URET: Universal Robustness Evaluation Toolkit (for Evasion)

Kevin Eykholt* *IBM Research*
Taesung Lee* *IBM Research*
Douglas Schales *IBM Research*
Jiyong Jang *IBM Research*
Ian Molloy *IBM Research*
Masha Zorin *University of Cambridge*

Abstract

Machine learning models are known to be vulnerable to adversarial evasion attacks as illustrated by image classification models. Thoroughly understanding such attacks is critical in order to ensure the safety and robustness of critical AI tasks. However, most evasion attacks are difficult to deploy against a majority of AI systems because they have focused on image domain with only few constraints. An image is composed of homogeneous, numerical, continuous, and independent features, unlike many other input types to AI systems used in practice. Furthermore, some input types include additional semantic and functional constraints that must be observed to generate realistic adversarial inputs. In this work, we propose a new framework to enable the generation of adversarial inputs irrespective of the input type and task domain. Given an input and a set of pre-defined input transformations, our framework discovers a sequence of transformations that result in a semantically correct and functional adversarial input. We demonstrate the generality of our approach on several diverse machine learning tasks with various input representations. We also show the importance of generating adversarial examples as they enable the deployment of mitigation techniques.

1 Introduction

The powerful automation capabilities of AI have been adopted to empower and drive numerous data-driven tasks. However, the safety, security, and privacy of machine learning has become a concerning issue. In particular, adversarial machine learning, which studies how small perturbations on the input by active adversaries can predictably influence model behavior [44], indicates that use of AI in safety and security critical tasks such as cybersecurity may introduce new vulnerabilities into the system [2].

To secure machine learning algorithms against evasion attacks, penetration testing of victim models is required to identify vulnerabilities and obtain a representative set of adversarial inputs. Recent studies in adversarial machine learning have developed numerous attack algorithms [12, 14, 16, 19, 23, 33, 38].

These approaches typically make small numerical changes to the input such as adding the loss gradient or averaging two inputs and examining the influence on the output of the model. However, many of these attacks are designed to target the image classification tasks, which limits their practical use for testing real systems. Image inputs are often a collection of continuous, numerical, and independent features. Furthermore, the preprocessing is often differentiable. These properties are key requirements for many existing attacks as they craft adversarial examples through backpropagation of the loss gradient through the classification pipeline, but many machine learning models use inputs lacking these properties. For example, in malware detection [11, 30, 43], the raw input is a discrete object (*i.e.,* a binary file) and the corresponding feature representation may be a combination of discrete, continuous, and categorical values (*e.g.,* file length, library imports, file type), some which may not be independent. These features were necessarily obtained through non-differentiable feature extraction. As such, image-based adversarial machine learning attacks cannot be used to generate an adversarial binary nor an adversarial set of features. In many machine learning applications, the input properties are often only loosely fulfilled in the feature space as the classifier input often must be numerical.

Many defenses relied on a functional evasion attack for their development and deployment. Adversarial training, for example, trains a model on adversarial samples generated on-the-fly to mitigate the effect of evasion attacks [31]. As the image-based evasion attack assumptions for non-image classifiers are often only generally true in the feature space, adversarial training currently is limited to training on adversarial features. However, the adversarial features used for training may not be representative of real inputs due to semantic correctness and functionality constraints, *i.e.,* there exists no input object with the generated adversarial features.

*These authors contributed equally.

Table 1: Effect of feature-level adversarial training. Standard training denotes training on the original unmodified dataset.

Training Method	Benign Acc.	Adversarial Acc.
Standard	92%	19%
Adversarial Training	82%	29%

Indeed, in Table 1, we found adversarial training on adversarial features generated by an image-based evasion attack was ineffective against functional and semantically correct adversarial input objects generated using our framework.

One common approach to enable evasion attacks for non-image domain tasks is to retool existing image-based evasion attacks to enforce additional domain specific constraints that ensure the semantic correctness of the generated adversarial features [8, 25]. While this approach generates semantically correct adversarial features, it remains non-trivial to find an adversarial object with features. The issue remains that adversarial gradients cannot propagate through the non-differentiable mapping of input objects to their numerical feature representation.

Another approach is to pick a task domain and define a domain specific set of *functionality preserving input transformations* for the input object. These transformations define the basic manipulation operations used to craft adversarial examples [10,20] during the attack process replacing the traditional addition and subtraction of adversarial noise to a real valued input. As transformations are performed directly on the object rather than the object's feature representation, the problem of non-differentiable feature extraction is bypassed. However, most works of this type make domain specific assumptions and use customized attack algorithms, making it difficult to easily adapt these works to other domains.

To this end, a *general purpose evasion attack framework* for AI systems and implement URET (Universal Robustness Evaluation Toolkit (for Evasion)), a toolkit that can be integrated into machine learning evaluation and remediation pipelines. We define a set of functionality and semantic preserving transformations for several common data representations. In addition, we also expose a transformation interface with which a user can define their own set of transformations for given data representation. These transformations establish the basic adversarial modification operations to be used by our framework. We then characterize the adversarial input generation process as a *graph exploration problem*. Given an initial unperturbed sample, our framework finds a sequence of edges (*i.e.,* sequence of transformations) that achieve the adversarial objective (*i.e.,* misclassification) and generates an adversarial input.

From the user's perspective, they simply need to identify the input data type(s) and how those types should be modified. They also select the graph exploration parameters, namely the vertex scoring function, the vertex ranking algorithm, and the graph search algorithm. Each of these components come pre-installed in the toolkit. Users can optionally define custom input constraints and inter-feature dependencies, which our framework automatically enforces during sample generation. Our contributions are as follows:

- A general purpose adversarial input generation framework that formulates the process as a graph exploration problem in order to generate adversarial inputs irrespective of the input representation. Unlike prior works, which are limited in scope or focused on a specific domain, our framework can generate adversarial samples for any machine learning task irrespective of input data representation.
- As part of an ongoing adversarial evaluation effort, we document three case studies on machine learning tasks with different input types, highlighting the functionality and use cases of our framework.
- We demonstrate how our work can be used for adversarial remediation through integration with an adversarial training pipeline, a popular defense technique against evasion attacks.
- An open-source implementation of our framework available at `https://github.com/IBM/URET`.

2 Related work

Although adversarial machine learning is well-studied, most of its development has been focused on image-related tasks, such as object classification, image segmentation, and instance segmentation. However, powerful automation capabilities provided by machine learning has encouraged its use in other domains [42]. Non-image task domains such as cybersecurity, text classification, and even traditional tabular datasets pose issues for adversarial machine learning as more considerations must be made when manipulating input objects. Unlike images, most other input objects (*e.g.,* malicious binaries, text strings, tabular data) must also remain semantically correct and operational after adversarial manipulation. Here, we discuss related works that make advances towards developing adversarial attacks for non-image data representations. We summarize some of more generic and promising frameworks in Table 2, and compare their functionalities.

2.1 Domain Specific Attacks

One group of related works focuses on designing adversarial attacks with domain specific assumptions. Given its security critical nature, many recent works focus on the malware classification task. Grosse *et al.* refactored the Jacobian Saliency Map attack (JSMA) to enable manipulations of malware binaries [25]. Given a set of binary features (*i.e.,* $[0, 1]$ features) for a malware sample, where a value of 1 indicates that the sample exhibited the feature (*e.g.,* the malware imported a specific library). With such a feature representation, their

Table 2: Comparison of supported functionalities of non-image evasion attack tools. ◯: supported; △: partially supported; ×: not supported. Typical tabular data includes categorical, boolean, integer, and float features. A config interface refers to the ability to quickly define and evaluate a machine learning model through use of a configuration file, command line call, or similar interface. All of the works here have implementations, but they either do not provide a user guide, are not maintained as the implementation exists for reproducibility, or are private.

Attacks	Input Types			Config Interface	Opt. Goal		Open Source
	Tabular	Text	Custom		Model	Feature	
SLEIPNIR [8]	×	×	Malware	×	◯	×	△
Gym-Malware [10]	×	×	Malware	×	◯	×	△
Graph Search [27]	△	×	×	×	◯	◯	△
Pierazzi *et al.* [40]	◯	◯	◯	Unknown	◯	×	△
Counterfit [5]	×	◯	×	◯	◯	◯	◯
URET (Ours)	◯	◯	◯	◯	◯	◯	◯

proposed attack computes the Jacobian Saliency map of the sample and finds the feature indices that would most likely cause adversarial misclassification. As they purposely used an interpretable feature representation, adversarial modifications were easily reproducible. Malware functionality was also preserved as their attack only modified 0-valued features. That is to say, the adversarial attack adds new capabilities rather than removes it. This approach using limited manipulations of binary feature representations has been leveraged by other works as well [8]. The limitation of works that refactor existing image-based adversarial attacks is a requirement for interpretable, independent feature representations in order to easily map modifications back to the original input representation.

Other works have forgone adapting image-based attacks and proposed their own custom algorithms that generate adversarial cybersecurity objects [10, 20, 30]. They first define a set of functionality preserving input object transformations and use varying methods to find a sequence of transformations to cause misclassification. Demetrio *et al.* and Lucas *et al.* used the traditional approach of optimizing an objective function to find the sequence of adversarial modifications [20, 30]. Anderson *et al.* trained a reinforcement learning agent to solve this problem [10]. Although these works share similarities with our framework, they are limited to support a single data representation and a task. On the contrary, our framework is *easily customizable for any data representation* through modification of a configuration file and, possibly, the transformation interface.

2.2 Comparison with generic frameworks

Of interest are works that, like ours, propose a generic adversarial generation framework with an open-source implementation. Kulynych *et al.* [27] propose representing the process of generating adversarial samples as a graph search problem. Similar to our framework, their methodology seeks to find a sequence of transformations that cause misclassification, but we note a few key differences. First, they focus their framework on binary classifiers given their security relevance. Although the case studies we show later are also binary classification tasks, our exploration objectives support non-binary classification tasks as well. A second difference is how edges and nodes are evaluated and explored during graph exploration. They focus on the A* search algorithm as it will find an optimal adversarial example, under certain assumptions, measured by the fewest number of transformations. The current version of our framework implements several exploration configurations recognizing that a user's needs may vary. For example, some of our exploration configurations implement predictive analytics, something their framework does not support, which trade adversarial success rate for lower runtime. This trade-off can be useful for exploring large graphs or if speed is a concern, *e.g.,* adversarial training. They attempt to alleviate this issue using preset heuristics, but the effect of this approach is unclear. Of issue is their proposed preset heuristics are based on the p-norm, which is not ideal to compare categorical features. This limits their approach to numeric types such as integer and Boolean features. Finally, the authors have only provided code and instructions to reproduce their experiment results[1], but do not appear to have a plan to support a set of general purpose tools and guidelines for use by the larger community, which is an issue our work is trying to address.

A closely related work is that of Pierazzi *et al.* who formalized "problem-space" evasion attacks and proposed a general attack framework [40]. They also observe extracting feature representations of inputs in the problem-space (*i.e.,* what we denote as objects in the paper) is a non-invertible and non-differentiable process and hinders traditional evasion attacks. From our reading of the paper, their approach requires the user to define a set of domain specific input transformations and constraints. Then, their framework uses either a problem driven (*e.g.,* genetic algorithms, Monte Carlo search tree,

[1] https://github.com/spring-epfl/trickster

etc.), gradient driven, or hybrid search algorithm to discover a sequence of transformations that result in misclassification. In their paper, they focus on Android malware classification as an example task, but describe multiple other tasks in Table I of their paper. Unfortunately, due to ethical concerns, their implementation is private[2], which prevents us from comparing their implementation with ours and understanding the user interface or operation. However, it remains true that their tools are not intended to be integrated into model evaluation and remediation pipeline in its current state.

Recently, Microsoft released Counterfit, an open source toolkit for testing AI systems against adversarial machine learning attack [5]. Despite rising concerns regarding the vulnerability of AI systems to adversarial attacks, most of the businesses they surveyed were not prepared nor had the tools necessary to secure AI systems [28]. Although several libraries exist for deploying machine learning attack, most of these libraries are designed for image and text inputs [21, 24, 34, 35, 37]. To fill the need for a general adversarial evaluation toolkit, Counterfit builds upon three existing libraries, the Adversarial Robustness Toolbox (ART) [35], TextAttack [34] and AugLy [36], and exposes a command line interface to run adversarial attacks and evaluations from these frameworks. Counterfit serves to lessen the burden of knowledge for users when attempting to deploy adversarial attacks against their own systems. Thanks to the use of blackbox adversarial attacks from ART and attacks from the TextAttack library, Counterfit can support more input types and task domains than traditional adversarial attacks. As a framework, Counterfit helps to simplify the adversarial attack and evaluation process for security experts by abstracting away the details such as hyperparameter tuning and attack selection.

Although Counterfit is a major step towards enabling adversarial evasion attacks and evaluations in a general context, it is limited by the attack libraries it uses in the background. First, Counterfit cannot support uncommon input objects such as file binaries because the attacks it employs operate in the feature space. Like the work by Grosse *et al.* and Huang *et al.* [8, 25], Counterfit can only be used to generate an adversarial input object if the features are interpretable. Second, Counterfit cannot enforce custom input constraints and inter-feature dependencies, as such constraints are not supported in ART or TextAttack.

3 Preliminary

In this section, we describe the challenges and need for a general adversarial attack and robustness evaluation framework. To avoid confusion, for the remainder of the paper we will use the following terminology to describe the data going through an AI model (Figure 1):

- *Object:* An **object** denotes the input value to the AI system before feature extraction has occurred. An object can be a singular data type or a collection of data types. A data type could be an abstract data type (*e.g.,* domain name or binary) or a common data type (*e.g.,* numerical, categorical, and textual data).
- *Feature:* A **feature** denotes the input value to the machine learning model after feature extraction has been performed, and it is a numerical value such as a real number or integer. A **feature vector** is composed of one or more features. We may use features and feature vector interchangeably.
- *Input:* An **input** denotes a value anywhere before the model's input layer (*i.e.,* either an object or a feature vector).

[2]They only allow academic researchers access to their code.

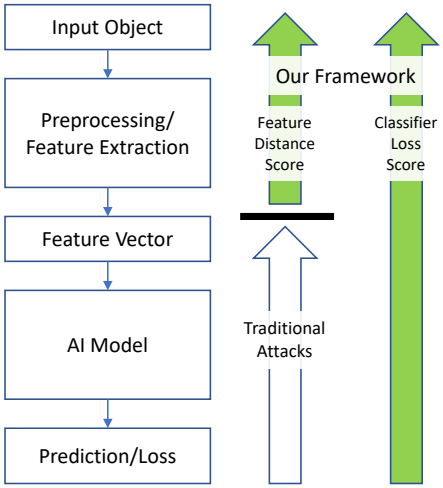

Figure 1: A typical model pipeline. Traditional adversarial attacks, which use loss gradients, are limited to generate adversarial feature vectors. Our framework can generate adversarial input objects or adversarial features using the classifier loss scoring function. Furthermore, our framework remains backward compatible with existing attacks using the feature distance scoring function.

Using the above terminology, a typical AI system consists of a feature extractor and a machine learning model. Given an *object* of a known structure, the system first converts the object into a numerical representation recognized by the machine learning model, optionally normalizing the feature values. This *feature vector* is processed by the machine learning model and an output prediction is made.

3.1 Adversarial Machine Learning

The increasing use of AI in safety and security critical systems has heightened concerns regarding the vulnerabilities AI introduces to such systems. Prior work with image-based classifiers has identified one class of attacks called *adversarial evasion attacks* [23, 44]. In such attacks, an adversary computes a precise set of input manipulations such that the perturbed

input is misclassified. Such attacks threaten the reliability of AI systems as adversaries with sufficient access can control the output decisions of the classifier. Many have proposed solutions to mitigate the effect of adversarial evasion attacks, but few have succeeded [12,18,31,39,45,46]. The main issue early adversarial defenses suffered was a reliance on gradient obfuscation, a technique that prevents correct estimation of a classifier's loss gradient, usually due to the use of non-differentiable operations. As early adversarial evasion attacks relied on the classifier's loss gradient to guide the adversarial image perturbations, gradient obfuscation based defenses appeared very effective at preventing adversarial evasion attacks. Unfortunately, Athalye *et al.* demonstrated that gradient obfuscation can be easily bypassed through use of alternative gradient approximation methods [12]. Despite the failure of early defenses, there exist several training-based defenses that are effective at improving the robustness of image-based classifiers against adversarial evasion attacks [18, 31, 45, 46].

3.2 Challenges

An evasion attack can be formulated as finding $T(x)$ for an input x such that

$$\begin{aligned} F(T(x)) &\neq F(x) \quad \text{if untargeted attack,} \\ \operatorname{argmax} F(T(x)) &= c(x) \quad \text{if targeted attack.} \end{aligned} \tag{1}$$

where F is an AI model, $c(x)$ is the target class, and T is a function applying a small change. For an image classification task, it is typically $T(x) = x + \delta$ such that $\|\delta\|_p < \varepsilon$, $\|\cdot\|_p$ is p-norm, and oftentimes δ is derived from the gradient of F with respect to x or through other numerical operations on x. However, how one can define T becomes nontrivial as we step out of the image domain.

Unlike the image domain where we can easily apply numerical modifications to x, doing so in other domains requires addressing the following challenges: 1) non-differentiable feature extraction; 2) variable input types; and 3) input semantics and functionality preservation.

Non-Differentiable Feature Extraction. Most state-of-the-art adversarial evasion attacks generate adversarial inputs by computing the loss gradient with respect to the model's input and using the loss gradient to guide the adversarial modifications. Within the model, computing the loss gradient is straightforward as machine learning models are composed of differentiable layers necessary for backpropagation. The challenge arrives when the loss gradient must be back propagated through the feature extraction layer. In the image domain, there is little to no structural difference between the object (the original image) and its extracted features as both are arrays of continuous, numerical values. When input transformations are performed, they are often done for data augmentation purposes and are differentiable (*e.g.,* shift and rotate). However, most other input objects are non-numerical, thus input transformations are required to obtain a numerical feature representation that the machine learning model can process. The difference in structure requires such transformations to be non-differentiable. As such the loss gradient cannot be backpropagated back to the input object preventing traditional attacks from generating adversarial objects. We are aware of gradient approximation techniques such as the Backward Pass Differentiable Approximation proposed by Athalye *et al.* [12], but such techniques require that the differentiable approximation be similar to the original feature extraction function and only permit slight gradient inaccuracies. Thus, existing attacks are limited to generating adversarial features.

Variable Input Types. In image recognition, the input object (an image) consists of an array of numerical, continuous values each sharing the same data type. In most domains, however, the input data types are variable. In malware detection, the input object may be a file binary. In Domain Generation Algorithm (DGA) detection, the input sample may be a string representing the domain name. In classification tasks with tabular datasets, the input object may be an array of mixed data types (*e.g.,* numerical, categorical, text, etc). As the task changes, the structure and data type(s) of the input object changes as well. This variability also extends to the input feature vector as the feature vector may consist of a mix of discrete and continuous numerical values that represent numerical quantities or preprocessed categorical features. Traditional adversarial attacks are not equipped to handle such varied input representations.

Input Semantics and Functionality Preservation. Images and their respective feature vector representations are simple inputs to adversarially modify because the modifications is a simple incremental addition of noise. Furthermore, the inputs usually consist of independent features, *i.e.,* the value of one pixel is not affected by the value of other pixels in the image. Beyond clipping the input to ensure the features remain in a certain valid range, these properties allow existing adversarial attacks to largely ignore the need to preserve the input semantics or functionality of an image object. When considering adversarial modifications on general object types or feature representations, we must take these constraints into account. Otherwise, the generated adversarial inputs might be meaningless as they would represent impossible inputs or would change the original intent of the input. For example, when designing MalGAN, the authors only allowed the attack to add new features because if a certain feature were removed, the corresponding change to the malicious binary may prevent its intended operation [26]. Other works looking to modify file binary rather than its feature representation also used custom modification functions to ensure the binary remained valid after modification [10, 20, 30].

3.3 Is adversarial machine learning a threat?

Recently, the research community has explored adversarial attacks to handle more diverse environmental conditionals and task domains against real world applications [17, 29, 47] further fueling concerns about the security of AI systems. NIST has been working on AI Risk Management Framework to foster the development of technologies to improve trustworthiness of AI including reliability, robustness, safety, and security [7]. In cybersecurity, researchers and industry have begun to recognize the threat of adversarial machine learning. MITRE recently released Adversarial Threat Landscape for Artificial-Intelligence Systems (ATLAS) [4] to systematically describe common attack tactics and techniques used in adversarial machine learning similar to MITRE ATT&CK framework [6], such as poisoning training data, evading ML model, and crafting adversarial data, with case studies. For example, a security vulnerability in the Proofpoint email protection system was discovered by building a copy-cat email classification model and evading the detection by crafting an adversarial spam email [3].

To motivate the need for generic tools that enable the study of adversarial machine learning in safety and security critical tasks, we studied the robustness of state-of-the-art malware detectors. We used an open source malware detector and a set of functionality-preserving transformations to disguise malware programs as goodware to the detector. Then, we uploaded these samples to see if they are correctly classified by the 60+ antivirus products used on VirusTotal which are commonly used to label malware samples. As Table 3 shows, the detection rate of the scanners on VirusTotal drops significantly, and some detectors could correctly detect only 6% of malware programs we generated. These results shows that we can adversarially modify an input object that remains functional and transfer the modified object to other classifiers to evade detection. Thus, there is a strong need to be able to synthetically generate such variants before an adversary exploits this vulnerability so we can fortify classifiers accordingly.

Table 3: The detection rate per scanner per malware sample.

Dataset	Average	Min	Max
Original	82%	50%	90%
Adversarial	58%	6%	91%

4 Design

In this section, we describe our generic attack framework enabling the generation of adversarial inputs irrespective of the input type and task domain. We characterize the adversarial generation process as a *graph exploration* problem in which we seek a sequence of edges from the original input vertex that results in a new vertex, but a different model prediction.

Our framework can be abstracted into two main steps: 1) Graph Definition (Sec 4.1) and 2) Graph Exploration (Sec 4.2).

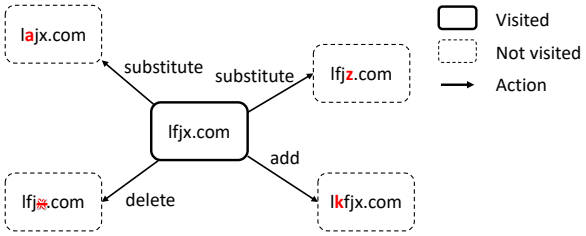

Figure 2: An illustration of graph exploration.

First, the user defines the exploration task by specifying the set of input transformations to be used as well as any functional or semantically correct constraints (Sec 4.3). Next, the user defines the graph explorer by selecting a **vertex scoring function**, an **edge ranking algorithm**, and a **graph search algorithm** from our framework. With the input transformation and explorer defined, our framework will automatically explore the graph until the objective is achieved (*e.g.,* an adversarial input is discovered) or the exploration budget is exhausted (*e.g.,* number of transformations).

4.1 Step 1: Graph Definition

Our framework can be viewed as defining a graph $G = (V, E)$ where the set of vertices V is all possible inputs and E are the edge types[3]. The edge types E are a set of semantic and functionality preserving input transformations that are chosen by the user. In our design, we only require that the input state after transformation remains semantically correct and functional. Therefore, their definition is flexible and can represent a variety of object modifications such as changing a categorical value (*e.g.,* browser `Chrome/74.0.3729.169` to `Safari/604.1`), manipulating a textual value (*e.g.,* domain name `lfjx.com` to `lfjz.com`), or directly modifying an executable file. Our current implementation includes pre-defined transformers for some common basic data types (*i.e.,* numerical, categorical, text). Figure 2 shows an example for DGA detection where the input is represented by a string value with three edge types representing the different possible input transformations. Through our transformation interface, users can easily define their own transformers in cases where the input data type or transformation is currently not supported. We used this interface to also define a set of binary file transformations used in prior work [10].

4.2 Step 2: Graph Exploration

Once the graph has been defined through selection of the input data type and its corresponding transformations, the next step is to define a graph explorer. Starting at a vertex representing the current input, the explorer evaluates the connected edges and selects one or more edges based on the search method.

[3]Note that this is a theoretical representation of the dynamic graph to be explored as building a static graph of all possible inputs is prohibitively expensive (*e.g.,* for domain names, there are up to $\sum_{i=3}^{63} 36 \times 37^{i-1}$ vertices)

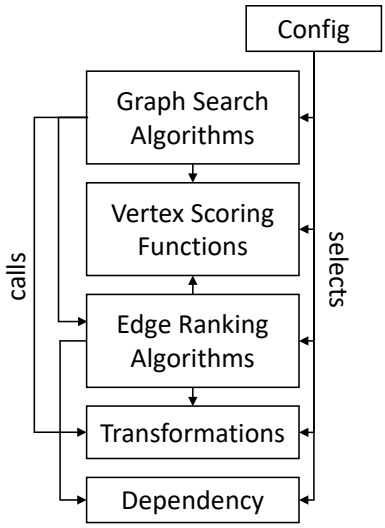

Figure 3: Our framework structure. A configuration file is read by our framework and automatically constructs an graph explorer consisting of several components.

The transformation corresponding to the selected edge(s) is applied and it is checked to see if the adversarial object has been achieved (*i.e.,* an adversarial input is found). If not, the edge is scored and exploration continues. This process is repeated with the selected vertices until the exploration objective is satisfied or the exploration budget is exhausted. If exploration was not terminated early due to the exploration objective being achieved, then the "best" scoring vertex is returned. In Figure 3, we show an overview of the graph explorer created by a configuration file.

4.2.1 Vertex Scoring Function

Our framework supports two vertex scoring functions by default: 1) classifier loss and 2) feature distance. The scoring function informs our framework the "closeness" a vertex is achieving the objective. In general, the goal of our framework is to find a vertex that is adversarial, *i.e.,* the vertex represents an input different from the original input, is within some number of hops away from the original vertex (or within some norm bound of the original input), and is misclassified by the classifier. When found, exploration is terminated. The vertex scoring functions define a metric to guide exploration in achieving this objective.

The **classifier loss scoring function** is used when the objective is to maximize the classifier loss for the current input z as is done in traditional adversarial machine learning. Given a classifier F and the classifier's prediction on the original input y, we score the vertex belonging to z as follows:

$$score = \mathbf{L}(F(z), y) \tag{2}$$

where L is the classifier's loss function. Traditionally, the cross entropy function is used. If the user desires to generate an adversarial input that is misclassified as a specific target label y_t rather than a random label that is not y, we modify Equation 2 to minimize the loss with respect to the target label.

While our framework can generate adversarial examples on its own, we recognize that some users would prefer to leveraging existing adversarial evasion attacks. With such attacks, a user would first generate an adversarial feature vector as they cannot propagate the modifications through the feature extraction component [14,23,38]. Then, the generated adversarial feature vector would be used by our framework as the exploration objective. The advantage of this approach is that existing adversarial attacks are well studied for numerical input types and may provide better results than relying on the classifier loss in certain scenarios. The **feature distance scoring function** encourages finding an input z that when passed through feature extractor E is as close to the target features f as possible with respect to a distance function D. The vertex score is expressed as follows:

$$score = -D(E(z), f) \tag{3}$$

where $E(z)$ is the feature representation of z if z is an object. If z is already a feature vector, then E is the identity function. The negative sign ensures that closer inputs with respect to the target features are given higher scores. As the framework may be unable to exactly find an input z with input features f due to semantic correctness and functionality constraints, early termination with this scoring function is still possible if the current input is adversarial.

As a final point, although our current design pre-defines these two loss vertex scoring functions, our exploration framework can support other scoring methods if defined by the user. For example, the cost-based scoring function proposed by Kulynych *et al.* [27] or a combination of multiple scoring functions could be used in our framework.

4.2.2 Edge Ranking Algorithm

Our framework pre-defines four edge ranking algorithms: a) Random, b) Brute-Force; c) Lookup Table; and d) Model Guided.

Given a vertex, the Random ranking algorithm randomly returns one or more edges without scoring them. It is the fastest of the ranking algorithms, but has the lowest success rate. As we will discuss later, this algorithm is usually used in combination with the Simulated Annealing exploration algorithm.

Given a vertex, the **Brute-Force** algorithm loops through each connected edge and applies the transformation specified by the edge (*i.e.,* visits the connected vertex). The score of a connected edge is equal to the vertex score of its connected vertex. For example, in Figure 2, assume the current vertex is "lfjx.com" and cross-entropy loss is used to score edges. For brevity, we'll also assume that the four edges

shown are the only possible transformations. Brute-Force would perform each transformation individually, and then classify the transformed inputs. Thus, it would generate the strings "lajx.com", "lfjz.com", "lfj.com", "lkfjx.com", score each one's edge based on its respective classification loss. As the transformations are done during runtime, the Brute-Force approach is often the slowest of the ranking algorithms. However, it is the most accurate as the edge rankings are based on the real state of the explored vertices.

The **Lookup Table** algorithm first runs a training phase using a small set of samples. For each training sample, its 1-hop neighborhood is explored. For each edge encountered, the current edge weight is computed using the Brute-Force approach and is stored a the lookup table, averaged with prior computed values of the edge weight. By exploring the 1-hop neighborhood of every training sample, the Lookup Table ranking algorithm generates a table of estimated edge weights for every unique edge encountered during training. During exploration, the ranking algorithm consults the table for the average edge weight of each connected edge and uses those values to rank the edges. As it does not consider the current vertex state, the edge ranking produced may differ from the true ranking, which often results in a lower success rate compared to Brute-Force. However, the table lookup is often faster than transforming the input as Brute-Force does.

The **Model Guided** algorithm also ranks edges without visiting the vertices by relying on a pre-trained model for edge selection. Compared to Lookup Table algorithm, this provides the user a possibility to consider the current vertex state to predict the edge ranking more accurately. A model can take the current vertex and the vertex score to predict which edges are likely to improve the score. While our framework can take any such model to rank edges, in our experiment, we leverage Keras-RL [41] to train a reinforcement learning policy as reinforcement learning considers both the current and future rewards for a particular transformation. As such, our framework includes support for implementing reinforcement learning policies to be used with the Model Guided algorithm. We describe the implementation detail in the Appendix.

4.2.3 Graph Search Algorithm

Our framework implements two search algorithms by default: Beam Search and Simulated Annealing. The search algorithm dictates which neighboring vertices are propagated to the next epoch of exploration.

The **Beam Search** algorithm is an algorithm that only passes the top-k best scoring nodes to the next epoch of exploration. The number of edges passed is determined by the **beam width**. We also include **depth** parameter, which defines the maximum number of edges to explore with respect to the original input and implicitly defines the **transformation budget**. The beam search algorithm can be combined with any of the edge ranking algorithms.

The **Simulated Annealing** algorithm is a temperature-guided time-restricted random search algorithm. Unlike Beam Search, this algorithm can only be used with the Random ranking algorithm by definition and has an additional **time budget** parameter, which defines the per-sample exploration time. During each exploration epoch, it randomly selects a random length sequence of edges starting from the current input vertex, evaluates the new input state, and then determines if the new input state should be kept (i.e set as the new current state) based on the new vertex's objective score and the current *temperature*. The temperature is a parameter that initially encourages exploration of new input states in the early phases of exploration. As time passes, the temperature gradually decreases, which discourages exploration and instead causes the algorithm to prefer to keep input states that better satisfy the exploration objective.

4.3 Semantic Correctness and Functionality

A key component of our framework is to ensure the semantic correctness and functionality of generated adversarial inputs. Our framework defines modification functions, which we denote as **input transformers**, for a few basic input types: integers, floats, booleans, strings, categorical values, and one-hot categorical values. These input types are often used in most AI systems, at least at the feature representation level, and they can be combined to obtain new input representations. Furthermore, for most of these input types, the constraints on semantic correctness and functionality are usually very simple to ensure. For example, given a vector of floats, e.g., representing an image, the constraints take the form of ensuring the floats are bounded in a certain range.

Due to the wide adoption of machine learning, there are many possible task domains and a variety of input types. Prior works in malware detection [10,20,30] used custom modification functions tailored for the task domain and our framework expands on this approach. Rather than enumerating every possible input type, we allow users to define new input transformations through an easy-to-use interface as users are likely to possess the necessary domain knowledge and understanding of input constraints. Through our interface, the user simply needs to define the transformation operation and, if necessary, methods to compute a set of possible transformations given an input value. Once defined, our framework automatically uses the transformations when encountering the respective input type. As the user provides the definition, the adversarial inputs generated by the framework will be semantically correct according to their definition.

Additionally, our framework also includes the ability for users to define individual constraints on an input value or, in the case of a vector with multiple input values, dependency constraints between input values. The L_p norm constraint used for image-based adversarial attacks is an example of an individual constraint for a float type of an input value. An-

other example is an edit distance for non-numerical inputs, such as strings. As inputs can have multiple representations, the definition of input transformations includes a function to enforce such constraints if necessary. After transforming an input, our framework checks if the user defines any dependency constraints as part of the attack. A dependency constraint is expressed as a function and multiple dependency constraints can be passed to our framework. For example, suppose that we have a feature vector of length three and the third index is computed by summing the first two indices. A user can express this relationship with a dependency function that takes in the current input state and the first two indices as input, sums their values, assigns it to the third index, and then returns the new input.

5 Using Our Framework

In this section, we'll give a brief overview of the typical user experience when using our framework. For this discussion, we will use the DGA classification task and the Beam Search (Brute-Force) exploration configuration as an example. First, the user must define the set of transformations to be used. A user defines the transformer and its parameters in a configuration file such as in Figure 4. Our framework reads in the **transformer_params** dictionary and creates a transformer object for each entry. In the DGA classification task, as the input object is a single text string representing a domain name, thus we only need a single string transformer. The transformer definition also includes some initialization arguments. The **subtransformer_args** define the modification actions, which we denote as subtransformers, that can be performed on the data type. In this example, we allow insertion, substitution, and deletion of alphanumeric characters in the string. Note that the modification actions may also have their own initialization arguments (*e.g.,* only numerical characters can transformed). The **input_constraints** define constraints that must be true after an input has been transformed. In this example, we have a simple action constraint that stipulates a string can only be modified three times. Furthermore, there is an additional constraint that insertion and substitution actions can only be performed three times and the deletion action can only remove at most half of the original string. Finally, the **input_processor_name**, informs our framework that before and after transforming the domain name, a user-defined function of the specified name must be used to process the domain name. This function is one process that ensures semantic correctness and functionality of the transformer input

In addition to defining the transformers in the configuration file, the user also defines the explorer parameters as shown in Figure 5. Observe that we split the ranking and exploration algorithm parameters. Both parameter sets define the **type** of algorithm to use as well as the initialization arguments. The vertex scoring algorithm is given in **scoring_alg** and we also provide the parameters specific to Beam Search. We note that

```
transformer_params:
    - data_type: "string"
      input_processor_name: "domain_name_processor"
      init_args:
          input_constraints:
              max_actions: 3
              max_subtransformer_actions:
                  - 3
                  - 3
                  - 0.5
          subtransformer_args:
              - name: "Insert"
              - name: "Substitution"
              - name: "Delete"
```

Figure 4: The string transformer definition for the DGA task.

with respect to the ranking algorithm, an additional Boolean flag, **multi_feature_input**, is set. It is False by default, but we include it here for illustration. This flag is only True for input types that combine multiple data types and would require multiple transformers such as with tabular data. Finally, due to the numerous machine learning frameworks and model definitions, the **predict_function_name** is an optional value that informs the framework what function can be used with the model object to obtain a prediction. By default, it is set to "predict".

```
explorer_params:
    type: "BeamSearch"
    predict_function_name: "predict"
    init_args:
        scoring_alg: "model_loss"
        search_size: 3
        max_depth: 3
ranker_params:
    type: "BruteForce"
    init_args:
        multi_feature_input: False
```

Figure 5: The explorer definition for the DGA task.

After the user loads a model to attack and defines required input processing and feature extraction functions, the configuration file is automatically processed by our framework and an explorer object is returned. Calling it automatically generates adversarial samples for a given set of inputs as shown in Figure 6.

```
cf = "configs/dga/brute.yml"

explorer = process_config_file(cf,
                               classifier,
                               feature_extractor=fe,
                               input_processor_list=[domain_name_processor])
adv_samples, transformations_records = explorer.explore(task_samples,
                                                        return_record=True)
```

Figure 6: Using the explorer to generate samples.

6 Evaluation

In this section, we present three case studies in which we used our framework as part of an ongoing model evaluation task to generate adversarial inputs. The generic nature of our framework was key in enabling such an evaluation due

to differences in input data representations, input semantics, and model types. First, we use the 2018 Home Mortgage Disclosure Act (HMDA) dataset and a mortgage approval classifier to explore *adversarial sample generation for tabular inputs*, an input data type not commonly studied in current adversarial work. Next, we revisit *text and numerical inputs* and generate adversarial samples for the DGA classification task. Finally, we explore generation of adversarial samples for a well-studied, but uncommon input data type, *a binary object*, using a model trained on the EMBER malware classification dataset [11]. Across all three case studies, we report the following metrics when evaluating our exploration algorithm configurations:

1. Success rate: The number of adversarial inputs that were misclassified by the classifier.
2. Average number of transforms: The average number of transformations applied to the original input before misclassification occurred. Note this metric is only computed across *successful* adversarial inputs.
3. Average time per sample: The average runtime of the exploration algorithm to return a potentially adversarial input. This time does not include time required for training for certain algorithms (*i.e.,* Lookup Table and Model Guided).

While our framework may not be the most efficient compared with individual prior work tailored for a specific domain, our framework is generic, *i.e.,* we can generate adversarial samples for *any input data type*, which enables its use in the evaluation pipeline. Even as we switched between machine learning tasks, the necessary modifications were mostly small-scale changes to our configuration files.

6.1 Tabular Input

In this first case study, we use our framework to generate adversarial examples for tabular inputs. To our knowledge, little to no prior works have studied adversarial attacks using tabular inputs or, more generally, inputs that are a combination of multiple data types. The closest works would be adversarial attacks against multi-modal systems [13, 22].

The 2018 release of the Home Mortgage Disclosure Act (HMDA) dataset contains mortgage data from 5,683 institutions. This dataset is a tabular dataset consisting of mix of categorical and numerical features. The classification task is given a tabular data point representing a consumer's mortgage application, predict if the application should be approved or rejected. We were asked to perform an evaluation of the adversarial robustness of five different pre-trained classifiers: a decision tree (DT) classifier, a gradient boosted classifier (GBC), a logistic regression (LR) classifier, a random forest (RF) classifier, and a multi-layer perceptron (MLP) classifier. The accuracy and F1 Score of each classifier are given in Table 4. We were also given a pre-processed version of the 2018 HMDA dataset. Compared to the original dataset, the pre-processing sanitized it and, through feature selection, extracts 13 features for classification.

Table 4: The performance of each of the HMDA classifiers.

Model Arch.	Accuracy	F1 Score
Decision Tree	91%	0.95
Gradient Boosted Classifier	95%	0.97
Logistic Regression	69%	0.81
Random Forest	81%	0.89
Multi-layer Perceptron	83%	0.90

Table 5: The 7 features selected for adversarial modification from the HMDA dataset

Feature Name	Type	# of Possible Values
Age	Categorical	7
Score Type	Categorical	8
Underwriter	Categorical	6
Loan Limit	Categorical	2
Loan Duration	Categorical	2
Gender	Categorical	2
Race	Categorical	2

Based our understanding of task and evaluation goals, we selected 7 categorical features of the 13 total features as potential candidates for modification. We describe these features in Table 5. In our configuration files, we defined 7 categorical transformers. With respect to exploration algorithm specific parameters, the Beam Search algorithm used a width of 5 and a depth of 2, *i.e.,* an input could be transformed at most twice. The Lookup Table ranking algorithm was trained on 500 randomly selected training inputs and the Model Guided was a reinforcement learning model (See Appendix). The Simulated Annealing algorithm was given a time budget of 1 s per sample and also was limited to 3 transformations. Finally, we use the classification loss scoring function to guide exploration.

In Table 6, we report the results for the 5 potential exploration algorithm configurations on each of the models we were given using 1000 approved and 1000 rejected applications. As a baseline comparison, we include the results for Beam Search using the Random algorithm. Within the Beam Search configurations, the Brute-Force ranking algorithm had the highest success rate across all experiments. It also uses the fewest number of transformations to craft an adversarial sample. The Lookup Table ranking algorithm is faster that Brute-Force due to the pre-training step, but had a lower success rate as it ignores the current state of the input when determining the next transformation to apply. We see that the Model Guided ranking algorithm improves upon Lookup Table with respect to success rate, but as its exploration is

still based on an estimation of the edge weights, it uses more transformations to succeed. Simulated Annealing with its fixed time budget also has a high success rate.

6.2 Text Input

In our second case study, we use our framework to generate adversarial examples for text inputs. We recognize that much prior work exists with respect to generating adversarial text inputs, namely Counterfit [5] and TextAttack [34]. While these works may be more efficient or successful at finding adversarial text inputs compared to our framework, they cannot be easily adapted to other input data types.

Domain name generation algorithms (DGA) are often used by malware to locate the command and control servers and avoid simple blacklisting. Instead of relying on fixed IP address or domain name hard-coded into the malware, domain name generation algorithms generate a large of number of domain name strings in a pseudorandom fashion, one of which connects to the actual command and control server [1]. The classification task is given a domain name, predict if was produced by a domain name generation algorithm. We were provided with a DGA classifier that was a 2-layer neural network. We were also given a test dataset containing 5,000 DGA names and 5,000 non-DGA names. Given a domain name, the classifier extracts 20 numerical bounded features to use for classification. The accuracy of the classifier on the test dataset is 93%.

As the domain name is a single string, we defined a string transformer with the ability to add, delete, or substitute alphanumeric characters in the string. We also defined a function to preprocess the domain name and extract its top level domain (*e.g.,* ".com") as this portion of the string was not valid for modification. With respect to exploration algorithm specific parameters, the Beam Search algorithm used a width of 3 and a depth of 3, *i.e.,* an input could be transformed at most three times. The Lookup Table ranking algorithm was trained on 500 randomly selected training inputs and the Model Guided algorithm was trained on all of the samples in the test set. The Simulated Annealing algorithm was given a time budget of 1 s per sample and was also limited to 3 transformations. Finally, we use the classification loss scoring function to guide exploration.

In Table 7, we report the results of our study on 1000 DGA and 1000 non-DGA samples. As a baseline comparison, we include the results for Beam Search using the Random algorithm. As before, we observe that the Brute-Force algorithm has the highest success rate and the fewest number of transformations. The Lookup Table algorithm is faster in comparison, but uses more transformation. The Model Guided algorithm, in this case, has worse performance compared to Brute-Force and requires more transformations on average. Finally, Simulated Annealing also has a high success rate and allows the user to exactly specify exploration time, but usually maximizes its transformation budget.

As the classifier's feature extractor generates an array of continuous numerical features, we can use existing image based adversarial attacks to generate an adversarial feature vector for a given domain name. In our second evaluation, we used the Projected Gradient Descent (PGD) attack [31] to generate adversarial feature target vectors for the 1000 DGA and 1000 non-DGA samples. Then, we define the feature distance scoring function to be the cosine similarity between the current feature perturbation induced by an input transformation and the PGD target vector. We also set a modification constraint specifying that a feature can be only modified by up to 30% of the feature's maximum value in order to mimic the inconspicuousness property image-based adversarial attacks enforce. Except for the change in scoring function, we use the same exploration parameters as before.

In Table 8, we report the results of this evaluation. Of note is that the Model Guided ranking algorithm is much faster compared to the other Beam Search configurations. Clearly, the RL model is very effective at estimating the current and future effects a particular transformation induces, leading to an overall decrease in time required to generate an adversarial sample. However, we see there is an overall drop in success rate and an increase in both average number of transformations and average time per sample for the other exploration algorithm configurations. The drop in success rate is likely due to the fact that some of the adversarial target vectors generated by the PGD attack are not realizable. The increase in number of transformation and time per sample indicates that the selected input transformations, although resulting in a feature perturbation in the target direction, are likely not the most efficient path to generate a misclassified adversarial input. This particular experiment shows that our framework is backwards compatible with traditional adversarial attacks through use of the feature distance scoring function.

As our framework enables the generation of adversarial inputs regardless of the task or input type, we now have a method to improve the model's accuracy on adversarial samples. Adversarial training is a state-of-the-art defense that improves a model's adversarial accuracy by generating adversarial inputs on-the-fly during training. [31]. As we showed in Table 1, although adversarial training could be used with adversarial features, it was not very effective on real adversarial objects. In Table 9, we report the performance of the DGA classifier when trained using standard training and adversarial training. As we see, a model trained on adversarial objects exhibits much higher adversarial accuracy with minimal impact to it natural (*i.e.,* non-adversarial) accuracy compared to training with adversarial features.

6.3 Non-Standard Input - Binary

In our last case study, we use our framework to adversarial examples for a non-standard input type, binary files. With

Table 6: HMDA experimental results.

Model Arch.	Algorithm	Success Rate	Avg. # of Transforms	Avg. Time/sample
Decision Tree	Beam Search (Random)	38%	1.30	0.001 s
	Beam Search (Brute-Force)	92%	1.13	0.010 s
	Beam Search (Lookup Table)	89%	1.63	0.002 s
	Beam Search (Model Guided)	81%	1.85	0.018 s
	Simulated Annealing	97%	1.87	1.000 s
Gradient Boosted Classifier	Beam Search (Random)	14%	1.43	0.003 s
	Beam Search (Brute-Force)	58%	1.08	0.044 s
	Beam Search (Lookup Table)	26%	1.41	0.026 s
	Beam Search (Model Guided)	52%	1.74	0.058 s
	Simulated Annealing	57%	2.00	1.000 s
Logistic Regression	Beam Search (Random)	34%	1.38	0.002 s
	Beam Search (Brute-Force)	100%	1.05	0.007 s
	Beam Search (Lookup Table)	69%	1.12	0.007 s
	Beam Search (Model Guided)	88%	1.93	0.020 s
	Simulated Annealing	100%	2.00	1.000 s
Random Forest	Beam Search (Random)	27%	1.46	0.352 s
	Beam Search (Brute-Force)	100%	1.04	1.462 s
	Beam Search (Lookup Table)	70%	1.08	1.177 s
	Beam Search (Model Guided)	86%	1.96	0.042 s
	Simulated Annealing	75%	1.87	1.000 s
Multi-Layer Perceptron	Beam Search (Random)	36%	1.41	0.198 s
	Beam Search (Brute-Force)	100%	1.04	0.724 s
	Beam Search (Lookup Table)	94%	1.39	0.369 s
	Beam Search (Model Guided)	71%	1.92	0.297 s
	Simulated Annealing	97%	1.90	1.000 s

Table 7: DGA experimental result - Generating Adversarial Objects with Classifier Loss Objective.

Algorithm	Success rate	Avg. # of Transforms	Avg. Time / sample
Beam Search (Random)	23%	1.84	0.093 s
Beam Search (Brute-Force)	85%	1.24	0.363 s
Beam Search (Lookup Table)	45%	1.61	0.277 s
Beam Search (Model Guided)	70%	2.56	0.400 s
Simulated Annealing	62%	2.28	1.000 s

Table 8: DGA experimental result - Generating Adversarial Objects with Feature Distance Objective.

Algorithm	Success rate	Avg. # of Transforms	Avg. Time / sample
Beam Search (Random)	27%	1.87	0.091 s
Beam Search (Brute-Force)	56%	1.93	22.835 s
Beam Search (Lookup Table)	50%	1.79	12.415 s
Beam Search (Model Guided)	43%	2.69	0.606 s
Simulated Annealing	26%	2.72	1.000 s

respect to manipulated binary inputs, there are several works that proposed adversarial attack algorithms against malware classifiers [10, 20, 25, 30], but, as with text inputs, these algorithms are not generic. It is a non-trivial task to adapt their proposed methods to other input data types.

We also focus our study on the malware classification task using the EMBER dataset, a collection of feature vectors extracted from 1.1 million binary files with goodware/malware labels [11]. Given a binary file, the classifier must predict if the file is malware or goodware. We use a pre-trained classifier trained on 900,000 training data points in which one third of the training data is unlabeled. The remaining two thirds are labeled and balanced between malware and goodware. On the 200,000 test data points, the accuracy of the classifier is 97%. The model authors also provide a binary file feature

Table 9: Evaluation of the adversarial accuracy of the DGA model when trained using standard and adversarial training. Standard training denotes training on the original unmodified dataset. Standard adversarial training uses PGD to generate adversarial feature vectors. DGA Adversarial training uses adversarial objects generated by Beam Search (Model Guided). Note that the ranking model was trained on a pre-trained DGA classifier trained on the original dataset.

Training Method	Natural Accuracy	Adversarial Accuracy
Standard training	92%	19%
Standard Adversarial Training (Adversarial Features)	82%	29%
Adversarial Training with Adversarial Objects (Ours)	92%	**45%**

extractor and SHA256 hashes of the dataset binaries so that the original binary files can be collected.

Unlike our previous case studies, binary files are a non-standard input type so we must first define a new transformer class responsible for perturbing and ensuring semantic correctness and functionality of the transformer binary. Drawing from prior work [9], we implemented six types of functionality-preserving transformations which modify the PE header: binary (un)packing, adding sections, renaming sections, adding imports, removing debugging header information, and appending header data. In this study, we will only report results for a Beam Search exploration with a Model Guided ranking algorithm. The ranking model follows reinforcement learning method described in the Appendix[4]. We also use the feature distance scoring objective, but instead use the L_2 distance between the current perturbation vector and target perturbation vector to guide exploration. The target adversarial perturbation vectors are generated using the HopSkipJump algorithm [15] against the previously described malware classifier. We evaluate the malware classifier against 93 correctly labelled malware samples from the test dataset. As the original dataset only contains the extracted feature vectors, we use the provided SHA256 hashes to obtain the original malware binaries. With a beam width of 5 and a depth of 3, *i.e.,* a binary could be transformed at most 3 times, our framework generated 38 adversarial malware binaries.

It is a well-known fact that adversarial examples in the image domain are transferable [44], *i.e.,* adversarial examples generated for one model are highly likely to be misclassified by other models trained for the same task. In Table 10, we report the success rate of the previously generated 38 adversarial malware binaries against a set of VirusTotal detectors. We see that the adversarial modifications performed by our framework cause the average detection rate to fall from 82% to 58%. In the worst case, one of the detectors only flagged 6% of the binaries are malicious. In Table 11, we provided the precision and recall of 6 different anonymized detectors found in VirusTotal computed across the original and adversarial binaries. While we recognize that VirusTotal is a mix of signature and non-signature based detectors, it remains true that many organizations rely on VirusTotal to label malicious binaries. These results suggest that adversaries can exploit adversarial transferability to evade detection, especially if more advanced input transformations are used.

Table 10: The detection rate of the VirusTotal detectors on the original and adversarially modified malware binaries.

Dataset	Average	Min	Max
Original Binaries	82%	50%	90%
Adversarial Binaries	58%	6%	91%

[4]As malicious binaries are risky to handle, we train the algorithm multiple times on 12 benign binaries.

Table 11: Performance of 6 different VirusTotal Detectors.

	Precision		Recall	
Scanner	Original	Adversarial	Original	Adversarial
A	100%	76%	100%	91%
B	100%	68%	100%	56%
C	100%	77%	100%	81%
D	100%	100%	99%	70%
E	100%	60%	60%	58%
F	100%	100%	43%	28%

7 Conclusion

The vulnerability of AI systems to adversarial machine learning motivates a need for tools that enable proper study. As traditional adversarial attack algorithms were designed for image-based systems, strict assumptions regarding input structure have been made that limit the use of such algorithms in other data domains. In this paper, we proposed a new adversarial attack framework capable of generating adversarial inputs for AI systems regardless of the modality or task. We represent the adversarial generation process as a graph exploration problem. Our framework searches for a sequence of edges representing input transformations that result in satisfying the exploration object, *e.g.,* finding an adversarial input. We pre-define several data transformers for common input types. In addition, we provide a transformation interface to address cases where a particular data type is unsupported. In such cases, the user can rely on their domain knowledge to define the new data transformer, which can be integrated into our tools for future use, as we did when defining binary transformations. In an effort to ease usability, we allow users to customize the adversarial generation process through use of configuration files, abstracting away many of the low level implementation details. As we highlight in our three evaluation case studies, the current version of our framework contains several possible exploration configurations, with respectable effectiveness despite the variety in data representation, model types, and semantic/functionality constraints. Switching between machine learning tasks in the studies only required small scale modifications to the exploration parameters and data transformation definitions.

As we are making our toolkit publicly available[5], we recognize there are ethical concerns regarding malicious use of our tools, motivating a push to privatize our framework as Pierazzi *et al.* have done [40]. However, we do not believe that keeping such tools private will benefit the long term health of machine learning as malicious actors can still develop such tools independently. Rather, by providing these tools to the community, we provide academic and industry researchers alike the ability to study the effects of adversarial attacks in new input domains and develop generic adversarial mitigation techniques.

Acknowledgements. We thank Mohinder Singh for providing

[5]https://github.com/IBM/URET

us with the pre-trained HMDA models and our anonymous reviewers for their valuable feedback. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] Domain generation algorithm. https://en.wikipedia.org/wiki/Domain_generation_algorithm. Accessed: 2021-07-09.

[2] CWE-1039: Automated recognition mechanism with inadequate detection or handling of adversarial input perturbations. https://cwe.mitre.org/data/definitions/1039.html, 2018.

[3] Response to CVE-2019-20634. https://www.proofpoint.com/us/security/security-advisories/pfpt-sn-2020-0001, 2020.

[4] Adversarial threat landscape for artificial-intelligence systems. https://atlas.mitre.org/, 2021.

[5] Counterfit. https://github.com/Azure/counterfit, 2021.

[6] MITRE ATT&CK. https://attack.mitre.org/, 2021.

[7] NIST AI Risk Management Framework. https://www.nist.gov/itl/ai-risk-management-framework, 2021.

[8] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. Adversarial deep learning for robust detection of binary encoded malware. In *Deep Learning and Security Workshop (DLS)*, 2018.

[9] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. 2018. arXiv:1801.08917.

[10] Hyrum S. Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. In *Black Hat USA*, 2017.

[11] Hyrum S. Anderson and Phil Roth. Ember: An open dataset for training static pe malware machine learning models. 2018. arXiv:1804.04637.

[12] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *International Conference on Machine Learning (ICML)*, 2018.

[13] Yulong Cao, Ningfei Wang, Chaowei Xiao, Dawei Yang, Jin Fang, Ruigang Yang, Qi Alfred Chen, Mingyan Liu, and Bo Li. Invisible for both camera and lidar: Security of multi-sensor fusion based perception in autonomous driving under physical-world attacks. In *Symposium on Security and Privacy (S&P)*, 2021.

[14] Nicholas Carlini and David A. Wagner. Towards evaluating the robustness of neural networks. In *Symposium on Security and Privacy (S&P)*, 2017.

[15] Jianbo Chen, Michael I. Jordan, and Martin J. Wainwright. Hopskipjumpattack: A query-efficient decision-based attack. 2020. arXiv:1904.02144.

[16] Pin-Yu Chen, Huan Zhang, Yash Sharma, Jinfeng Yi, and Cho-Jui Hsieh. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2017.

[17] Shang-Tse Chen, Cory Cornelius, Jason Martin, and Duen Horng Chau. Shapeshifter: Robust physical adversarial attack on faster r-cnn object detector. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, 2018.

[18] Jeremy M Cohen, Elan Rosenfeld, and J. Zico Kolter. Certified adversarial robustness via randomized smoothing. 2019. arXiv:1902.02918.

[19] Francesco Croce and Matthias Hein. Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In *International Conference on Machine Learning (ICML)*, 2020.

[20] Luca Demetrio, Scott E. Coull, B. Biggio, Giovanni Lagorio, Alessandro Armando, and Fabio Roli. Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. 2020. arXiv:2008.07125.

[21] Gavin Weiguang Ding, Luyu Wang, and Xiaomeng Jin. AdverTorch v0.1: An adversarial robustness toolbox based on pytorch. 2019. arXiv:1902.07623.

[22] Ivan Evtimov, Russell Howes, Brian Dolhansky, Hamed Firooz, and Canton Cristian. Adversarial evaluation of multimodal models under realistic gray box assumptions. In *Workshop on Adversarial Machine Learning in Real-World Computer Vision Systems and Online Challenges (AML-CV)*, 2021.

[23] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR)*, 2014.

[24] Dou Goodman, Hao Xin, Wang Yang, Wu Yuesheng, Xiong Junfeng, and Zhang Huan. Advbox: a toolbox to generate adversarial examples that fool neural networks. 2020. arXiv:2001.05574.

[25] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European Symposium on Research in Computer Security (ESORICS)*, 2017.

[26] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. 2017. arXiv:1702.05983.

[27] Bogdan Kulynych, Jamie Hayes, Nikita Samarin, and Carmela Troncoso. Evading classifiers in discrete domains with provable optimality guarantees. 2018. arXiv:1810.10939.

[28] Ram Shankar Siva Kumar, Magnus Nyström, John Lambert, Andrew Marshall, Mario Goertzel, Andi Comissoneru, Matt Swann, and Sharon Xia. Adversarial machine learning - industry perspectives. 2020. arXiv:2002.05646.

[29] Jinfeng Li, Shouling Ji, Tianyu Du, Bo Li, and Ting Wang. Textbugger: Generating adversarial text against real-world applications. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, and Saurabh Shintre. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Asia Conference on Computer and Communications Security (ASIA CCS)*, 2021.

[31] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representation (ICLR)*, 2018.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[33] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Computer Vision and Pattern Recognition Conference (CVPR)*, 2016.

[34] John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. In *Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*, 2020.

[35] Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Beat Buesser, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Ian Molloy, and Ben Edwards. Adversarial robustness toolbox v1.0.0. 2018. arXiv:1807.01069.

[36] Zoe Papakipos and Joanna Bitton. Augly: Data augmentations for robustness. 2022. arXiv:2201.06494.

[37] Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, and Rujun Long. Technical report on the cleverhans v2.1.0 adversarial examples library. 2018. arXiv:1610.00768.

[38] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *European Symposium on Security and Privacy (EuroS&P)*, 2016.

[39] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Symposium on Security and Privacy (S&P)*, 2016.

[40] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *Symposium on Security and Privacy (S&P)*, 2020.

[41] Matthias Plappert. keras-rl. https://github.com/keras-rl/keras-rl, 2016.

[42] Ethan M. Rudd, Felipe N. Ducau, Cody Wild, Konstantin Berlin, and Richard Harang. ALOHA: Auxiliary loss optimization for hypothesis augmentation. In *USENIX Security Symposium*, 2019.

[43] Octavian Suciu, Scott E. Coull, and Jeffrey Johns. Exploring adversarial examples in malware detection. In *Deep Learning and Security Workshop (DLS)*, 2019.

[44] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.

[45] Eric Wong and Zico Kolter. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning (ICML)*, 2018.

[46] Huan Zhang, Hongge Chen, Chaowei Xiao, Sven Gowal, Robert Stanforth, Bo Li, Duane Boning, and Cho-Jui Hsieh. Towards stable and efficient training of verifiably robust neural networks. 2019. arXiv:1906.06316.

[47] Zhe Zhou, Di Tang, Xiaofeng Wang, Weili Han, Xiangyu Liu, and Kehuan Zhang. Invisible mask: Practical attacks on face recognition with infrared. 2018. arXiv:1803.04683.

A Model used for Model Guided Ranking

There are multiple ways to train a model to map an input to select the best action (*i.e.,* input transformation) to produce an adversarial example. In our experiments, we use reinforcement learning for the Model Guided algorithm because such models consider the effect a transformation has on both current and future rewards. In our specific implementation, we adapted Deep Q Learning [32], to our problem domain.

In general, Deep Q Learning trains a neural network or *neural policy* that takes the current state to predict the rewards of actions. The training process samples an action from the neural policy and applies it to get actual reward score. If it leads to a state with a higher reward, the model is updated to take more of the chosen action for the similar states. The reward for an action includes the future reward to backpropagate the effect to earlier action choices. In our case, the reward is vertex score, and actions are input transformations, and we use this Deep Q Learning implementation provided by Keras-RL [41] for this process.

Our main adaptation is on training policy that decides which action to take during training. Plain Deep Q Learning using a randomly initialized neural policy can have low chance of success due to the high complexity of search space. The large number of possible edges and edge combinations in our graph makes learning an effective neural policy difficult as Deep Q Learning is normally trained with a small action space (*e.g.,* 2-16 actions). To address this, rather than choosing a random sample as the starting vertex and letting the model explore the entire graph with a randomly initialized policy, we do the following for each training:

1. Pre-generate a goal vertex using a sequence of random transformations on the current training sample. This sequence of transformations represents the *ideal sequence* of transformations for the current training input.
2. At each transformation step, randomly generate an action from one of three sources:
   - The ideal sequence - This is the pre-generated transformation sequence that acts as an answer key
   - The current neural policy - This is the policy learned by the reinforcement learning model.
   - The Random policy - This simply selects one of the available transformations based on the current input state.
3. Once an end state has been reached, use the exploration object (*e.g.,* classifier loss) and the ideal sequence to evaluate the fitness of the selected actions and update the neural policy.

The above process repeats for all inputs in the training set and can be looped multiple times. Furthermore, as training progresses, the actions generated in step 2 progressively become more likely to be from the current neural policy rather than the ideal sequence of the Random policy.

B Numerical Inputs

As the DGA classifier first extracts 20 numerical continuous features, we also tested our framework on numerical data. Specifically, we were interested in the success rate of our framework at generating adversarial feature vectors, similar to traditional adversarial attacks, but with some constraints on the transformation process. We selected 13 of the 20 numerical features that, when modified, were likely to be realizable if necessary (*e.g.,* relative number of consonants/vowels, number of dashes, length, etc.). Each transformer was only allowed to modify at most 30% of the feature's current value. As one of the features recorded the length of the domain name, we added an additional constraint to this transformer that the length could only be increased if modified. We present these results in Table 12. We observe an overall increase in success rate and decrease in average time per sample compared to Table 8. Note that we did not evaluate the Model Guided algorithm for this experiment.

Table 12: DGA experimental result - Generating Adversarial Features.

Algorithm	Success rate	Avg. # of Transforms	Avg. Time / sample
Beam Search (Random)	6%	1.87	0.010 s
Beam Search (Brute-Force)	65%	1.73	0.801 s
Beam Search (Lookup Table)	42%	2.02	0.069 s
Simulated Annealing	52%	2.96	1.000 s