Log-related Coding Patterns to Conduct Postmortems of Attacks in Supervised Learning-based Projects

FARZANA AHAMED BHUIYAN, Meta
AKOND RAHMAN, Auburn University

Adversarial attacks against supervised learning**a** algorithms, which necessitates the application of logging while using supervised learning algorithms in software projects. Logging enables practitioners to conduct postmortem analysis, which can be helpful to diagnose any conducted attacks. We conduct an empirical study to identify and characterize log-related coding patterns, i.e., recurring coding patterns that can be leveraged to conduct adversarial attacks and needs to be logged. A list of log-related coding patterns can guide practitioners on what to log while using supervised learning algorithms in software projects.

We apply qualitative analysis on 3,004 Python files used to implement 103 supervised learning-based software projects. We identify a list of 54 log-related coding patterns that map to six attacks related to supervised learning algorithms. Using ***Lo****g Assistant to conduct* ***P****ostmortems for* ***Su****pervised* ***L****earning* (***LOPSUL***), we quantify the frequency of the identified log-related coding patterns with 278 open-source software projects that use supervised learning. We observe log-related coding patterns to appear for 22% of the analyzed files, where training data forensics is the most frequently occurring category.

CCS Concepts: • **Security and privacy → Software security engineering;**

Additional Key Words and Phrases: Empirical study, forensics, logging, adversarial machine learning

ACM Reference format:
Farzana Ahamed Bhuiyan and Akond Rahman. 2023. Log-related Coding Patterns to Conduct Postmortems of Attacks in Supervised Learning-based Projects. *ACM Trans. Priv. Sec.* 26, 2, Article 17 (April 2023), 24 pages.
https://doi.org/10.1145/3568020

1 INTRODUCTION

Supervised learning algorithms use training data provided in the form of inputs labeled with corresponding outputs to construct models [69]. Constructed models are then used to make predictions on unseen data [69]. Since the 1990s, supervised learning algorithms have been used in diverse domains, such as finance, healthcare, and transportation [44].

While supervised learning algorithms, such as **Naive Bayes** (**NB**), and **deep neural network** (**DNN**) have yielded benefits, these algorithms are susceptible to attacks [7, 29, 45]. In the context of machine learning, attacks are actions that target supervised learning to cause malfunction [50]. Attacks against supervised learning-based projects can have serious consequences for people's

The research was partially funded by the National Science Foundation (NSF) Award #2247141 and #2209636.
Authors' addresses: F. A. Bhuiyan, Meta, Seattle, Washington; email: fbhuiyan42@fb.com; A. Rahman, Auburn University, Auburn, Alabama; email: akond@auburn.edu.

2471-2566/2023/04-ART17 $15.00
https://doi.org/10.1145/3568020

well-being [19, 20, 54, 55]. Examples include but are not limited to: (i) minuscule changes to an image can malfunction a DNN-based diagnosis software to misclassify a benign mole as malignant [20], and (ii) hand-crafted stickers can reduce the performance of traffic sign classification software by 100%, potentially jeopardizing transportation safety [19].

The above-mentioned examples show that attacks against supervised learning algorithms can have real-world consequences, which necessitates the application of logging so that practitioners can conduct postmortem analysis. The importance of logging in supervised learning-based projects, i.e., software projects that use supervised learning algorithms, have been advocated by policymakers, such as the U.S. and Europe ACM Public Policy Council [24], as well as by researchers [49, 52]. Logging in supervised learning-based projects can help practitioners to perform postmortem analysis of attacks directed towards supervised learning-based projects [52].

Despite the importance of logging, practitioners lack guidance on how logging can be applied while developing supervised learning-based projects [64]. A lack of guidance related to logging can either lead practitioners to not log at all [22, 34], or log too much [22, 34], which can cause performance concerns [75], and hinder troubleshooting [40, 46]. Existing research [43, 76, 82] has provided guidelines on what code elements can be logged, for example, exception blocks, return-value variables, and logic branches. However, these guidelines do not consider the attack types and mechanisms, which are pivotal to detecting and performing postmortem analysis of attacks in supervised learning-based projects [52].

We conduct an empirical study of log-related coding patterns to guide practitioners on what to log while developing supervised learning-based projects. Log-related coding patterns are recurring coding patterns that can be leveraged to conduct adversarial attacks and needs to be logged. Our hypothesis is that through systematic investigation, we can identify log-related coding patterns and the attacks they map to, which can aid practitioners to make informed decisions on what to log to facilitate postmortem analysis of any conducted attacks.

We answer the following research questions:

— ***RQ*$_1$: What categories of log-related coding patterns appear in supervised learning-based projects?** [Section 4]
— ***RQ*$_2$: How frequently do identified log-related coding patterns appear in supervised learning-based projects?** [Section 6]

We conduct our empirical study by applying qualitative analysis on 3,004 Python files to identify log-related coding patterns that map to attacks against supervised learning algorithms. We collect our set of 3,004 Python files from 103 OSS repositories maintained by ModelZoo [2]. We construct a static analysis tool called ***Log Assistant to conduct Postmortems for Supervised Learning*** (***LOPSUL***). We use LOPSUL to quantify the frequency of log-related coding patterns in 278 OSS-supervised learning-based projects. The datasets and source code used to conduct our empirical study are available online [5]. An overview of our methodology is presented in Figure 1. The source code of LOPSUL is available online [8].

Contributions: We list our contributions as follows:

— A list of log-related coding pattern categories for supervised learning-based projects;
— An empirical evaluation of how frequently log-related coding patterns appear in supervised learning-based projects; and
— A tool called LOPSUL to automatically identify log-related coding patterns in software projects.

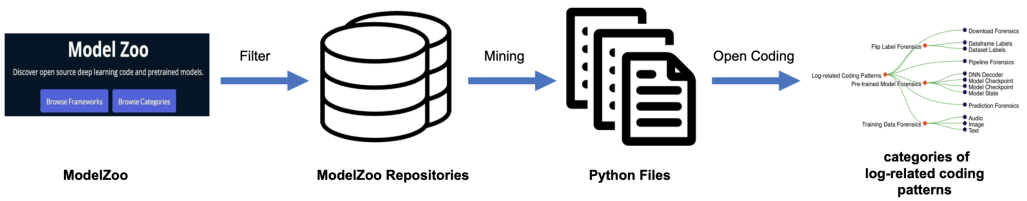

Fig. 1. An overview of our research methodology.

2 MOTIVATING EXAMPLE

We use a hypothetical example to motivate our article. Tracy is a data scientist working for a U.S.-based medical insurance company. As part of automating the process of approving medical benefit claims, Tracy has been asked to develop a binary classification software that will leverage a patient's medical history to decide if a patient will be awarded requested medical benefits. Tracy starts building the classification software by first creating a data import method shown in Listing 1. The data import method is used for data pre-processing, and model building using supervised classification algorithms. Upon construction, the classification software is evaluated using an oracle dataset provided by the company. Satisfied with performance Tracy and higher-ups of the company decide to use the classification software in practice.

Within three months of usage, the company starts noticing benefit claims getting approved by the classification software for patients who are not insured by the company. Higher-ups from the company assume that these are fraudulent claims, possibly done by creating adversarial samples from a user with or without malicious intent, and ask Tracy to investigate if their hypothesis is valid. Unfortunately, as evident from Listing 1 while implementing the data import module for the classification software no logging practices were applied that could have helped Tracy to perform the necessary postmortem analysis. "*I wish I knew code snippets used for data imports can be leveraged to conduct malicious attacks*", Tracy contemplates and starts looking for resources that describe what coding patterns can be leveraged for adversarial supervised learning.

```
1 from pathlib import Path
2 import pandas as pd
3 def import_bill_data(path_to_bill: Path) -> pd.DataFrame:
4     df = pd.read_csv(path_to_bill, header=None, sep=" ")
5     return df
```

Listing 1. Use of `read_csv(path_to_bill)` to import data. The `read_csv()` method can be used to provide dataset with adversarial samples using `path_to_bill`.

Our article aims at helping practitioners on identifying coding patterns that can be leveraged to conduct adversarial attacks in supervised learning-based projects. We refer to these coding patterns as log-related coding patterns, i.e., recurring coding patterns that can be leveraged to conduct adversarial attacks and needs to be logged. In Listing 1, `pd.read_csv()` is an example log-related coding pattern, where the method `pd.read_csv()` can be used to provide datasets with adversarial samples using the `path_to_bill` parameter.

3 BACKGROUND

We provide background information on machine learning, adversarial machine learning, and logging to help the reader gain background necessary for our article.

Machine Learning: ML is the science of getting machines to learn autonomously from real-world interactions and experiences through data that we feed them without being explicitly programmed [32]. ML encompasses a broad range of ML tools, techniques, and ideas. Depending on what type of feedback is available to the learning system, ML techniques are divided into three broad categories: supervised learning, unsupervised learning, and reinforcement learning. SLPs use supervised learning algorithms, that build a mathematical model of a dataset that includes both inputs and outputs [59]. The data is known as training data and consists of a set of training examples. Supervised learning algorithms look for patterns from the training dataset. A high-impact area of progress in supervised learning in recent years involves DNN, which are multi-layer threshold unit networks, each of which calculates simple parameterized function of its inputs [61].

Unsupervised learning algorithms take a dataset that contains only inputs and find structure or commonality in data, such as data point grouping or clustering [25]. Unsupervised algorithms learn from unlabeled data, known as test data. Whereas supervised ML algorithms find patterns in a dataset of correct answers, unsupervised learning tasks look for patterns that are often impossible to identify by humans. Reinforcement learning is an ML area concerned with how software agents should take action in an environment to maximize the notion of cumulative reward [67]. Reinforcement learning, as opposed to providing the computer with correct input-output pairs, provides the machine with a method for measuring its performance with positive reinforcement and the machine learns behavior through trial-and-error interactions with the environment [33].

Adversarial Machine Learning: Although ML involves multiple knowledge-based systems, the data-driven approach of ML presents additional security challenges in the training and testing phases of system operations. ML has become so interconnected with security that the ability of the technical community to implement ML in a secure manner will be vital to future environments [47]. Adversarial machine learning has emerged to study the weaknesses of machine learning approaches in adversarial settings and to develop methods to make learning stable for adversarial exploitation [71]. Adversarial machine learning is concerned with designing ML algorithms that can withstand security challenges, studying the capabilities of attackers, and understanding the consequences of an attack [68]. To make a system secure, it is not sufficient to have an effective strategy, it is also necessary to anticipate the response of the opponent to that strategy [31]. It is important to explore the attacks along with defenses in order to get a deepen understanding of the security issue of ML systems, with the aim of providing an effective defense to mitigate attacks on security-sensitive applications such as autonomous driving, healthcare, and finance.

Logging: Logging is a common programming practice that developers use to record the run-time behavior of a software system for software forensics. Logs have been used for a variety of purposes like debugging [76], system monitoring [51], security compliance [76], and business analytics [6]. Security incidents can arise from the misuse of existing software systems. Thus, appropriate logging mechanisms should be implemented at the software level to support the detection and investigation of security incidents.

Logs are generated during runtime by the output statements that developers insert into the source code. It is crucial to avoid logging too little or too much. To achieve so, developers need to make informed decisions on where to log and what to log in their logging practices during development. There are no well-defined guidelines for software logging [23, 77]. Some of the common information that needs to be logged for easier forensic are the name of the identity provider or security realm that vouched for the username, if that information is available; the affected system component or other objects (such as a user account, data resource, or file); the status that says if the object succeeded or failed; the application context, such as the initiator and target systems, applications, or components; "from where" information for messages related to network connectivity or distributed application operation; and the time stamp and time zone

to help answer "when". Recently, there have been many research works devoted to the area of where-to-log [16, 23, 76, 82], what-to-log [28, 42, 78], and how-to-log [11, 13, 77]. However, most of these works focus on improving the quality of log printing code [12].

4 RQ_1: LOG-RELATED CODING PATTERNS

In this section, we describe our threat model, and provide the methodology to answer ***RQ1****: What categories of log-related coding patterns appear in supervised learning-based projects?* Later in Section 4.3, we provide answers to RQ1.

4.1 Threat Model

Our threat model can have two categories of users: users with malicious intent, and users with no malicious intent. A malicious user performs adversarial attacks with the intent to cause harm to the system, while a regular user may perform adversarial attacks for socially beneficial methods as explained by Albert et al. [4]. In SLPs, malicious users i.e., users with malicious intent, can attempt to manipulate the input data, corrupt the model or tamper with the output with the goal of impacting confidentiality, integrity, availability, and privacy of the systems. A user with no malicious intent can also perform attacks for desirable aims as documented by Albert et al. [4]. Attacks can happen during the training phase or during the testing phase. Attacks at testing time do not tamper with the targeted model but instead either cause it to produce adversary-selected outputs (i.e., integrity attack) or collect evidence about the model characteristics (i.e., confidentiality attack). Attacks on training attempt to corrupt the model itself through explicit attacks or via an untrusted data collection component. Users can perturb the training data by inserting adversarial inputs into the existing training data (injection), or altering the training data directly (modification). Besides tampering with the training data, users may modify the category labels or tamper with the features. Users can tamper with the learning algorithm by colluding with an untrusted ML training component. We define "log-related coding patterns" as recurring coding patterns that can be leveraged to conduct adversarial attacks and need to be logged. We focus on identifying "vulnerable points" within SLPs with the help of log-related coding patterns.

4.2 Methodology for RQ_1

Log-related coding patterns are recurring coding patterns that can be leveraged to conduct adversarial attacks. We use verb-object pairs to determine log-related coding patterns because King et al. [34] reported that verb-object pairs express actions that need to be logged to detect security-related breaches. Our hypothesis is that by identifying verb-object pairs we can determine what coding patterns need to be logged to conduct postmortem analysis if supervised learning-based projects are attacked. We answer RQ_1 using the following steps:

4.2.1 Step-1: Dataset Collection. We use supervised learning-based projects maintained and curated by ModelZoo. ModelZoo is a platform, which curates software projects that use machine learning algorithms, such as supervised learning, deep learning, and reinforcement learning. Many researchers and practitioners are using ModelZoo for different tasks with all kinds of architectures and data [2]. These models are learned and applied for problems ranging from simple regression to large-scale visual classification. Our assumption is that by using software projects maintained by ModelZoo we will be able to apply qualitative analysis to a diverse set of projects that use supervised learning. We download 103 repositories that use supervised learning from ModelZoo on August, 2020. Attributes of the collected repositories are provided in Table 1. While downloading the repositories we delete all data except the Python files and the number of commits to make

Table 1. Attributes of Supervised Learning-based Projects Used in Section 4.2

Attribute	Statistic
Total Repositories	103
Total Commits	11,662
Total Python Files	3,004
Total Lines Of Code	5,71,054
Applications	Audio Speech, Computer Vision, Natural Language Processing, Generative Models

sure we do not include any personal data. We only collect metadata of the repositories that do not include any personal information of the repository contributors or users.

4.2.2 Step-2: Qualitative Analysis. We apply qualitative analysis by applying the following steps:

Step-2.1: Verb-Object Pair Identification. From the downloaded repositories we collect 3,004 Python files that we use to identify verb-object pairs. We use a similar approach to King et al. [34], where a rater manually inspects each file to identify verb-object pairs. We repeat the process for all files in our dataset and identify all unique verb object pairs.

Step-2.2: Validation with CRUD Heuristics. The derivation process of log-related coding patterns is susceptible to rater bias. We mitigate the bias by applying closed coding to determine if the identified verb-object pair maps to **Create, Read, Update, Delete** (**CRUD**) action provided by King et al. [34]. King et al. [34] identified seven CRUD actions that must be logged to detect security breaches for software projects that are used in the healthcare domain. If a mapping exists between the verb-object pair and a CRUD action, then we can mitigate the rater bias that is inherent within the verb-object pairs identified in Step-2.1.

From the collected verb-object pairs from Step-2.1, we *first* determine the action that is expressed. *Next*, we map the expressed action to each of the seven CRUD actions provided by King et al. [34]. The rater uses the definitions for each CRUD action to determine if the identified action can be mapped to a CRUD action.

Step-2.3: Mapping to Supervised Learning Attacks. After separating the verb-object pairs, we determine if the identified verb-object pairs can be used to conduct an attack against a supervised learning algorithm. A rater determines if a verb-object pair can be mapped to an attack by *first*, identifying the action expressed by the verb-object pair. *Second*, the rater examines if the action can be leveraged to conduct an attack by using four publications that describe how attacks can be conducted for supervised learning algorithms. The four publications are: "*SoK: Security and privacy in machine learning*", "*Towards Security Threats of Deep Learning Systems: A Survey*", "*A Survey on Security Threats and Defensive Techniques of Machine Learning: A Data Driven View*", and "*The security of machine learning*", respectively, authored by Papernot et al. [53], He et al. [29], Liu et al. [45], and Barreno et al. [7]. We use the four publications because these publications discuss the categories of adversarial attacks against supervised learning algorithms, such as random forest, and the mechanisms on how to conduct such attacks. *Fourth*, as the final step, the rater separates the verb-object pairs that can be used to conduct attacks, along with the applicable algorithms. Upon completion of this step, we will separate coding patterns that map to attacks against supervised learning-based projects.

Step-2.4: Open Coding to Determine Categories. We apply open coding on the identified coding patterns from Step-2.3. While there are no duplicates amongst the identified coding patterns, semantic similarities may exist between multiple coding patterns. For example, the following two coding patterns, `load_images(params)` and `load_audio(audio_path)` are different in syntax but are similar with respect to semantics, i.e., reading data from a file. We systematically identify these similarities and derive categories using open coding. We use open coding because open coding can

Table 2. Example to Demonstrate Our Qualitative Analysis Approach for RQ_1

Example Coding Pattern	Step-2.1 (Verb Object Pair)	Step-2.2 (CRUD Action)	Step-2.3 (Attack Mapping)	Step-2.4 (Open Coding)
`load_images(params)`	verb:load, object:params	CRUD action: read	action: read, attack: data poisoning	*Category: Training Data Forensics, Coding patterns:* `load_images()`, `load_audio()`
`load_audio(audio_path)`	verb:load, object:audio_path	CRUD action: read	action: read, attack: data poisoning	

be used to generate categories from text mined from software artifacts, e.g., source code snippets. After completion of open coding, the rater identifies a category. Furthermore, the rater separates the verbs from the verb-object pairs from the coding patterns, which was used to derive the category. In our categorization, a coding pattern can belong to multiple categories, as the same coding pattern can map to multiple attacks.

We use two code snippets listed in the column "Example Coding Pattern" of Table 2 to demonstrate our qualitative analysis process. As shown in the "Step-2.1 (Verb Object Pair)" column, we identify `load` and `params`, respectively, as a verb and object for the coding pattern `load_images(params)`. Next, we map the verb-object pair to the CRUD action read, following the definition of King et al. [34]. The action "read" can be used to conduct data poisoning attacks as reported by Papernot et al. [53] and He et al. [29].

Steps-2.1, 2.2, and 2.3 are similar for the two coding patterns: `load_images(params)` and `load_audio(audio_path)`. As both coding patterns can be used to conduct data poisoning attacks by reading training data, we create a category called "Training Data Forensics" in Step-2.4.

Rater verification: The first and second author, who respectively, has experience in software security of 3 and 6 years, individually apply the above-mentioned steps on the collected Python files. Upon completion of the open coding process, the authors discussed their agreements and disagreements. The first and second authors, respectively, identified six and seven categories. The first author identified one category not identified by the second author namely "Download Forensics". At this stage the Cohen's Kappa [14] is 0.6, indicating moderate agreement [39]. Upon completing the discussion, both raters individually revisit their categories and agreed on six security-relevant categories. At this stage the Cohen's Kappa is 1.0.

4.3 Answer to RQ_1

We identify six categories of log-related coding patterns that should be logged in supervised learning-based projects, as shown in Figure 2. We provide the definition, description, corresponding ML attacks, and subcategories for each category. We list the identified coding patterns, corresponding attacks, and applicable classifier algorithms for each category in Table 3.

I. Download Forensics: This category includes coding patterns that can be used to conduct attacks due to malformed input and therefore, need to be logged to enable forensics. For training, supervised learning models need data, which can be downloaded from the Internet. However, unsolicited downloads may result in downloading corrupt data from the Internet that can impact supervised learning model performance. According to Kurita et al. [38], downloading untrusted pre-trained weights poses a security threat. Downloading models from remote sources can facilitate attacks due to malformed input [74]. Therefore, logging needs to be enabled for the coding patterns included in the download forensics category.

In Listing 2, logging needs to be enabled for the coding pattern `wget.download()`, because if we have the information of the source of the remote dataset, it could help the practitioner determine if an attack occurred and perform necessary postmortem analysis. We identify seven log-related coding patterns that belong to the download forensics category.

Fig. 2. A taxonomy of log-related coding patterns for supervised learning-based projects.

```
if args.tar_path and os.path.exists(args.tar_path):
    target_file = args.tar_path
else:
    wget.download(TED_LIUM_V2_DL_URL, target_dl_dir)
    target_file = os.path.join(target_dl_dir, "TEDLIUM_release2.tar.gz")
```

Listing 2. Example of a coding pattern that belongs to the download forensics category.

II. Flip Label Forensics: Label perturbation attack usually happens when labels of training data are collected from external sources. For instance, a collaborative spam filtering process updates the e-mail classifier based on feedback from end-users, where malicious users can mislabel e-mails in their inboxes to feed false information to the update method [80]. A malicious user can significantly reduce the performance of supervised learning algorithms by flipping the labels of train data [80]. This technique can be used to effectively fool road sign classifiers for autonomous vehicles by perturbing the labels for a fraction of the training data. To keep track of whether or not labels are being manipulated or not, logging needs to be enabled. Flip label forensics includes two subcategories:

II-A. Creating Labels with Dataframe Manipulations: This subcategory includes coding patterns that can be used to conduct label perturbation attacks while creating labels through dataframe manipulations and therefore, need to be logged to enable forensics. Code snippets, such as `hfw.create_dataset(''labels'')` can be used to create labels by manipulating dataframes. However, malicious users might perturb the created labels to perform label perturbation attacks. In case of such an attack, it can be helpful to have the logged information to troubleshoot.

In Listing 3, logging needs to be enabled for the coding pattern `hfw.create_dataset()`, because if we have the information of the source of the labels, in case of a label perturbation attack it could help the practitioner perform necessary postmortem analysis. We identify four log-related coding patterns that belong to this subcategory.

```
label = hfw.create_dataset("labels", data=df_attr[list_col_labels].values)
label.attrs["label_names"] = list_col_labels
```

Listing 3. Example of a coding pattern that belongs to the creating labels with dataframe manipulations subcategory.

Table 3. Answer to RQ_1: Log-related Coding Pattern Categories and Corresponding Attacks

Categories	Coding Patterns	Attack	Algorithms
Download Forensics	wget.download(); urlopen(); prepare_url_image(); load_url(); misc.download_model(); latest_blob.download_to_filename(); _download(); download_from_url();	Malformed input: In malformed input attack the malicious user concocts an input to the supervised learning system that reliably produces an output different from the intended output [26].	DNN
Flip Label Forensics	read_h5file(); hf.get(); load_data_and_labels(); load_image(); scipy.io.loadmat(); hfw.create_dataset(); interpreter.get_tensor(); evaluate(); coco_gt.loadRes();	Label perturbation: The baseline strategy of label perturbation attack is to perturb the labels for a fraction of the training data to reduce the prediction accuracy of supervised learning systems [53].	Support Vector Machine (SVM), Logistic Regression (LR)
Pipeline Forensics	pipeline_pb2.TrainEvalPipelineConfig(); get_configs_from_pipeline_file(); ArgumentParser();	Physical domain: In physical domain attacks, malicious users find perturbations preserved by the data pipeline that precedes the classifier in the overall targeted system [37].	DNN
Prediction Forensics	get_tensor(); show_data_summary();	Model stealing: Model stealing attack attempts to replicate a supervised learning model via the APIs provided, without prior knowledge of training data and algorithms [30].	LR, SVM, DNN
Pre-trained Model Forensics	load_decoder(); load_previous_values(); load_pretrained(); patch_path(); sp_model.Load(); load_model_package(); load_model(); load_state_dict(); load_param(); load_checkpoint();	Model poisoning: In model poisoning attack, a malicious user pollutes a supervised learning model with certain latent behavior, to be unwittingly adopted by third parties and later exploited by the malicious user [53].	LR, SVM, MLP, DNN
Training Data Forensics	open(); load_celebA(); load_images(); load_wav(); load_randomly_augmented_audio(); load_generic_audio(); load_audio(); _load_vocab_file(); json.load(); load_lua(); get_raw_files(); load_attribute_dataset(); load(); upload_from_filename(); read_file(); from_tensor_slices(); read_csv(); MNIST(); open(); File(); frombuffer(); get_loader(); read_h5file();	Data poisoning: Data poisoning attack aims to reduce the prediction accuracy of supervised learning systems by polluting training data in a manner so that it is imperceptible to the human eye [53]	NB, SVM, DT, MLP, DNN

II-B. Loading Labels From Datasets Where Labels are Predefined: This subcategory includes coding patterns that can be used to conduct label perturbation attacks by perturbing the predefined labels in a dataset and therefore, need to be logged to enable forensics. Code snippets, such as `hf.get('label')` can be used to load labels from a dataset. In supervised learning, the labels might be loaded from a remote or local file. Loading classification labels from the file can facilitate label perturbation attacks [53] as malicious users may change the labels used to train models. In case of such an attack, it can be helpful to have the logged information to troubleshoot.

In Listing 4, logging needs to be enabled for the coding pattern `hf.get()`, because if we have the information of the source of the loaded predefined labels, in case of a label perturbation attack it could help the practitioner perform necessary postmortem analysis. We identify five log-related coding patterns that belong to this subcategory.

```
with h5py.File(path, 'r') as hf:
    data = np.array(hf.get('data'))
    label = np.array(hf.get('label'))
```

Listing 4. Example of a coding pattern that belongs to the loading labels from datasets where labels are predefined subcategory.

III. Pipeline Forensics: This category includes coding patterns that can be used to conduct physical domain attacks [53] while loading pipeline configurations and, therefore, need to be logged to enable forensics. A machine learning pipeline includes the following stages: training the model, evaluating the model, deploying the model, and using the model for predictions. Data pipelines used in supervised learning often are susceptible to attacks. In physical domain attacks, malicious users find perturbations preserved by the data pipeline that precedes the classifier in the overall targeted system [37]. When the malicious user is unable to directly modify feature values used as model inputs, a physical domain attack helps to reduce the accuracy of the model classification.

As a consequence of a physical domain attack, autonomous vehicles may over speed if the road sign recognition models inside were compromised [18]. In such a physical attack, a malicious user change existing physical road signs with adversarial perturbations that is the change is done after the training process but before the deploying process of the pipeline. To keep track of whether or not an attack on the data pipeline is happening, logging needs to be enabled.

In Listing 5, logging needs to be enabled for the coding pattern `ArgumentParser()`, because if we have the information of the source of the loaded pipeline configurations, in case of an adversarial attack it could help the practitioner determine if an attack occurred and perform necessary postmortem analysis. We identify three log-related coding patterns that belong to the pipeline forensics category.

```
parser = argparse.ArgumentParser(description='Input pipeline')
parser.add_argument('--audio_dir', default=os.path.expanduser(pathToAudioFile), help='Location of sound
  ↪  files')
args = parser.parse_args()
```

Listing 5. Example of a coding pattern that belongs to the pipeline forensics category.

IV. Prediction Forensics: This category includes coding patterns that can be used to conduct model stealing attacks and, therefore, need to be logged to enable forensics. Model stealing attack attempts to replicate a supervised learning-based model via the APIs provided, without prior knowledge of training data and algorithms [30]. In this attack, the malicious users first submit input to the target model and get the predicted values. Then they use input-output pairs and methods to extract confidential data including parameters, hyper-parameters, architectures, decision boundaries, and functionality. The malicious user could use the stolen model to extract private information contained in the training data of the original model or to construct adversarial examples that will force the victim model to make incorrect predictions [36].

A malicious user may perform malicious activities by continuously getting prediction output of supervised learning models using certain inputs by performing model stealing attacks [30, 62]. A malicious user might aim at leveraging model predictions to compromise user privacy. For instance, Fredrikson et al. [21] demonstrated that using prediction results, attacks can infer an individual's private genotype information. Model stealing attacks compromise the intellectual property and algorithm confidentiality of the learner [70, 72]. To keep track of whether or not a model stealing attack is happening, logging needs to be enabled.

In Listing 6, logging needs to be enabled for the coding pattern `get_tensor()`, because if we have the information of when the prediction outputs of supervised learning models are shown, in case of a model stealing attack it could help the practitioner determine if an attack occurred and perform necessary postmortem analysis. We identify two log-related coding patterns that belong to the prediction forensics category.

```
tensor_name = 'AttentionOcr_v1/InceptionV3/Conv2d_2a_3x3/moving_mean'
reader = tf.compat.v1.train.NewCheckpointReader(_CHECKPOINT)
moving_mean_expected = reader.          (tensor_name)
```

Listing 6. Example of a coding pattern that belongs to the prediction forensics category.

V. Pre-trained Model Forensics: This category includes coding patterns that can be used to conduct model poisoning attacks by importing pre-trained models, i.e., models that are constructed *a priori* and therefore, need to be logged to enable forensics. *A priori* supervised learning models can be imported using binary files. Loading a pre-trained model can facilitate model poisoning attacks

[30]. Kurita et al. [38] showed that it is possible to construct attacks where pre-trained models are injected with vulnerabilities that expose backdoors after fine-tuning, enabling the malicious user to manipulate the model prediction simply by injecting an arbitrary keyword. In supervised learning, a backdoor is similar to a hidden behavior of the model, which only happens when it is queried with an input containing a secret trigger [60]. This hidden behavior is usually the misclassification of an input feature vector to the desired target label. Kurita et al. [38] showed how a pre-trained poisonous model that is indistinguishable from a non-poisoned model as far as the task performance is concerned reacts to the trigger keyword in a way that systematically allows the malicious user to control the model's output. When loading such binary files, logging needs to be enabled to keep track of what binary files are being loaded, and if corrupted model files are being loaded or not. The pre-trained model forensics category includes the following subcategories:

V-A. Poisonous Model Checkpoint: This subcategory includes coding patterns that can be used to conduct model poisoning attacks by poisoning saved model checkpoints and therefore, need to be logged to enable forensics. A checkpoint is an intermediate dump of a model's entire internal state, such as its weights, current learning rate, and so on. so that the framework can resume the training from this point whenever desired. Code snippets, such as `load_checkpoint()` can be used to load model checkpoints. When training deep learning models, the checkpoint is the weight of the model. These weights can be loaded to make predictions as is, or used as the basis for ongoing training. However, malicious users might inject malicious data to change the model checkpoints, and loading those poisonous checkpoints can cause a model poisoning attack [30]. The user model may carry a backdoor after fine-tuning the pre-trained injected weights which allows the malicious user to manipulate model prediction [30]. In case of such an attack, it can be helpful to have the logged information to troubleshoot.

In Listing 7, logging needs to be enabled for the coding pattern `load_checkpoint()`, because if we have the information of what model checkpoint was used in case of a poisonous attack, it could help the practitioner perform necessary postmortem analysis. We identify one log-related coding pattern that maps to poisonous model checkpoint attacks.

```
1 args1, auxs1 = load_checkpoint(prefix1, epoch1)
```

Listing 7. Example of a coding pattern that belongs to the poisonous model checkpoint subcategory.

V-B. Pre-trained DNN: This subcategory includes coding patterns that can be used to conduct model poisoning attacks by poisoning pre-trained DNN and therefore, need to be logged to enable forensics. Code snippets, such as `load_decoder()` can be used to load a pre-trained decoder. Loading a pre-trained decoder is a way to initialize the weights when training DNNs. Initialization with pre-training can have better convergence properties than simple random training. Since it is common for users to build on and deploy DNN models designed and trained by third parties [73], adversaries may alter the model's behavior by manipulating the data that is used to train it. For example, Gu et al. [10] generated a backdoor in a street sign classifier by inserting images of stop signs with a special sticker into the training set and labeling them as speed limits. As a result, the model learned to properly classify standard street signs, but misclassify stop signs possessing the backdoor trigger. Thus, adversaries can trick the model by executing this attack to identify any stop sign as a speed limit simply by putting a sticker on it, causing possible accidents in self-driving cars. It is difficult to detect this type of attack given that backdoor triggers are, absent further analysis, only known by adversaries [10]. As the pre-trained decoder might be poisonous, logging can be used to postmortem the attacks in such cases.

In Listing 8, logging needs to be enabled for the coding pattern `load_decoder()`, because if we have the information of which model was used in case of a poisonous attack, it could help the

practitioner perform necessary postmortem analysis. We identify one log-related coding pattern that maps to poisonous pre-trained DNN attacks.

```
model = load_model(device =device, model_path =cfg.model.model_path, use_half =cfg.model.use_half)
decoder = load_decoder(labels=model.labels, cfg =cfg.lm)
```

Listing 8. Example of a coding pattern that belongs to the pre-trained DNN subcategory.

V-C. Pre-trained Model Parameters: This subcategory includes coding patterns that can be used to conduct model poisoning attacks by poisoning pre-trained model parameters and therefore, need to be logged to enable forensics. Model parameters include parameters, such as learning rate, batch size, momentum, bias, and weight decay. Code snippets, such as `load_param()` can be used to load pre-trained model parameters. In machine learning, parameters are important, as for the same training dataset, if we change the value of the parameters of a supervised algorithm, the supervised algorithm could learn models with significantly varying performance on the test dataset [72]. Model poisoning attacks can be designed by poisoning the model parameters. If the loaded parameters are poisonous, then it can be helpful to have the logged information to troubleshoot if a model poisoning attack occurs.

In Listing 9, logging needs to be enabled for the coding pattern `load_param()`, because if we have the information of what pre-trained model parameters were used in case of a poisonous attack, it could help the practitioner perform necessary postmortem analysis. We identify one log-related coding pattern that maps to poisonous model parameter attacks.

```
if config.TRAIN.RESUME:
    arg_params, aux_params = load_param(prefix, begin_epoch, convert=True)
else:
    arg_params, aux_params = load_param(pretrained, epoch, convert=True)
```

Listing 9. Example of a coding pattern that belongs to the pre-trained model parameters subcategory.

V-D. Pre-trained Model State: This subcategory includes coding patterns that can be used to conduct model poisoning attacks by poisoning pre-trained model states, that is the weights and architecture of a pre-trained model and, therefore, need to be logged to enable forensics. Code snippets, such as `load_previous_values()` can be used to load pre-trained model states. malicious users might insert malicious input to change the model states and as long as the resulting models have high predictive capacity for the specified tasks, without knowing what this code is doing, benign users use the pre-trained model states [65]. If the loaded model states are poisonous, then it can be helpful to have the logged information to troubleshoot if a model poisoning attack occurs.

In Listing 10, logging needs to be enabled for the coding pattern `load_pretrained()`, because if we have the information of which pre-trained model was used in case of a poisonous attack, it could help the practitioner perform necessary postmortem analysis. Altogether we identify seven log-related coding patterns that map to pre-trained model state attacks.

```
model = models.alexnet(pretrained=False)
if pretrained is not None:
    settings = pretrained_settings['alexnet'][pretrained]
    model = load_pretrained(model, num_classes, settings)
```

Listing 10. Example of a coding pattern that belongs to the pre-trained model state subcategory.

VI. Training Data Forensics: This category includes coding patterns that can be used to conduct data poisoning attacks and therefore, need to be logged to enable forensics. Poisoning attack aims at reducing the prediction accuracy of supervised learning-based systems by polluting training data in a manner so that it is imperceptible to the human eye [53]. In this attack, data are altered by data injection or data manipulation. Adversarial inputs are inserted into the original training data, thereby changing the underlying data distribution without changing the features or labels of the original training data [68]. As a consequence, the poisoned model could not represent the correct data and is prone to making the wrong predictions. Steinhardt et. al [66] reported that, even under strong defenses, a 3% training dataset poisoning leads to an 11% drop in accuracy.

Training data forensics is different from download forensics because download forensics maps to malformed input attacks. In a malformed input attack, the malicious user concocts an input to the supervised learning system, such as input data, training data, or models that come from external sources to reliably produces an output different from the intended output [26]. Supervised learning models can be poisoned using datasets that are inherently incorrect and this poisonous training data can facilitate data poisoning attacks [53]. The impact of data poisoning attacks can be fatal for many businesses and industries, and even life-threatening for the healthcare sector, the aviation industry, or road safety. For instance, a malicious user may add new adversarial training data to a healthcare ML model to falsely classify a hypothyroid patient [48]. In case of such an attack, if we can map the loaded dataset to the attack, it will help to facilitate postmortem analysis of the conducted attack. That is why it is important to log whenever a data loading event is used for training. Training data forensics include the following subcategories:

VI-A. Audio Poisoning: This subcategory includes coding patterns that can be used to conduct data poisoning attacks by poisoning audio datasets used for training and, therefore, need to be logged to enable forensics. Using code snippets, such as `load_audio()`, an audio file is being imported. However, there have been several attempts at producing targeted adversarial attacks on automatic speech recognition using poisonous audio data. Given a natural waveform x, Carlini and Wagner [9] were able to construct a perturbation δ that was nearly inaudible but $x+\delta$ is recognized as any desired phrase. They were able to construct 10 adversarial examples simultaneously and reported to achieve 100% success in generating the targeted adversarial examples for each of the source-target pairs. Poisonous audio data can be used for impersonation attack [63], a malicious user can use the audio data maliciously to authorize the fraudulent credit card or utility charges. If the loaded audio file is poisonous, then it can be helpful to have the logged information to troubleshoot if an audio-related poisoning attack occurs.

In Listing 11, logging needs to be done for the coding pattern `load_audio()` and `load_randomly_augmented_audio()`, because if we have the logged information of the loaded audio file in case of a poisonous attack, this can help the practitioner to perform necessary postmortem analysis. Altogether we identify five log-related coding patterns that map to audio poisoning attacks.

```
def parse_audio(self, audio_path):
    if self.aug_conf and self.aug_conf.speed_volume_perturb:
        y = load_randomly_augmented_audio(audio_path, self.sample_rate)
    else:
        y = load_audio(audio_path)
```

Listing 11. Example of a coding pattern that belongs to the audio poisoning subcategory.

VI-B. Image Poisoning: This subcategory includes coding patterns that can be used to conduct data poisoning attacks by poisoning image datasets and, therefore, need to be logged to enable

forensics. For image classification, adversarial examples are intentionally synthesized images in the training set, which look almost the same as the original images, but can mislead the classifier to provide wrong prediction outputs. Using a poisonous image dataset, malicious users may compromise real-world systems with adversarial examples without breaking into the system. For instance, malicious users may freely pass face authentication-based entrance access doors if the face authentication models were compromised [63]. An example from Sharif et al. [63] shows that image-related data poisonous attacks can have implications for authentication, which demonstrates the need to apply logging for coding patterns that are used to input image datasets. As shown in Table 3, code snippets, such as `Image.open()`, `load_images(params)`, and `load_celebA(img_dim)` are used to load training data from local directory.

In Listing 12, logging needs to be done for the coding pattern `load_images()`, because if we have the information of the image file used in case of a poisonous attack, it could help the practitioner perform necessary postmortem analysis. Altogether we identify three log-related coding patterns that map to image poisoning attacks.

```
data, attributes = load_images(params)
train_data = DataSampler(data[0], attributes[0], params))
```

Listing 12. Example of a coding pattern that belongs to the image poisoning subcategory.

VI-C. Text Poisoning: This subcategory includes coding patterns that can be used to conduct data poisoning attacks by poisoning text datasets used for training and therefore, need to be logged to enable forensics. Code snippets, such as `json.load()` can be used to load training data that are in text form. Poisonous attacks using training text data can be done by inserting typos to a sentence that can fool text classification or dialogue systems [17]. Ebrahimi et al. [17] showed a method for generating adversarial examples with character substitutions and reported that while character-edit operations have little impact on human understanding, character-level models are highly sensitive to adversarial perturbations. If the practitioner who developed the classification model is not aware of the poisonous text, the malicious user can leverage it to get the supervised learning-based system to do what they want. For example, the substitution of carefully selected synonyms can cause a classification software to misclassify opioid abuse risk [20]. In case of such a poisonous attack using training text data, logging can be used to track the attacks.

In Listing 13, logging needs to be done for the coding pattern `json.load()`, because the application of logging can help the practitioner determine if an attack occurred and perform necessary postmortem analysis. Altogether we identify 15 log-related coding patterns that map to text poisoning attacks.

```
if not annotation_file == None:
    dataset = json.load(open(annotation_file, 'r'))
```

Listing 13. Example of a coding pattern that belongs to the text poisoning subcategory.

Differences with Prior Research Our findings from RQ_1 can complement existing logging-related research [28, 34, 42]. Li et al. [42] assumed that developers of a project can keep consistent logging practices design and based on the assumption they proposed a regression model to recommend the log level in a logging statement. He et al. [28] categorize the logging descriptions by conducting an empirical study on the natural language descriptions of logging statements based on the purpose of those descriptions. Using the categorization, they designed a method to automatically generate static log descriptions. Compared to He et al. [28]'s research ours is more prioritized as

Fig. 3. An overview of constructing the oracle dataset.

we only have identified coding patterns that are related to supervised learning algorithms. King et al. [34] provided heuristics but did not identify coding patterns that map to attacks related to supervised learning algorithms. In short, none of the above guidelines consider the attack types and mechanisms, which are pivotal to detecting and performing postmortem analysis of attacks for supervised learning-based projects [52].

5 LOPSUL

LOPSUL is a static analysis tool that can identify the six categories of log-related coding patterns listed in Table 3. As input, the practitioner will provide the path where the supervised learning repositories reside, and LOPSUL will (i) report the location of the identified log-related coding pattern and (ii) output the count for each detected category in a file.

Log-related Coding Patterns Detection Process. Here we describe how LOPSUL detects log-related coding patterns:

Parsing: LOPSUL uses the AST of a Python file to detect log-related coding patterns. LOPSUL parses each Python file into an AST. *First*, LOPSUL mines the ASTs and identifies code elements, such as class objects, exception classes, function declarations and their arguments, variable assignments, and library imports. *Second*, LOPSUL applies pattern matching to identify if any of the coding patterns listed in Table 3 appear in the mined code elements. LOPSUL uses the Python ast library [1] for parsing.

Evaluation of LOPSUL: Static analysis tools are subject to evaluation [57]. We evaluate LOPSUL's accuracy using an oracle dataset. A graduate student, who is not an author of the article, volunteered to construct the oracle dataset. The student has 5 years of experience in software security. We use 156 randomly-selected files from the ModelZoo repositories. The student worked as a rater and constructed the dataset using closed coding, which is the process of mapping an entry to a pre-defined category [15]. The rater applied closed coding to identify which of the log-related coding pattern categories appear in the provided 156 files. The rater read each of the 156 Python files and assign the categories. We do not impose any time limit for the rater to conduct closed coding. We describe the process of creating the oracle dataset in Figure 3. We provided the rater a guidebook that included the names, definitions, and examples of each category. The guidebook is available online [5].

The rater took 48 hours to conduct closed coding. Upon completion of the closed coding process, we apply LOPSUL on the oracle dataset and compute LOPSUL's precision and recall for the oracle dataset. Precision refers to the fraction of correctly identified categories among the total identified categories, as determined by LOPSUL. Recall refers to the fraction of correctly identified categories that have been retrieved by LOPSUL. The first author inspected the rater's labeling and did not identify any log-related coding patterns missed by the rater. Altogether, the rater identifies 86 instances of log-related coding patterns that appeared in 44 files. The average precision and recall of LOPSUL are, respectively, 0.87 and 0.98. A complete breakdown of LOPSUL's precision and recall values is provided in Table 4. For the new dataset, we observe LOPSUL's precision to be >0.90, and

Table 4. Evaluation of LOPSUL with Oracle Dataset Constructed from ModelZoo

Categories	Count	Precision	Recall
Download Forensics	1	1.00	1.00
Flip Label Forensics	1	0.50	1.00
Pipeline Forensics	9	0.89	0.89
Prediction Forensics	9	1.00	1.00
Pre-trained Model Forensics	3	1.00	1.00
Training Data Forensics	63	0.85	1.00
Average		0.87	0.98

Table 5. Evaluation of LOPSUL with Oracle Dataset Constructed from GitHub and GitLab

	Count		Precision		Recall	
Categories	GITHUB	GITLAB	GITHUB	GITLAB	GITHUB	GITLAB
Download Forensics	1	1	1	1	1	0
Flip Label Forensics	1	2	1	1	0	1
Pipeline Forensics	5	7	0.83	0.58	1	1
Prediction Forensics	7	17	0.86	0.85	0.86	1
Pre-trained Model Forensics	7	8	1	1	0.57	0.88
Training Data Forensics	48	72	1	1	0.96	0.97
Average			0.95	0.91	0.73	0.81

recall to be >0.70, which shows LOPSUL to generate not a lot of false positives, while missing a few log-related coding patterns.

The oracle dataset could be limiting to evaluate LOPSUL's detection accuracy. We mitigate this limitation by constructing another dataset and evaluating LOPSUL with this extra dataset. The first author, who has experience in software engineering and software security of 3 years, constructs another oracle dataset using files from GitHub and GitLab. We randomly select 500 files from the GitHub repositories and 500 files from the GitLab repositories. The first author applied closed coding to identify which of the log-related coding pattern categories appear in the 1,000 files. Altogether, the first author identifies 176 instances of log-related coding patterns. For GitHub, the average precision and recall of LOPSUL are, respectively, 0.95 and 0.91. For GitLab, the average precision and recall of LOPSUL are, respectively, 0.73 and 0.81. A complete breakdown of LOPSUL's precision and recall values are provided in Table 5.

6 RQ_2: FREQUENCY OF LOG-RELATED CODING PATTERNS

In this section, we provide the methodology and results to answer **RQ_2: *How frequently do identified log-related coding patterns appear in supervised learning-based projects?***

6.1 Methodology for RQ_2

We answer RQ_2 (i) by mining OSS repositories that use supervised learning, and (ii) using metrics to quantify the frequency of log-related coding patterns.

6.1.1 Repository Mining. We answer RQ_2 by mining OSS repositories. Our categories are derived from the ModelZoo dataset. Quantifying the frequency of the identified log-related coding patterns in multiple datasets could increase the generalizability of our findings. We use three data sources: (i) OSS GitHub repositories, (ii) OSS GitLab repositories, and (iii) ModelZoo repositories. We use these three repositories as popular SLPs are hosted on these repositories [3]. Our assumption is that by collecting repositories from the three data sources we will be able to quantify the prevalence of log-related coding patterns for projects that use supervised learning.

Table 6. Answer to RQ_2: Frequency of Log-related Coding Patterns

	Count			PropFile (Per File)			Density (Min, Max, Median)		
Categories	**GITHUB**	**GITLAB**	**MODELZOO**	**GITHUB**	**GITLAB**	**MODELZOO**	**GITHUB**	**GITLAB**	**MODELZOO**
Download Forensics	27	8	42	0.13	0.08	1.62	(0.00,0.43,0.00)	(0.00,0.31,0.00)	(0.00,7.37,0.00)
Flip Label Forensics	43	54	44	0.23	0.22	1.25	(0.00,0.49,0.00)	(0.00,0.34,0.00)	(0.00,7.35,0.00)
Pipeline Forensics	184	227	242	1.11	1.97	7.96	(0.00,2.71,0.00)	(0.00,1.83,0.00)	(0.00,5.99,0.00)
Prediction Forensics	579	1,135	101	0.97	4.45	2.38	(0.00,4.18,0.00)	(0.00,4.59,0.16)	(0.00,7.87,0.00)
Pre-trained Model Forensics	46	238	149	0.28	1.08	5.92	(0.00,1.16,0.00)	(0.00,1.83,0.00)	(0.00,10.10,0.00)
Training Data Forensics	4,614	3,167	1,383	11.79	15.12	32.10	(0.00,16.28,1.05)	(0.00,12.72,1.77)	(0.00,40.40,4.15)
Total	5,493	4,829	1,961	13.52	21.35	36.79	(0.00,17.44,1.28)	(0.00,12.97,3.41)	(2.19,9.01,7.11)

Table 7. Selection Criteria to Construct Datasets

Criteria	**GITHUB**	**GITLAB**	**MODELZOO**
Initial	3,405,303	546,000	411
Criterion-1	611	636	176
Criterion-2	541	430	163
Criterion-3	487	139	127
Criterion-4	109	66	103
Final	109	66	103

Table 8. Attribute of the Three Datasets

Attribute	**GITHUB**	**GITLAB**	**MODELZOO**
Total Repositories	109	66	103
Total Commits	4,03,196	65,714	11,662
Total Python Files	22,212	9,086	3,004
Total Lines Of Code of Python Files	62,19,441	16,91,060	5,71,054

We apply filtering criteria to identify quality repositories: *Criterion-1:* We select repositories where the percentage of Python files is greater than 50% of the total files in the repository. *Criterion-2:* We select repositories that have at least five commits per month as it indicates these repositories have enough development activities. *Criterion-3:* We select repositories that have at least 10 contributors. *Criterion-4:* Since we are interested in supervised learning-based development, we select only those repositories that are related to supervised learning-based projects. To select the supervised learning-based repositories, we used the README files of the repositories, as the README files describe the content of the project. We inspect the README file for each repository to determine if the repository uses supervised learning algorithms, such as DNN to develop a software feature. Using all the above criteria, we collected 109, 66, and 103 repositories, respectively, for Github, Gitlab, and ModelZoo datasets. We describe how many of the repositories satisfied each of the four criteria in Table 7. Attributes of the repositories are available in Table 8.

6.1.2 Metrics for Frequency Analysis. Upon collection of the repositories, we run LOPSUL on 278 repositories and answer RQ_2 using three metrics: (i) Count, (ii) PropFile, and (iii) Density. Using the "PropFile(x)" metric we quantify the proportion of files that are identified having one or more categories of log-related coding patterns. Using the "Density(x)" metric we quantify the frequency of the presence of each category. We use Equations (1) and (2), respectively, to calculate "PropFile" and "Density".

$$\text{PropFile}(x) = \frac{\text{\# of files with >= 1 log-related coding pattern of category } x}{\text{total Python files in the repository}} \tag{1}$$

$$\text{Density}(x) = \frac{\text{\# of log-related coding pattern with category } x}{\frac{\text{total lines of code in the repository}}{1000}} \tag{2}$$

6.2 Answer to RQ_2

We identify 12,283 instances of log-related coding patterns in 278 OSS repositories for supervised learning. The most frequent category is training data forensics. A breakdown of the categories count for the three datasets is provided in Table 6. Considering all categories, the total count of identified log-related coding patterns is 5,493, 4,829, and 1,961, respectively, for GitHub, Gitlab, and ModelZoo as shown in "Total" for Table 6.

In the "PropFile (Per File)" column of Table 6, we report the PropFile metric. The "Total" row presents the PropFile for each dataset when all six categories are considered. For all three datasets, we observe the dominant category is training data forensics. We observed 13.52%, 21.35%, and 36.79% files, respectively, for Github, Gitlab, and ModelZoo repositories, to contain at least one category of log-related coding patterns, as shown in the "PropFile" column.

We describe the minimum, maximum, and median values for the "Density" metric, respectively, in the column "Density (Min, Max, Median)" of Table 6. The median values of the "Density" metric for four of the six categories are 0.00 for all three repositories. Considering all six categories, the minimum, maximum, and median values for Github repositories are respectively, 0.00, 17.44, and 1.28 as shown in the "Density" column.

7 DISCUSSION

Implications Related to Accountability As the use of supervised learning is becoming prevalent in critical domains, such as healthcare [20, 54], accountability is of paramount importance to all stakeholders [27]. At a tutorial in NeurIPS 2018 [49, 56] researchers considered logging in machine learning development as pivotal to facilitate accountability. Our derived log-related coding pattern categories listed in Table 3 can help practitioners to integrate accountability into machine learning development, especially when attacks are launched. If supervised learning algorithms are attacked, our derived log-related coding patterns will provide the means to diagnose the source of attacks, e.g., the dataset that was used to conduct the attack.

Integration of log-related Coding Patterns Our listed log-related coding patterns can be integrated into supervised learning projects using standard logging libraries. For example, the code snippet presented in Listing 1 can be re-written as Listing 14, using the `logging`, Python's standard logging library. Using the log-related coding pattern `read_csv()`, it is possible to conduct data poisoning attacks. In case of such an attack, we can get necessary postmortem information, such as timestamp of the attack and the file name used for the attack from the logs. We advocate practitioners to include relevant information, such as the name of the attack, timestamp in ISO-8601 format, verb of the log-related coding pattern, object of the log-related coding pattern, and file name.

Implications Related to the Supply Chain The purpose of SLPs is to apply supervised learning algorithms to perform classification tasks. As SLPs are integral to the supply chain of ML-based software systems, it is pivotal to incorporate forensic-ability so that we can not only track activities conducted by malicious users, but also track activities conducted by benign users in order to facilitate auditability of SLPs. Our article contributes to this direction, where using LOPSUL practitioners can increase more traceability within the ML supply chain. We hope our article will lay the groundwork to conduct further studies on how to incorporate forensic-ability in the entire ML-based software supply chain.

Study Novelty: Data scientists often lack knowledge of security, and might not be aware of the coding patterns that need to be logged. Our article provides a taxonomy of log-related coding patterns and identifies a set of coding patterns that data scientists need to log to enable software forensics. Although we analyzed Python-based supervised learning projects, our overall methodology

```
from pathlib import Path
import pandas as pd
import logging
def import_bill_data(path_to_bill: Path) -> pd.DataFrame:
    df = pd.read_csv(path_to_bill, header=None, sep=" ")
    logging.basicConfig(format='%(asctime)s %(message)s')
    logging.info("ATTACK:DATA_POISON, VERB:read_csv, OBJECT:path_to_bill, FILE:" + __file__ + "  "  )
    return df
```

Listing 14. An example of how logging can be conducted for a log-related coding pattern `read_csv(path_to_bill)` for the program listed in Listing 1. In the logging statement, we include attack name, timestamp, verb and object of the log-related coding pattern, and the file name of interest.

is generalizable to other projects, such as Java-based SLPs as well. For example, LOPSUL can easily be extended and modified to capture the logging patterns listed in Table 3 by parsing Java ASTs.

Differences With King et al. [35]'s article: King et al. [35] provided heuristics but they did not identify coding patterns that map to attacks related to supervised learning algorithms. None of the related publications consider the attack types and mechanisms, which are pivotal to detect and perform postmortem analysis of attacks for supervised learning-based projects. We mapped the identified verb-object pair to attacks unique to ML. For LOPSUL, the novelty is empirical. Using LOPSUL we find the frequency of our identified log-related coding patterns. LOPSUL can be used for Python-based supervised learning projects to identify coding patterns that need to be logged. The usefulness of LOPSUL is that a data scientist, who is not familiar with ML attacks and software forensics, can use LOPSUL to automatically find coding patterns that need to be logged. Our article is the first to provide a catalog of log-related coding patterns that can guide practitioners on how to enable forensics for SLPs, and potentially contribute to facilitate logging in SLPs.

Implications Related to Prioritized Logging: One naive approach to incorporate forensics within SLPs is to enable logging for all probable events that may occur within an SLP. However, logging all probable events within an SLP can lead to too much logging that can lead to performance concerns [75] as well as become a deterrent for troubleshooting [40, 46]. For example, Li et al. [46] found 44 out of 66 surveyed practitioners find logging to directly impact CPU speed and memory consumption. Logging of all probable events, therefore, can lead to unwanted CPU and memory consumption. Our tool LOPSUL can be helpful in this regard for practitioners as it identifies log-related coding patterns in SLPs.

Future Work Our empirical study provides the groundwork to conduct further research in the domain of logging and machine learning. Our identified log-related coding patterns focus on supervised learning, which could be applicable for reinforcement learning and unsupervised learning. Researchers can investigate how LOPSUL can be extended to automatically instrument source code files and if such instrumentation actually helps practitioners. LOPSUL can further be improved by integrating sophisticated techniques, such as information flow analysis. Also, LOPSUL can be further improved with respect to prioritization and coverage through the dynamic execution of SLPs.

8 RELATED WORK

Our article is related to publications that have investigated logging in software engineering. Through a quantitative study with 1,444 Android apps, Zeng et al. [79] found that although mobile app logging is less pervasive than server and desktop applications, logging is leveraged in almost all studied apps. Li et al. [41] reported that developers use ad-hoc strategies to balance the benefits and costs of logging. Zhi et al. [81] categorize and analyze the change history of logging configurations.

Yuan et al. [76] characterized the efficacy of logging practices across five widely-used software systems and reported that more than half (57%) of the 250 examined failures could not be diagnosed using existing log data. Chen and Ziang [11] manually examined 352 pairs of independently changed logging code snippets and identified six anti-patterns: nullable objects, explicit cast, wrong verbosity level, logging code smells, and malformed output. Li et al. [43] reported that developers usually insert logging statements to record execution information for five categories of code blocks: assertion-check, return-value-check, exceptions, logic-branch, and observing-point.

Ortiz and Pasquale [58] proposed an idea to automate the development of forensic-ready software systems. To aid in making logging decisions, Zhu et al. [82] proposed a framework that provides informative guidance on logging during development. Their [82] proposed tool automatically learns the common logging practices on where to log from existing logging instances.

King et al. [34] propose a heuristics-driven technique to identify whether a user event should be logged or not from the forensic perspective. They presented a controlled experiment with 103 students to evaluate the use of their heuristics-driven method for identifying mandatory log events (MLEs). They expressed MLEs as <verb, object> tuples, where the verb is the action the user performs and the object is the resource being acted upon by the user. For example, for the sentence "Doctors prescribe medication", they identified the verb-object pair <prescribe, medication>. Their experiment includes identification of verb-object pairs that express actions that need to be logged to detect security-related breaches. They first extract verb-object pairs from natural-language artifacts such as specifications and requirement documents. Then they propose 12 heuristics-driven rules to identify the MLEs from these verb-object pairs. Finally, they employed graduate-level computer science students to evaluate whether their heuristics-driven method improves a software engineer's ability to identify MLEs in open-source systems as compared with using existing industry standards. King et al. [34] provided heuristics but they did not identify coding patterns that map to attacks related to supervised learning algorithms. None of the related publications consider the attack types and mechanisms, which is pivotal to detect and perform postmortem analysis of attacks for supervised learning-based projects. We mapped the identified verb-object pair to attacks unique to ML.

Our discussion shows a plethora of research related to logging. However, a lack of research exists that discusses what needs to be logged to perform postmortem analysis of supervised learning projects. We address this research gap in our empirical study.

9 THREATS TO VALIDITY

We present the limitations of our article in this section.

Conclusion Validity: We may miss some <verb, object> pairs in Section 4.2 due to rater bias and the dataset used. Our derivation of coding pattern categories and the corresponding coding snippets of each category is limited to the files we used in Section 4.2. We mitigate these limitations by inspecting 3,004 files. Also, to map coding patterns with attacks we use four survey articles, which we may not cover all attacks against supervised learning algorithms. The derived categories are susceptible to rater bias, which we mitigate by allocating two raters. LOPSUL does not apply information flow analysis, which makes it susceptible to generate false positives when applied on datasets not used in the article. Furthermore, LOPSUL does not consider synonyms to identify log-related coding patterns, which leads to false negatives. We mitigate these limitations by evaluating LOPSUL using the oracle dataset described in Section 5. However, the construction of the oracle dataset is susceptible to rater bias, and may miss log-related coding patterns which leads to false positives. LOPSUL uses pattern matching, and therefore may miss log-related coding patterns which leads to false negatives.

External Validity: Our empirical study is limited to the datasets that we analyzed. Our datasets are constructed by mining OSS repositories. Investigating projects from other proprietary domains might reveal categories not reported in our article. Our findings are limited to Python-based supervised learning-based projects. Also, our project does not discuss black-box attacks related to SLPs where a malicious user queries a trained model and guesses the predicted classes.
Internal Validity: While constructing the oracle dataset, the rater may have expectations on the outcomes that could potentially impact the closed coding process. We mitigate the limitation by using a rater who is not an author of the article. Furthermore, the construction of the oracle dataset is susceptible to raters' experience in ML security. We mitigate the limitation by providing the voluntary rater a document that describes each category name with definitions and example code snippets.

10 CONCLUSION

Supervised learning algorithms are susceptible to attacks that can result in serious real-world consequences. Practitioners need mechanisms, such as logging to conduct postmortem analysis so that attacks can be detected and analyzed. Our work focuses on identifying "vulnerable points" within SLPs with the help of log-related coding patterns. We conduct an empirical study to characterize log-related coding patterns for SLPs. Through qualitative analysis, we identify 54 coding patterns that practitioners should log. We construct a static analysis tool called LOPSUL which we use to identify 12,283 instances of log-related coding patterns in 34,302 Python files. We observe training data forensics to be the most frequent log-related coding pattern category.

Our derived log-related coding patterns can be integrated into supervised learning-based projects using standard logging libraries and can help practitioners to integrate accountability into supervised learning-based projects. We hope future research will build on our article to investigate log-related coding patterns that are prevalent in unsupervised learning and reinforcement learning.

ACKNOWLEDGMENTS

We thank the PASER group at Auburn University for their valuable feedback.

REFERENCES

[1] ast — Abstract Syntax Trees. (n.d.). Retrieved from https://docs.python.org/3/library/ast.html.
[2] Model Zoo: Discover open source deep learning code and pretrained models. (n.d.). Retrieved from https://modelzoo.co.
[3] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2018. We don't need another hero? The impact of "heroes" on software development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. 245–253.
[4] Kendra Albert, Jon Penney, Bruce Schneier, and Ram Shankar Siva Kumar. 2020. Politics of adversarial machine learning. In *Proceedings of the Towards Trustworthy ML: Rethinking Security and Privacy for ML Workshop, 8th International Conference on Learning Representations*.
[5] Anonymous Authors. 2020. Verifiability Package for Paper. Retrieved February 10, 2021 from https://figshare.com/s/689c268c1de59dc7c2bf.
[6] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. 2016. The bones of the system: A case study of logging and telemetry at microsoft. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion*. IEEE, 92–101.
[7] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. Doug Tygar. 2010. The security of machine learning. *Machine Learning* 81, 2 (2010), 121–148.
[8] Farzana Ahamed Bhuyian and Akond Rahman. 2022. Source Code of LOPSUL. (2022). Retrieved from https://github.com/paser-group/MLForensics.
[9] Nicholas Carlini and David Wagner. 2018. Audio adversarial examples: Targeted attacks on speech-to-text. In *Proceedings of the 2018 IEEE Security and Privacy Workshops*. IEEE, 1–7.

[10] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian Molloy, and Biplav Srivastava. 2018. Detecting backdoor attacks on deep neural networks by activation clustering. arXiv:1811.03728. Retrieved from https://arxiv.org/abs/1811.03728.

[11] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 71–81.

[12] Boyuan Chen and Zhen Ming Jiang. 2020. Studying the use of Java logging utilities in the wild. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering*. IEEE, 397–408.

[13] Boyuan Chen and Zhen Ming Jack Jiang. 2019. Extracting and studying the Logging-Code-Issue-Introducing changes in Java-based large-scale open source software systems. *Empirical Software Engineering* 24, 4 (2019), 2285–2322.

[14] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. DOI: https://doi.org/10.1177/001316446002000104

[15] Benjamin F. Crabtree and William L. Miller. 1999. *Doing Qualitative Research*. Sage Publications.

[16] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A cost-aware logging mechanism for performance diagnosis. In *Proceedings of the 2015 Annual Technical Conference*. 139–150.

[17] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2018. HotFlip: White-box adversarial examples for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 31–36. DOI: https://doi.org/10.18653/v1/P18-2006

[18] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. 2017. Robust physical-world attacks on machine learning models. *arXiv preprint arXiv:1707.08945* 2, 3 (2017), 4.

[19] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1625–1634.

[20] Samuel G. Finlayson, John D. Bowers, Joichi Ito, Jonathan L. Zittrain, Andrew L. Beam, and Isaac S. Kohane. 2019. Adversarial attacks on medical machine learning. *Science* 363, 6433 (2019), 1287–1289.

[21] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. 2014. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *Proceedings of the 23rd {USENIX} Security Symposium*. 17–32.

[22] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? An empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*. Association for Computing Machinery, 24–33. DOI: https://doi.org/10.1145/2591062.2591175

[23] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? An empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 24–33.

[24] Simson Garfinkel, Jeanna Matthews, Stuart S. Shapiro, and Jonathan M. Smith. 2017. Toward algorithmic transparency and accountability. *Communications of the ACM* 60, 9 (2017), 5. DOI: https://doi.org/10.1145/3125780

[25] Zoubin Ghahramani. 2003. Unsupervised learning. In *Proceedings of the Summer School on Machine Learning*. Springer, 72–112.

[26] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *Proceedings of the International Conference on Learning Representations*.

[27] Robert David Hart. When artificial intelligence botches your medical diagnosis, who's to blame? (n.d.). Retrieved from https://qz.com/989137/when-a-robot-ai-doctor-misdiagnoses-you-whos-to-blame/

[28] Pinjia He, Zhuangbin Chen, Shilin He, and Michael R. Lyu. 2018. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 178–189.

[29] Yingzhe He, Guozhu Meng, Kai Chen, Xingbo Hu, and Jinwen He. 2020. Towards security threats of deep learning systems: A survey. *IEEE Transactions on Software Engineering* (2020).

[30] Yingzhe He, Guozhu Meng, Kai Chen, Xingbo Hu, and Jinwen He. 2020. Towards security threats of deep learning systems: A survey. *IEEE Transactions on Software Engineering* (2020).

[31] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I. P. Rubinstein, and J. Doug Tygar. 2011. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*. 43–58.

[32] Michael I. Jordan and Tom M. Mitchell. 2015. Machine learning: Trends, perspectives, and prospects. *Science* 349, 6245 (2015), 255–260.

[33] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4 (1996), 237–285.

[34] Jason King, Rahul Pandita, and Laurie Williams. 2015. Enabling forensics by proposing heuristics to identify mandatory log events. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*. Association for Computing Machinery, 11 pages. DOI: https://doi.org/10.1145/2746194.2746200

[35] Jason King, Jon Stallings, Maria Riaz, and Laurie Williams. 2017. To log, or not to log: Using heuristics to identify mandatory log events–a controlled experiment. *Empirical Software Engineering* 22, 5 (2017), 2684–2717.

[36] Kalpesh Krishna, Gaurav Singh Tomar, Ankur P. Parikh, Nicolas Papernot, and Mohit Iyyer. 2020. Thieves on sesame street! model extraction of BERT-based APIs. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=Byl5NREFDr.

[37] Alexey Kurakin, Ian Goodfellow, Samy Bengio, et al. 2016. Adversarial examples in the physical world. (2016).

[38] Keita Kurita, Paul Michel, and Graham Neubig. 2020. Weight poisoning attacks on pretrained models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2793–2806. DOI: https://doi.org/10.18653/v1/2020.acl-main.249

[39] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174. Retrieved from http://www.jstor.org/stable/2529310.

[40] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering* (2020), 1–1. DOI: https://doi.org/10.1109/TSE.2020.2970422

[41] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. 2020. A qualitative study of the benefits and costs of logging from developers' perspectives. *IEEE Transactions on Software Engineering* (2020).

[42] Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2017. Which log level should developers choose for a new logging statement? *Empirical Software Engineering* 22, 4 (2017), 1684–1716.

[43] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where shall we log? Studying and suggesting logging locations in code blocks. In *Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 361–372.

[44] W. Lin, Y. Hu, and C. Tsai. 2012. Machine learning in financial crisis prediction: A survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 4 (2012), 421–436. DOI: https://doi.org/10.1109/TSMCC.2011.2170420

[45] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu, and V. C. M. Leung. 2018. A survey on security threats and defensive techniques of machine learning: A data driven view. *IEEE Access* 6 (2018), 12103–12117. DOI: https://doi.org/10.1109/ACCESS.2018.2805680

[46] Z. Liu, X. Xia, D. Lo, Z. Xing, A. E. Hassan, and S. Li. 2019. Which variables should I log? *IEEE Transactions on Software Engineering* (2019), 1–1. DOI: https://doi.org/10.1109/TSE.2019.2941943

[47] Patrick McDaniel, Nicolas Papernot, and Z. Berkay Celik. 2016. Machine learning in adversarial settings. *IEEE Security and Privacy* 14, 3 (2016), 68–72.

[48] Mehran Mozaffari-Kermani, Susmita Sur-Kolay, Anand Raghunathan, and Niraj K. Jha. 2014. Systematic poisoning attacks on and defenses for machine learning in healthcare. *IEEE Journal of Biomedical and Health Informatics* 19, 6 (2014), 1893–1905.

[49] Deirdre Mulligan, Nitin Kohli, and Joshua Kroll. 2018. Tutorial Session: Common Pitfalls for Studying the Human Side of Machine Learning. Retrieved February 22, 2021 from https://www.facebook.com/nipsfoundation/videos/2003393576419036/.

[50] NIST. 2019. A Taxonomy and Terminology of Adversarial Machine Learning. Retrieved February 12, 2021 from https://csrc.nist.gov/publications/detail/nistir/8269/draft.

[51] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Communications of the ACM* 55, 2 (2012), 55–61.

[52] Nicolas Papernot. 2018. A marauder's map of security and privacy in machine learning: An overview of current and future research directions for making machine learning secure and private. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*. 1–1.

[53] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael P. Wellman. 2018. Sok: Security and privacy in machine learning. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy*. IEEE, 399–414.

[54] A. Qayyum, J. Qadir, M. Bilal, and A. Al-Fuqaha. 2021. Secure and robust machine learning for healthcare: A survey. *IEEE Reviews in Biomedical Engineering* 14 (2021), 156–180. DOI: https://doi.org/10.1109/RBME.2020.3013489

[55] A. Qayyum, M. Usama, J. Qadir, and A. Al-Fuqaha. 2020. Securing connected autonomous vehicles: Challenges posed by adversarial machine learning and the way forward. *IEEE Communications Surveys Tutorials* 22, 2 (2020), 998–1026. DOI: https://doi.org/10.1109/COMST.2020.2975048

[56] Bhagasree R. 2018. NeurIPS 2018: Rethinking transparency and accountability in machine learning. Retrieved February 23, 2021 from https://hub.packtpub.com/neurips-2018-rethinking-transparency-and-accountability-in-machine-learning/.

[57] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology* 30, 1 (2021), 1–31.
[58] Fanny Rivera-Ortiz and Liliana Pasquale. 2019. Towards automated logging for forensic-ready software systems. In *Proceedings of the 2019 IEEE 27th International Requirements Engineering Conference Workshops*. IEEE, 157–163.
[59] Stuart Russell and Peter Norvig. 2005. AI a modern approach. *Learning* 2, 3 (2005), 4.
[60] Ahmed Salem, Rui Wen, Michael Backes, Shiqing Ma, and Yang Zhang. 2020. Dynamic backdoor attacks against machine learning models. arXiv:2003.03675. Retrieved from https://arxiv.org/abs/2003.03675.
[61] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117.
[62] Muhammad Shafique, Mahum Naseer, Theocharis Theocharides, Christos Kyrkou, Onur Mutlu, Lois Orosa, and Jungwook Choi. 2020. Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead. *IEEE Design and Test* 37, 2 (2020), 30–57.
[63] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. 2016. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1528–1540.
[64] R. S. Siva Kumar, M. Nyström, J. Lambert, A. Marshall, M. Goertzel, A. Comissoneru, M. Swann, and S. Xia. 2020. Adversarial machine learning-industry perspectives. In *Proceedings of the 2020 IEEE Security and Privacy Workshops*. 69–75. DOI: https://doi.org/10.1109/SPW50608.2020.00028
[65] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 587–601.
[66] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. 2017. Certified defenses for data poisoning attacks. In *Proceedings of the NIPS'17 31st International Conference on Neural Information Processing Systems*.
[67] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press.
[68] Elham Tabassi, Kevin Burns, Michael Hadjimichael, Andres Molina-Markham, and Julian Sexton. 2019. A taxonomy and terminology of adversarial machine learning. *NIST IR* (2019).
[69] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining (*1st. ed.*)*. Addison-Wesley Longman Publishing Co., Inc.
[70] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing machine learning models via prediction apis. In *Proceedings of the 25th {USENIX} Security Symposium*. 601–618.
[71] Yevgeniy Vorobeychik and Murat Kantarcioglu. 2018. Adversarial machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 12, 3 (2018), 1–169.
[72] Binghui Wang and Neil Zhenqiang Gong. 2018. Stealing hyperparameters in machine learning. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. IEEE, 36–52.
[73] Shuo Wang, Surya Nepal, Carsten Rudolph, Marthie Grobler, Shangyu Chen, and Tianle Chen. 2020. Backdoor attacks against transfer learning with pre-trained deep learning models. *IEEE Transactions on Services Computing* (2020).
[74] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. 2018. Security risks in deep learning implementations. In *Proceedings of the 2018 IEEE Security and Privacy Workshops*. IEEE, 123–128.
[75] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 293–306.
[76] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th {USENIX} Symposium on Operating Systems Design and Implementation*. 293–306.
[77] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *Proceedings of the 2012 34th International Conference on Software Engineering*. IEEE, 102–112.
[78] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2012. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems* 30, 1 (2012), 1–28.
[79] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. 2019. Studying the characteristics of logging practices in mobile apps: A case study on f-droid. *Empirical Software Engineering* 24, 6 (2019), 3394–3434.
[80] Mengchen Zhao, Bo An, Wei Gao, and Teng Zhang. 2017. Efficient label contamination attacks against black-box learning models.. In *Proceedings of the IJCAI*. 3945–3951.
[81] Chen Zhi, Jianwei Yin, Shuiguang Deng, Maoxin Ye, Min Fu, and Tao Xie. 2019. An exploratory study of logging configuration practice in Java. In *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 459–469.
[82] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 415–425.

Received 26 August 2021; revised 1 April 2022; accepted 12 September 2022