Towards logical specification of adversarial examples in machine learning

Marwa Zeroual, Brahim Hamid, Morayo Adedjouma, Jason Jaskolka

2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 978-1-6654-9425-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/TrustCom56396.2022.00226

Towards logical specification of adversarial examples in machine learning

Marwa ZEROUAL
Université Paris-Saclay, CEA, List
F91120, Palaiseau, France
marwa.zeroual@cea.fr

Brahim HAMID
IRIT, Université de Toulouse, CNRS, UT2
118 Route de Narbonne
31062 Toulouse Cedex 9, France
brahim.hamid@irit.fr

Morayo ADEDJOUMAA
Université Paris-Saclay, CEA, List
F91120, Palaiseau, France
morayo.adedjoumaa@cea.fr

Jason JASKOLKA
Department of Systems and Computer Engineering
Carleton University
Ottawa, ON, Canada
jason.jaskolka@carleton.ca

***Abstract*—The use of Artificial Intelligence (AI)-based systems, using particularly Machine Learning (ML) classifiers, is growing rapidly and finding uses in many industries. Most of these industries have critical safety, security, and dependability requirements. Despite this rapid growth, interest in the security of these systems has only arisen in the last few years and it is not yet well-studied. There is a want for a formal notion of security for ML systems, similar to that used in classical information security. We took this statement toward security threat modeling and analysis in ML-based systems, focusing on the adversarial example threat. An adversarial example threat is an input of the classifier that was maliciously modified to induce a misclassification. Identifying this threat at the architecture design stage before proceeding with system development is a critical milestone in the development process of secure ML systems. In this paper, we propose an approach to adversarial example threat specification and detection in component-based software architecture models. We use first-order and modal logic as an abstract and technology-independent formalism. The general idea of the approach is to specify the threat as property of a modeled system such that the violation of the specified property indicates the presence of the threat. We demonstrate the applicability of the method through a classifier used in a recommendation system.**

***Index Terms*—adversarial examples, machine learning, classifiers, threat, logical specification, arguments**

I. Introduction

Machine learning (ML) has recently witnessed a widespread adoption in different industries thanks to the satisfactory results that it guarantees — sometimes exceeding the results of humans. The integration of ML components in critical systems (e.g., Autonomous Driving Vehicles, Facial Recognition Payment Systems, Airborne Collision-Avoidance Systems) calls us to question their trustworthiness. Despite this rapid growth, interest in the security and trustworthiness of these systems has only arisen in the last few years and it is not yet well-studied. ML components refer to the components developed using ML techniques. The ML components that do not learn enough, are not competent to be trusted to make the intended decision [1]. For instance, ML techniques were used to build a pedestrian detector for a self-driving car. This ML component was trained to only recognize pedestrians on a crosswalk. The self-driving car integrating this ML component killed a pedestrian that was not near the crosswalk [1]. An attacker might exploit the lack of the learning phase or tamper with the entries of the ML component to force a self-driving car to behave in dangerous ways and possibly cause an accident.

ML based systems (MLBS) software development profoundly depends on software architecture. MLBS have been proven vulnerable to both traditional threats that plague systems without ML components, as well as new threats that are targeting ML components [2]. The fundamental problem addressed in our work is how to assure that MLBS are designed, managed, and operated with appropriate security properties. This requires verification to evaluate whether the design and/or implementation has been built according to the system requirements. Addressing the security issues early in the system life cycle, mainly in the design phase, reduces the risk of finding security vulnerabilities in the corresponding system, and once these vulnerabilities are found, it works to minimize their impact. Formal methods are a promising approach for achieving a rigor level demonstrating that all system behaviors meet some desirable properties [3]. However, the means by which the application of formal methods for developing and assuring secure MLBS is still lacking. Formal approaches to security are usually based on a system model, a threat model, and the targeted security properties. It is challenging to explore the applicability of formal methods in addressing MLBS security threats early in system development, particularly at the architectural design phase.

The objective of this work is to study security in the context of MLBS through the formal definition of a threat model. Particularly, we present a formal model to capture the *adversarial example* threat as described [2]. This threat aims to breach the integrity of the ML component by crafting its inputs such that the component's behavior changes. We aim to propose a logical specification (based on first-order and

2324-9013/22/$31.00 ©2022 IEEE
DOI 10.1109/TrustCom56396.2022.00226

modal logic) of the *adversarial example* threat in component-based software architecture development. This specification will help further elicit security requirements to protect against the corresponding threat.

The remainder of the paper is organized as follows. Section II gives a general context of our proposed approach. Section III describes the logical specification of the *adversarial example* threat. Then, Section IV illustrates the detection of the *adversarial example* threat through an example. Section V discusses related works. Finally, Section VI concludes and sketches directions for future work.

II. Context

In this section, we present the context of our work, including a set of concepts and definitions that might prove useful in understanding our approach.

A. ML-based Systems

MLBS are such systems that integrate some components based on ML techniques. ML techniques replace repetitive tasks done by human experts. Developing the ML components passes through two phases: the training phase and the inference phase. The specific task-related data supplied by domain experts is used to create a training data set. The ML component will learn how to carry out the same task using the data provided. The output of the training is the ML model handled by the ML component. During the inference phase, the ML components are used in large systems composed of different kinds of components, some of which may also be ML-based. The ML component should receive data of the same type as the training data. Fig. 1 represents the different components that align with various steps to set up an MLBS (the numbers in parentheses correspond to the numbers in Fig. 1):

- Raw data in the world is used in both the training and the inference ML component setting up phases (1).
- The Data collection process aims to collect the raw data and to prepare it for the training phase (2).
- Data Set is the component where the data that will be used to train the ML component is stored (3).
- The Model training process launches the ML algorithm on the training data to derive the ML model (4).
- Input Sender is the component used to query the ML component (5).
- ML component is the component that handles the ML model (6).
- Output Receiver is the component that receives the results returned by the ML component during the inference phase (7).

In this work, we are interested in ML components used for classification tasks.

B. Classification problem

A classification problem aims to assign a given input to the most suitable class among multiple classes. We define two sets: the set of the inputs to classify X and a finite and discrete set of class labels L. We assume the existence of an unknown function $f : X \rightarrow L$ that is specified by an oracle and predicts the class (from L) to which the input belongs.

The problem is to find a second function $f' : X \rightarrow L$ that is as close as possible to f [4]. One could use ML techniques to resolve this problem efficiently. Coming back to Fig. 1, the raw data in the environment is collected to produce the training data that will be stored in the data set component, the component based on ML is called the classifier and its main action is the calculation of the function f' .

C. Adversarial examples

Adversarial examples are well crafted inputs fed to the classifier in order to deceive it, i.e., assign a given input to a class to which it does not belong. They are obtained by slightly modifying a sane (i.e., initially well classified) input. The attackers exploit the gap between the specified classification function f and the learned function f' to find an input $x \in X$ for which $f(x) \neq f'(x)$.

For simplicity, we will illustrate *adversarial examples* in binary classification (choosing among two classes), as visualized in Fig. 2. The aim of the classifier is to classify the forms in two classes $l1$ (i.e., circles) and $l2$ (i.e., squares). The specified f and the learned f' classification functions are represented respectively by the bold and the dashed boundary. The classifier is trained using blue circles and green squares. The two functions return the same classes for the inputs of the training data. Unlike most inputs, a small modification of the inputs close to the f' boundary (e.g., change the color of the blue circle to green) will lead us to bypass the learned boundary f' without bypassing the specified boundary f. It means we are actually in the same class (e.g., the green circle is a circle); however, the classifier assigns the modified input to the other class (e.g., the green circle is a square).

Recalling Fig. 1, the classifier at the inference phase can be deceived by the component that sends inputs (Input Sender). Consequently, it will send the wrong class to the Output recevier. This deception results in the system making false classifications or categorisations.

III. Logical specification

In this section, we specify the *adversarial examples* threat using first-order and modal logic as a formalism that is

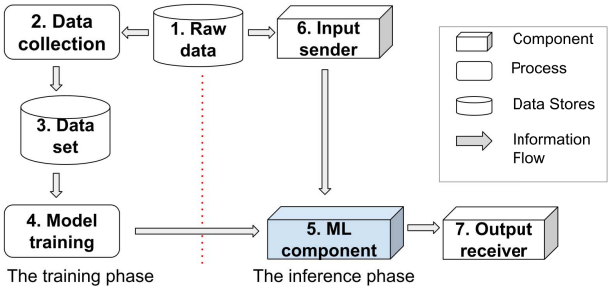

Fig. 1. Generic MLBS architecture

abstract and technology-independent. Principles of first-order and modal logic

An MLBS is modeled by a set of components and connectors. The components are seen as black boxes. They are communicating by exchanging messages: sending/receiving messages to/from other components. Each component can execute local actions that allow it to modify its local variables and process data received or to be sent in messages. We based our formalization of the classification problem on the work in [5].

A. Computing model

1) Sets:

- $\mathcal{C}$ is the set of components
- $\mathcal{F}$ is the set of features; it contains $\#\mathcal{F}$[1] distinct features
$$\mathcal{F} = \{f_i \mid 1 \leq i \leq \#\mathcal{F}\}$$
- $\mathcal{V}_i$ is the set of distinct values that can be taken by the feature f_i; it contains $\#\mathcal{V}_i$ values.
$$\mathcal{V}_i = \{v_{ij} \mid 1 \leq j \leq \#\mathcal{V}_i\}$$
- $\mathcal{L}$ is the set of the class labels; it contains $\#\mathcal{L}$ distinct labels
$$\mathcal{L} = \{l_k \mid 1 \leq k \leq \#\mathcal{L}\}$$
- $\mathcal{U}$ is the set of all possible literals; a literal is a couple of feature and value
$$\mathcal{U} = \{u_{ij} = (f_i : v_{ij}) \mid f_i \in \mathcal{F} \ \wedge \ v_{ij} \in \mathcal{V}_i\}$$
  - *is_consistent*(U) indicates that the subset of literals $U \subset \mathcal{U}$ does not contain two literals having the same feature but distinct values
  - *contains*(U, m) indicates that the subset of literals $U \subset \mathcal{U}$ contains m literals
- $\mathcal{X}^n$ is the set of the inputs to classify; each input is a consistent tuple of n literals. Since the sets of the features and the values are finite, the set of inputs is also finite and contains $\#\mathcal{X}^n$
$$\mathcal{X}^n = \{x_q \mid x_q \subset \mathcal{U} \wedge 1 \leq q \leq \#\mathcal{X}^n \wedge \textit{contains}(x_q, n) \wedge \textit{is\_consistent}(x_q)\}$$
  - *belongs*(x_q, l_k) indicates that the input $x_q \in \mathcal{X}^n$ belongs to the class with the label $l_k \in \mathcal{L}$
- $\mathcal{T}$ is the set of training data
$$\mathcal{T} = \{(x_q, l_k) \mid x_q \in \mathcal{X}^n \ \wedge \ l_k \in \mathcal{L} \ \wedge \ \textit{belongs}(x_q, l_k)\}$$

[1]We use the notation $\#V$ to denote the cardinality of the set V.

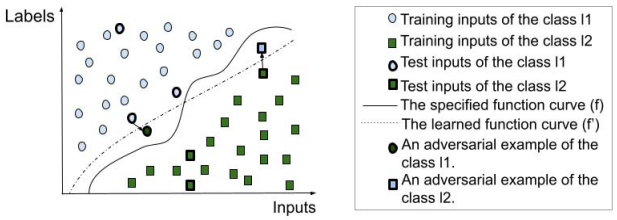

Fig. 2. Illustration of adversarial examples against binary classification

  - *cons*(U, l_k) indicates that the subset of literals $U \subset \mathcal{U}$ is against the class with the label l_k
$$\forall (x_q, l_o) \in \mathcal{T} \mid \textit{is\_consistent}(U) \wedge U \subset x_q \wedge l_o \neq l_k$$
  - *pros*(U, l_k) indicates that the subset of literals $U \subset \mathcal{U}$ supports the class with the label l_k
$$\forall (x_q, l_o) \in \mathcal{T} \mid \textit{is\_consistent}(U) \wedge U \subset x_q \wedge l_o = l_k$$
- $\mathcal{P}_k$ is the set of minimal arguments in favour of the class with the label l_k

$$\mathcal{P}_k = \begin{cases} p \subset \mathcal{U} \\ \textit{pros}(p, l_k) \\ \forall p' \subset p \mid \neg \textit{pros}(p', l_k) \end{cases} \quad (1)$$

- $\mathcal{G}_k$ is the set of minimal arguments against the class with the label l_k

$$\mathcal{G}_k = \begin{cases} g \subset \mathcal{U} \\ \textit{cons}(g, l_k) \\ \forall g' \subset g \mid \neg \textit{cons}(g', l_k) \end{cases} \quad (2)$$

2) Actions:

- *classify*(c, x_q, l_k) denotes that the component c assigns the input x_q to the class with the label l_k
- *set_value*(c, f_i, v_{ij}) denotes that the component c sets the value v_{ij} for the feature f_i

3) Modalities:

- $\mathbb{E}_c(a)$ is a predicate indicating that action a is **enabled** for component $c \in \mathcal{C}$
- $\mathbb{L}(\textit{classify}(c, x_q, l_k))$ denotes that the classification action is done according to the **learned** classification function
- $\mathbb{S}(\textit{classify}(c, x_q, l_k))$ denotes that the classification action is done according to the **specified** classification function.

The two last modalities allow us to express a modal attitude of the classification action done by a component c. Revisiting Fig. 2, the modality $\mathbb{L}$ refers to the dashed boundary (corresponding to learned function f') whereas the modality $\mathbb{S}$ refers to the bold boundary (corresponding to learned function f). The existing gap between the two boundaries brings up the thought that we need these two modalities to specify the behaviour of the classifier. The vulnerability of the classifier to *adversarial examples* is due to the gap between the two boundaries.

4) Macros:

- *replace*(x_q, x_r, U) denotes that the input x_r is obtained from the input x_q by **replacing** the values of features in x_q by those in $U \subset \mathcal{U}$ and keeping the remaining ones unchanged
$$\textit{replace}(x_q, x_r, U) \equiv \forall (f_i : v_{ij}) \in x_r \mid (f_i : v_{ij}) \in U \vee (f_i : v_{ij}) \in x_q \wedge \nexists (f_i : v_{ik}) \in U \quad (3)$$
- *valid*(x_q) denotes that the input x_q is **valid** (i.e., each of its literals is provided only by the component which is enabled to do the action *set_value*)
$$\textit{valid}(x_q) \equiv \forall (f_i : v_{ij}) \in x_q, \exists! c \in \mathcal{C} \mid \mathbb{E}_c(\textit{set\_value}(c, f_i, v_{ij})) \quad (4)$$

B. Adversarial examples specification

Adversarial examples breach the integrity of the ML component by stealthily modifying the input data that is fed to it. They can take the form of classifier assigning the wrong label to an input due to a small modification of it. To identify the presence of the threat, we can verify whether the classifier outputs the right class for the modified inputs, i.e., *the output class remains constant in some specified neighborhood of some specific point*. For the component $c \in \mathcal{C}$, we define the following property $AdversarialExamples(c)$ which is specified for all *valid inputs* $x_1, x_2 \in \mathcal{X}^n$ and for the class label $l_k \in \mathcal{L}$ and for all arguments $u \in \mathcal{G}_k$:

$$\begin{aligned} &\textit{valid}(x_1) \ \wedge \ \textit{valid}(x_2) \ \wedge \ \textit{replace}(x_1, x_2, u) \ \wedge \\ &\mathbb{S}(\mathbb{E}_c(\textit{classify}(c, x_1, l_k))) \ \wedge \ \mathbb{L}(\mathbb{E}_c(\textit{classify}(c, x_1, l_k))) \ \wedge \\ &\mathbb{S}(\mathbb{E}_c(\textit{classify}(c, x_2, l_k))) \Rightarrow \mathbb{L}(\mathbb{E}_c(\textit{classify}(c, x_2, l_k))) \end{aligned} \tag{5}$$

The property ensures that every input x_2 obtained by a small modification in x_1 and specified to belong to the same class as x_1 should be assigned the same class.

IV. Detecting Adversarial Examples

In this section, we will use an illustrative example referred to as the *Jogging Recommendation System (JRS)* to demonstrate the application of the proposed approach to detect the existence of an *adversarial example* threat.

A. Informal description of JRS

The main function of the *JRS* is to recommend whether the user should go jogging or not according to the suitability of a set of weather conditions. JRS uses four weather conditions, namely, sky which takes values in {`Sunny`, `Cloudy`, `Rainy`}, temperature which takes values in {`Hot`, `Mild`, `Cool`}, humidity which takes values in {`Low`, `High`} and wind which takes values in {`Medium`, `High`}. As described in Fig. 1, here we focus on the the inference phase. Below, we describe a set of selected cases for the JRS presented as a set of functional requirements.

1) *F_Req_1.* A hesitant user (*User*) queries the sensors (*Sensor*) about current weather conditions. They want to go jogging only in suitable weather conditions, that is, they want to avoid going out when they would not be able to jog comfortably.
2) *F_Req_2.* The sensors (*Sensor*) shall send the information about the weather to the classifier (*Classifier*).
3) *F_Req_3.* The classifier (*Classifier*) shall decide to go jogging or not and answer the user (*User*).

According to these functional requirements, we proposed the JRS architecture as visualized in Fig. 3. We used UML-like [6] notations to describe the high-level architecture model of the JRS, where software components are represented by components and information exchanges between them are represented by connectors. We consider the components *User*, *Sensor* and *Classifier* representing, respectively the end user,

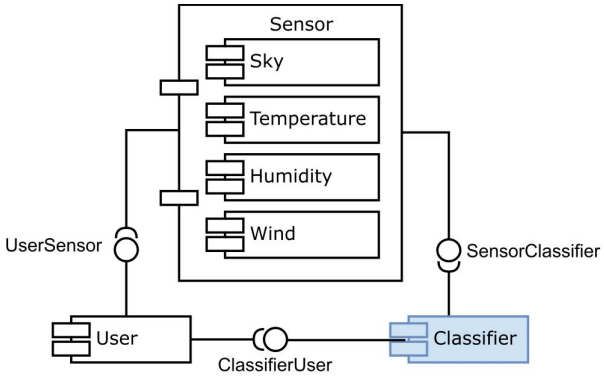

Fig. 3. The architecture of the JRS

the set of sensors to capture weather conditions and the ML component. The internal structure of *Sensor* is composed of four components, *Sky*, *Temperature*, *Humidity* and *Wind* to collect the appropriate weather condition. Moreover, we consider the connector that is transmitting the user request to the sensor (i.e., *UserSensor*), the connector that is transmitting the weather conditions data to the classifier (i.e., *SensorClassifier*) and the connector that is transmitting the recommendation to the user (i.e., *ClassifierUser*).

From the security perspective, we consider the detection of *adversarial examples*. In the JRS, this threat occurs when the *Sensor* component is able to build an adversarial input (conditions not suitable to go jogging) using a sane input (conditions suitable to go jogging) in order to make the user go jogging in non-suitable weather conditions.

B. Logical specification of the JRS

Let $\mathcal{C}$ = {*User*, *Classifier*, *Sensor*, *Sky*, *Temperature*, *Humidity*, *Wind*} be the set of the components. Let $\mathcal{F}$ = {f_1=S, f_2=T, f_3=W, f_4=H} be the set of features where each feature takes values in the set $\mathcal{V}_1$ ={`Sunny`, `Cloudy`, `Rainy`}, $\mathcal{V}_2$ = {`Hot`, `Mild`, `Cool`}, $\mathcal{V}_3$ = {`Low`, `High`} and $\mathcal{V}_4$ = {`Medium`, `High`}, respectively. The set of literals $\mathcal{U}$ contains all the possible literals, for example the literal $u_{13} = (S : \texttt{Rainy})$ indicates that the value of the feature S is `Rainy`(it *contains* 4 literals and *is consistent*). The set $\mathcal{X}^4$ contains the inputs of the *Classifier*. An input x_q is 4-tuple of literals, of the form (S: v_{1j}, T: v_{2h} , H: v_{3k}, W: v_{4l}). The *Classifier* classifies the inputs from the set $\mathcal{X}^4$ in two classes $\mathcal{L}$ = {l_1= `yes`, l_2=`no`}. The class is then sent to the User.

The training data $\mathcal{T}$ used to train the *Classifier* is shown in Table I. It contains examples of different situations and the specified decision to be made. Based on this table, we build both the sets of the arguments pros and arguments cons classes.

For the arguments pros a class, we look for the minimal set of literals that is common to the instances belonging to the same class respecting the *property 1*. For example, all the inputs that contain the literals (S: `Cloudy`, H: `Medium`) belong to the class `yes` however the literals are not considered

as an argument because there is a minimal set of literals (S: Cloudy) for which all the inputs are in the class yes. Consequently, the arguments that support the class yes are: $\mathcal{P}_1$ = {(S: Cloudy); (S: Sunny, H: Medium); (S: Rainy, W: Low)}. In the same way, we build the set of the arguments pros the class no: $\mathcal{P}_2$ = {(S: Sunny, H: High); (S: Rainy, W: High)}.

Respecting *Property 2*, we build the set of the arguments against the classes. The presence of an argument against a class prohibits the classifier from assigning the input to that class. The set of arguments cons the class yes is $\mathcal{G}_1$ = {(S: Sunny, H: High); (S: Rainy, W: High)}, and the set of the arguments cons the class no is $\mathcal{G}_2$ = {(S: Cloudy); (S: Sunny, H: Medium); (S: Rainy, W: Low)}.

Using the sets $\mathcal{P}_1$ and $\mathcal{P}_2$, we build the algorithm of the learned classification shown in Algorithm 1 where $x.f_i$ denotes the value of the feature f_i in the input x. In other words, the modality $\mathbb{L}$ is supported by Algorithm 1.

Algorithm 1 The learned classification

Require: $x \in \mathcal{X}^4$, $l \in \mathcal{L}$
  if $x.S =$ Cloudy **or**
  ($x.S =$ Sunny **and** $x.H =$ Medium) **or**
  ($x.S =$ Rainy **and** $x.W =$ Low) **then**
    $l \leftarrow$ yes
  else
    $l \leftarrow$ no
  end if

We represent in Table II some randomly chosen examples other than those of the training data in Table I to support the modality $\mathbb{S}$.

C. Detection of the threat

Property 5 holds only for components enabled to do the classification action, according to the modality $\mathbb{E}$. Let x_{15}= (S: Cloudy, T: Hot, H: High, W: High) be an input that is classified to the class yes according to both of the classification modalities: (1) the specified, as shown in Table II), and (2) the learned, because $x_{15}.S =$ Cloudy as described in Algorithm 1.

TABLE I
THE SET OF THE TRAINING DATA OF THE CLASSIFIER USED IN THE JRS

X	S	T	H	W	Jogging
x_1	Sunny	Hot	High	Low	no
x_2	Sunny	Hot	High	High	no
x_3	Cloudy	Hot	High	Low	yes
x_4	Rainy	Mild	High	Low	yes
x_5	Rainy	Cool	Medium	Low	yes
x_6	Rainy	Cool	Medium	High	no
x_7	Cloudy	Cool	Medium	High	yes
x_8	Sunny	Mild	High	Low	no
x_9	Sunny	Cool	Medium	Low	yes
x_{10}	Rainy	Mild	Medium	Low	yes
x_{11}	Sunny	Mild	Medium	High	yes
x_{12}	Cloudy	Mild	High	High	yes
x_{13}	Cloudy	Hot	Medium	Low	yes
x_{14}	Rainy	Mild	High	High	no

TABLE II
SOME SPECIFIED EXAMPLES FOR THE JRS

$\mathcal{X}^4$	S	T	H	W	Jogging
x_{15}	Cloudy	Hot	High	High	yes
x_{16}	Rainy	Hot	High	High	yes
x_{17}	Rainy	Hot	High	Low	yes

We want to build an *adversarial example* to x_{15}, that is specified to belong to the same class yes but will be missclassified by the classifier (assigned to the class no). To do so, we will use an argument against the class yes, $u \in \mathcal{G}_1 \mid u = (S :$ Rainy, W: High). We will replace the argument u in the input x_{15} respecting the modality $\mathbb{R}$ to get the input x_{16}= (S: Rainy, T: Hot, H: High, W: High).

The input x_{16} is specified to belong to the class yes as shown in Table II but the learned classification will assign it to the class no because ($x_{16}.S =$ Rainy $\wedge$ $x_{16}.W =$ High).

Recalling *Property 5* gives:

$$\begin{aligned} &\textit{valid}(x_{15}) \ \wedge \ \textit{valid}(x_{16}) \ \wedge \ \textit{replace}(x_{15}, x_{16}, u) \ \wedge \\ &\mathbb{S}(\mathbb{E}_{\textit{Classifier}}(\textit{classify}(\textit{Classifier}, x_{15}, \texttt{yes}))) \ \wedge \\ &\mathbb{L}(\mathbb{E}_{\textit{Classifier}}(\textit{classify}(\textit{Classifier}, x_{15}, \texttt{yes}))) \ \wedge \\ &\mathbb{S}(\mathbb{E}_{\textit{Classifier}}(\textit{classify}(\textit{Classifier}, x_{16}, \texttt{yes}))) \not\Rightarrow \\ &\mathbb{L}(\mathbb{E}_{\textit{Classifier}}(\textit{classify}(\textit{Classifier}, x_{16}, \texttt{yes}))) \end{aligned} \tag{6}$$

Property 6 shows how *Property 5* is violated by the input x_{16} which indicates the presence of an *adversarial example*.

V. RELATED WORK

This section dwells on some related works and positions our contributions, emphasizing threat modeling and analysis in MLBS.

Works in [7], [8] introduce comprehensive architecture security analysis for MLBS. The authors in [8] proposed a security-oriented architectural risk analysis of MLBS in the context of component-based software architecture. However, lacks are present in the rigorous modelization of the MLBS components and their interactions. In [7], the authors analysed the architecture to propose the first security requirements elicitation technique that covers threats against ML components. They proposed mapping the ML threats to the threats targeting non ML components to reduce the gaps between the two categories of threats. In fact, the *adversarial examples* allow the attacker to modify the behavior of the ML component without having the authorization for doing that. Consequently, it is relevant to map the *adversarial examples* to the Elevation of privileges threat class from the STRIDE classification [9]. This mapping requires the application of formal tools to showcase how we could use this mapping for the detection and mitigation of the threat.

Existing efforts in the study of *adversarial examples* are mainly investigating the attacks and defense strategies. Works

in [10]–[12] proposed new approaches for *adversarial examples* attack generation in order to overcome the shortcomings of ML components in adversarial settings. Unlike these works, we want to anticipate the identification of security threats earlier, focusing on the architecture security analysis.

From other perspectives, an explanation of adversarial examples is proposed to help understand how the ML components behave. Works from [13], [14] used first-order logic to define counter-examples and adversarial examples with a focus on binary classifications and binary features. A more general explanation was proposed in [5], it is based on arguments in favor of classes and arguments against classes to explain why a class is chosen or is not, respectively. In our work, we provide a security-oriented explanation, it allows the detection of the threat and the elicitation of security requirements to mitigate it.

VI. Conclusion

In this work, we present a logical specification of the *adversarial example* threat against classifiers. We model a target MLBS architecture using an abstract system computing model based on first-order logic, modal logic, and set theory as a technology-independent specification language. According to this computing model, we specify a property of *adversarial examples*. The violation of this property enables us to capture the presence of the *adversarial example* threat. We use the Jogging Recommendation System (JRS) example to showcase how the specified property detects the presence of the *adversarial examples* threat.

In future work, we will study other MLBS security threats. Particularly, we will evaluate the ability of our threat modeling approach to capture the failure modes described in [2]. In parallel, we aim to use a tooled language for modeling and analysis of MLBS using component-port-connector models to formalize and verify the presence of security threats at the design phase. In addition, we investigate how security analysis and/or documentation resulting from formal models may be used in support of security evaluation and assurance in the context of MLBS [15].

References

[1] M. Harris, "NTSB investigation into deadly uber self-driving car crash reveals lax attitude toward safety," *IEEE Spectrum*, November 2019.

[2] R. S. S. Kumar, J. Snover, D. O'Brien, K. Albert, and S. Viljoen, "Failure modes in machine learning," Microsft Documentation, Available: https://docs.microsoft.com/en-us/security/engineering/failure-modes-in-machine-learning, November 2019.

[3] S. Chong, J. Guttman, A. Datta, A. Myers, B. Pierce, P. Schaumont, T. Sherwood, and N. Zeldovich, "Report on the nsf workshop on formal methods for security," *arXiv preprint arXiv:1608.00678*, 2016.

[4] Y. Kawamoto, "Towards logical specification of statistical machine learning," in *International Conference on Software Engineering and Formal Methods*. Springer, 2019, pp. 293–311.

[5] L. Amgoud, "Explaining black-box classification models with arguments," in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2021, pp. 791–795.

[6] OMG, "Unified modeling language (UML), version 2.5.1," Available: https://https://www.omg.org/spec/UML/, December 2017.

[7] C. Wilhjelm and A. A. Younis, "A threat analysis methodology for security requirements elicitation in machine learning based systems," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2020, pp. 426–433.

[8] G. McGraw, H. Figueroa, V. Shepardson, and R. Bonett, "An architectural risk analysis of machine learning systems: Toward more secure machine learning," *Berryville Institute of Machine Learning, Clarke County, VA. Accessed on: Mar*, vol. 23, 2020.

[9] A. Shostack, "Experiences threat modeling at microsoft." *MODSEC@ MoDELS*, vol. 2008, p. 35, 2008.

[10] M. Khoshpasand and A. Ghorbani, "On the generation of unrestricted adversarial examples," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020, pp. 9–15.

[11] T. Di Noia, D. Malitesta, and F. A. Merra, "Taamr: Targeted adversarial attack against multimedia recommender systems," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2020, pp. 1–8.

[12] M. Ozdag, "Adversarial attacks and defenses against deep neural networks: a survey," *Procedia Computer Science*, vol. 140, pp. 152–161, 2018.

[13] A. Shih, A. Choi, and A. Darwiche, "A symbolic approach to explaining bayesian network classifiers," *arXiv preprint arXiv:1805.03364*, 2018.

[14] A. Ignatiev, N. Narodytska, and J. Marques-Silva, "On relating explanations and adversarial examples," *Advances in neural information processing systems*, vol. 32, 2019.

[15] C. Picardi, R. Hawkins, C. Paterson, and I. Habli, "A pattern for arguing the assurance of machine learning in medical diagnosis systems," in *Computer Safety, Reliability, and Security*, A. Romanovsky, E. Troubitsyna, and F. Bitsch, Eds. Cham: Springer International Publishing, 2019, pp. 165–179.