zeph iris map the internet: A resilient reinforcement learning approach to distributed ip route tracing. *SIGCOMM Computer Communication Review*, 2022.
[49] L. Grinsztajn, E. Oyallon, and G. Varoquaux. Why do tree-based models still outperform deep learning on tabular data?, 2022.
[50] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *CCS*, 2018.
[51] S. Gupta and A. Gupta. Dealing with noise problem in machine learning datasets: A systematic review. *Procedia Computer Science*, 2019.
[52] C. Gutterman, K. Guo, S. Arora, T. Gilliland, X. Wang, L. Wu, E. Katz-Bassett, and G. Zussman. Requet: Real-time qoe metric detection for encrypted youtube traffic. *ACM Transactions on MCCA*, 2020.
[53] M. Harrity, K. Bock, F. Sell, and D. Levin. GET /out: Automated discovery of Application-Layer censorship evasion strategies. In *USENIX Security*, 2022.
[54] D. Hendrycks and K. Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv:1610.02136*, 2016.
[55] Hydra. https://github.com/vanhauser-thc/thc-hydra.
[56] A. S. Jacobs, R. Beltiukov, W. Willinger, R. A. Ferreira, A. Gupta, and L. Z. Granville. Ai/ml for network security: The emperor has no clothes. In *CCS*, 2022.
[57] Jenkins. https://www.jenkins.io/.
[58] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. In *USENIX Security*, 2017.
[59] G. Kovács. An empirical comparison and evaluation of minority oversampling techniques on a large number of imbalanced datasets. *ASC*, 2019.
[60] Kubernetes - production-grade container orchestration. https://kubernetes.io/.
[61] I. Kunakorntum, W. Hinthong, and P. Phunchongharn. A synthetic minority based on probabilistic distribution (symprod) oversampling for imbalanced datasets. *IEEE Access*, 2020.
[62] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *SIGCOMM Workshop on Hot Topics in Networks*, New York, NY, USA, 2010. Association for Computing Machinery.
[63] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 2018.
[64] Luigi. https://github.com/spotify/luigi.
[65] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *NeurIPS*. 2017.
[66] K. Maharana, S. Mondal, and B. Nemade. A review: Data pre-processing and data augmentation techniques. *Global Transitions Proceedings*, 2022.
[67] memory-profiler. https://pypi.org/project/memory-profiler/.
[68] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *NDSS*, 2018.
[69] F. Molder, K. Jablonski, B. Letcher, M. Hall, C. Tomkins-Tinch, V. Sochat, J. Forster, S. Lee, S. Twardziok, A. Kanitz, A. Wilm, M. Holtgrewe, S. Rahmann, S. Nahnsen, and J. Koster. Sustainable data analysis with snakemake. *F1000Research*, 2021.
[70] C. Molnar. *Interpretable machine learning*. Lulu. com, 2020.
[71] A. Natekin and A. Knoll. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics*, 7:21, 2013.
[72] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, 2015.
[73] System code of netunicorn. https://github.com/netunicorn/netunicorn.
[74] Library of tasks for netunicorn. https://github.com/netunicorn/netunicorn-library.
[75] Supplementary materials for netunicorn paper. https://github.com/netunicorn/netunicorn-search.
[76] H. Nori, S. Jenkins, P. Koch, and R. Caruana. Interpretml: A unified framework for machine learning interpretability. *arXiv preprint arXiv:1909.09223*, 2019.
[77] ns-3 | a discrete-event network simulator for internet systems. https://www.nsnam.org/.
[78] P4runtime specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html.
[79] Patator. https://github.com/lanjelot/patator.
[80] Platforms for advanced wireless research. https://advancedwireless.org/.
[81] J. Petch, S. Di, and W. Nelson. Opening the black box: The promise and limitations of explainable machine learning in cardiology. *Canadian Journal of Cardiology*, 2022.
[82] Puppet. https://puppet.com/.
[83] J. Quinonero-Candela, M. Sugiyama, A. Schwaighofer, and N. D. Lawrence. *Dataset shift in machine learning*. Mit Press, 2008.
[84] S.-A. Rebuffi, S. Gowal, D. A. Calian, F. Stimberg, O. Wiles, and T. A. Mann. Data augmentation can improve robustness. In *NeurIPS*, 2021.
[85] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1135–1144, New York, NY, USA, 2016. Association for Computing Machinery.
[86] M. Richards. *Software Architecture Patterns: Understanding Common Architecture Patterns and when to Use Them*. O'Reilly Media, 2015.
[87] J. Rivero, B. Ribeiro, N. Chen, and F. S. Leite. A grassmannian approach to zero-shot learning for network intrusion detection. In *ICONIP*, 2017.
[88] S. Ruder. An overview of multi-task learning in deep neural networks. *arXiv:1706.05098*, 2017.
[89] Salt project. https://saltproject.io/.
[90] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security*, 2017.
[91] Seclists. https://github.com/danielmiessler/SecLists.
[92] S. Shankar, V. Piratla, S. Chakrabarti, S. Chaudhuri, P. Jyothi, and S. Sarawagi. Generalizing across domains via cross-gradient training. *arXiv preprint arXiv:1804.10745*, 2018.
[93] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *International Conference on Information Systems Security and Privacy*, 2018.
[94] S. Shi, X. Zhang, and W. Fan. Explaining the predictions of any image classifier via decision trees, 2019.
[95] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 2019.
[96] R. Shwartz-Ziv and A. Armon. Tabular data: Deep learning is not all you need. *Information Fusion*, 2022.
[97] Sidecar. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar.
[98] J.-P. Smith, L. Dolfi, P. Mittal, and A. Perrig. QCSD: A QUIC Client-Side Website-Fingerprinting defence framework. In *USENIX Security*, 2022.
[99] Unsw datasets. https://iotanalytics.unsw.edu.au/.
[100] D. A. Van Dyk and X.-L. Meng. The art of data augmentation. *Journal of Computational and Graphical Statistics*, 2001.
[101] M. Vasić, A. Petrović, K. Wang, M. Nikolić, R. Singh, and S. Khurshid. MoËT: Mixture of expert trees and its application to verifiable reinforcement learning. *Neural Networks*, 151:34–47, jul 2022.
[102] Vmware vsphere. https://www.vmware.com/products/vsphere.html.
[103] F. Wei, H. Li, Z. Zhao, and H. Hu. Xnids: Explaining deep learning-based network intrusion detection systems for active intrusion responses. In *Security*, 2023.
[104] F. Y. Yan, H. Ayers, C. Zhu, S. Fouladi, J. Hong, K. Zhang, P. Levis, and K. Winstein. Learning in situ: a randomized experiment in video streaming. In *NSDI*, 2020.
[105] F. Y. Yan, J. Ma, G. D. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. Pantheon: the training ground for internet congestion-control research. In *USENIX ATC*, 2018.
[106] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang. {CADE}: Detecting and explaining concept drift samples for security applications. In *USENIX Security*, 2021.
[107] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.
[108] Y. Zhang and Q. Yang. An overview of multi-task learning. *NSR*, 2018.
[109] Q. Zhou and D. Pezaros. Evaluation of machine learning classifiers for zero-day intrusion detection–an analysis on cic-aws-2018 dataset. *arXiv preprint arXiv:1905.03685*, 2019.