Summoning Demons

The Pursuit of Exploitable Bugs in Machine Learning

Rock Stevens Octavian Suciu Andrew Ruef Sanghyun Hong Michael Hicks
Tudor Dumitras
University of Maryland, College Park

ABSTRACT

Governments and businesses increasingly rely on data analytics and machine learning (ML) for improving their competitive edge in areas such as consumer satisfaction, threat intelligence, decision making, and product efficiency. However, by cleverly corrupting a subset of data used as input to a target's ML algorithms, an adversary can perturb outcomes and compromise the effectiveness of ML technology. While prior work in the field of adversarial machine learning has studied the impact of input manipulation on correct ML algorithms, we consider the exploitation of bugs in ML implementations. In this paper, we characterize the attack surface of ML programs, and we show that malicious inputs exploiting implementation bugs enable strictly more powerful attacks than the classic adversarial machine learning techniques. We propose a semi-automated technique, called guided fuzzing, for exploring this attack surface and for discovering exploitable bugs in machine learning programs, in order to demonstrate the magnitude of this threat. As a result of our work, we responsibly disclosed five vulnerabilities, established three new CVE-IDs, and illuminated a common insecure practice across many machine learning systems. Finally, we outline several research directions for further understanding and mitigating this threat.

Keywords

Machine learning, vulnerability research, application security, vulnerability exploitation, fuzzing

1. INTRODUCTION

Governments and businesses increasingly employ data analytics to improve their competitive edge. For example, the United States Environmental Protection Agency has outlined its vision for leveraging machine learning (ML) to improve their everyday operations [12]. IBM offers businesses a platform for conducting sentiment analysis to gauge their effectiveness within a target audience [15]. OpenDNS uses ML to automate protection against known and emerging threats [25]. Machine learning allows these organizations to extrapolate trends from massive data sets, of often uncertain provenance.

However, ingesting unfiltered, public information into data analytic engines also introduces a threat, as miscreants can corrupt eventual inputs to ML algorithms to bias their outputs. Cretu et al. [7] discussed the importance of "casting out demons," or sanitizing the training datasets for safe machine learning ingestion. Research on *adversarial machine learning* [16, 19, 1, 3] has explored various attacks against ML *algorithms*, with a focus on skewing their outputs through malicious perturbations to the input data.

In this paper, we discuss another attack vector: ML algorithm *implementations*. Like all software, ML algorithm implementations have bugs and some of these bugs could affect learning tasks. Thus, attacks can construct malicious inputs to ML algorithm implementations that exploit these bugs. Indeed, such attacks can be more powerful than traditional adversarial machine learning techniques. For example, a memory corruption vulnerability could allow an adversary to corrupt the entire feature matrix, not just the entries that correspond to adversary-controlled inputs. More generally, bugs in the cost function, minimization algorithm, model representation, prediction or clustering steps, could allow an adversary to arbitrarily skew learning outcomes, to initiate a denial of service attack or to cause model divergence.

While considerable efforts have beed devoted to discovering software vulnerabilities and mitigating the impact of exploits, these generally focus on bugs that allow the adversary to subvert the targeted system, e.g. by executing arbitrary code or by achieving privilege escalation. In contrast, adversaries attacking an ML system are interested in bugs that allow them to induce mispredictions, misclustering, or to suppress outputs. Such logic bugs are difficult to discover using existing tools.

As a first step toward understanding and mitigating this threat, we characterize the attack surface of ML programs, which derives from a general architecture that many ML algorithms share, and we identify decision points whose outcome we may corrupt. We discuss how bugs around those decision points could be exploited and the potential outcomes of these exploits. We also propose a semi-automated technique called *guided fuzzing* for finding and exploiting ML

implementation bugs. We wrap important decision points from the ML architecture with instrumented code to convert a logical failure of the algorithm (e.g. misprediction) into a crash that can be detected by a fuzzing tool [20], which generates test cases and records program exceptions on these inputs. We then apply a coverage-based fuzzing tool, American Fuzzy Lop [36], to *summon demons*, i.e. to automatically discover inputs that mislead the ML algorithms by exploiting bugs in their implementation.

We utilize this technique to discover attacks against OpenCV [4] and Malheur [31], two open source ML implementations. As an example, we started fuzzing with a seed image that was recognized as having a face by OpenCV. We added a logic branch that crashed on non-recognition. Guided fuzzing then proceeded to generate a mutant input that was clearly still a face, and yet was not recognized. This exploit relies on a bug in the rendering library used by OpenCV, which allows for incorrect rendering of input images. In total, we found seven bugs: three in OpenCV, two in Malheur, one in Scikit-Learn, and one in `libarchive` (used by Malheur). Of these, three were assigned a CVE-ID; only one was not exploitable.

In summary, this paper makes three contributions. First, we explore the attack surface of ML implementations, as compared to ML algorithms, highlighting potential attack vectors and impact on various components within these systems. Second, we introduce a novel technique for exploiting ML bugs to corrupt classification outcomes and the data provided to ML systems from benign sources. This technique is possible through *guided fuzzing*, which expands upon existing fuzzing techniques for discovering bugs in software applications. Finally, we discover several new ML implementation bugs in important open-source software; our work has led to these bugs being patched.

2. PROBLEM STATEMENT

We consider an exploit a piece of code aiming to subvert the intended functionality of software. Limiting our scope to machine learning, an exploit would be designed with the goal of corrupting the outputs of programs or to inhibit their operation. Such exploitable bugs may be present either in the core implementation of the ML algorithm or in libraries used for feature extraction or model representation.

In terms of impact, we distinguish between four possible outcomes of successful exploits. First, an exploit that causes specific instances to be assigned an incorrect label achieves *mispredictions*. More generally, we use the term *divergence* to refer to attacks that succeeded in skewing the predictive model away from an otherwise converged state. Similarly, an exploit targeting a clustering algorithm may cause inputs to be placed in different clusters, resulting in *misclustering*. Because machine learning systems are often utilized as black boxes, it may be difficult to detect that the system has been compromised by using one of these exploits, as they typically have no other side effects besides skewing the learned model and cause the ML system to fail silently. Finally, an exploit may also result in *denial of service*, e.g. by stopping data ingestion prematurely or by crashing the application to prevent it from providing any output. While easier to detect, such exploits may render the system temporarily unusable.

In this paper, we address the problem of discovering exploitable vulnerabilities in machine learning *implementations*. The goals of our work are: (i) to provide a general ML architectural description, discussing possible attack vectors and their impact on different system components; (ii) to develop a semi-automated technique for discovering ML vulnerabilities by exploring this attack surface; and (iii) to demonstrate the magnitude of this threat by discussing several real vulnerabilities we discovered in popular ML systems.

Non-goals: We do not address limitations of machine learning algorithms (the area of study in adversarial machine learning). Instead, we aim to unearth implementation bugs as an orthogonal attack vector against ML systems. Additionally, we do not aim to develop a fully automated technique for identifying these bugs. Instead, by relying on guided code instrumentation and program output manipulation, we are able to bootstrap existing fuzzing tools in order to discover bugs.

2.1 Threat Model

We consider an adversary who aims to subvert the execution of machine learning algorithms by exploiting bugs in algorithm implementations. We assume that the adversary has access to the program's source code. We also assume that the adversary controls some of the program's inputs, but is unable to prevent benign users from providing additional inputs. These assumptions are realistic in many settings; for example, the machine learning techniques proposed for malware classification or clustering [33, 28, 10, 31] operate on inputs that come from many sources, including possible adversaries. Additionally, much ML software is open source, e.g., OpenCV [4] and Scikit-Learn [27].

In searching for exploitable bugs, the adversary does not pursue the usual goals of vulnerability exploitation, e.g., gaining control over remote hosts, achieving privilege escalation, escaping sandboxes, etc. Instead, the adversary's goal is to corrupt the outputs of machine learning programs using silent failures.

From a spectrum-of-control perspective, arbitrary code execution exploits represent the strongest means for achieving the adversary's goal, as such an exploit permits an adversary to manipulate all aspects of the target system. However, the adversary may achieve her goals with less powerful exploits, e.g., targeting memory corruption bugs that allow modifying data in memory but do not enable code execution or bugs that trigger loss of precision in floating point computations. Denial of service attacks represent the weakest control of targeted systems and may be conducted by inducing early termination of the ML processing. In some settings, the weaker attacks may be more attractive as they could allow the adversary to remain stealthy and bypass defense mechanisms.

3. ATTACKING ML IMPLEMENTATIONS

To begin exploring the vulnerabilities of machine learning applications, we must first understand their attack surface. Enumerating the components of ML applications that an attacker may target allows us to reason about where the bugs may be and what impact they may have. We then build on

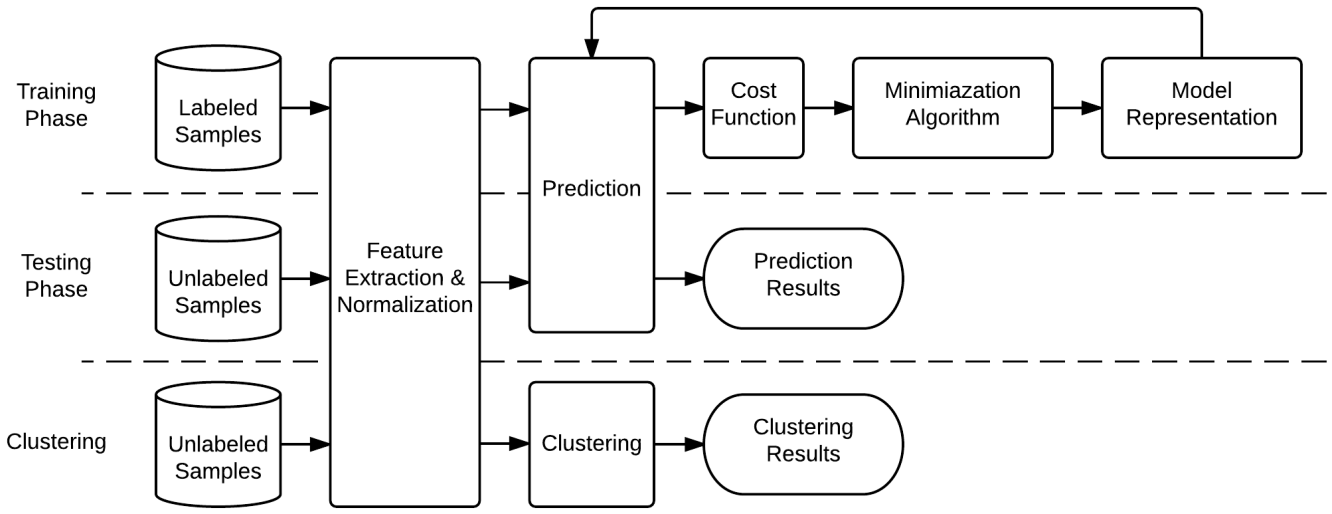

Figure 1: General architecture of ML systems.

this understanding and expand upon existing fuzzing techniques to create exploits for these bugs to induce misclassifications (false negatives or false positives), incorrect clustering results, and denial of service.

3.1 Machine Learning Architecture

Machine learning algorithms vary in structure and design. The two main categories of learning algorithms are supervised and unsupervised. In supervised learning, the algorithm receives a set of labeled examples and computes a predictive model that fits the training examples. The predictive model is then used to classify new, unlabeled samples. In contrast, unsupervised learning relies solely on unlabeled examples with the goal of finding clusters of similar samples. While there may not be a generic representation that fits all algorithms, the most popular supervised techniques are variations of *iterative minimization algorithms*[13]. For unsupervised learning, *clustering* is one of the most prevalent classes of algorithms. Figure 1 presents the general flow of a learning algorithm, highlighting the key particularities of each phase. In this setting, the input samples are transformed into a feature matrix representation that serves as the input to the classifier. A common, but optional, practice is to normalize the features prior to feeding them to the algorithm. This involves feature scaling and standardization. In the training phase, the (normalized) features are applied onto the current model in order to obtain the perceived prediction. The predictions are compared to the actual class labels using a cost function. The cost function output quantifies the distance between the current model and the ground truth. The model is then updated to reduce the cost through a minimization algorithm. This iterative process is repeated until the model becomes a sufficiently accurate representation of the ground truth. Upon convergence, the model is used to predict new class labels. In the testing phase, the unlabeled samples are transformed using the same feature extraction and normalization processes. The predicted class labels are obtained using the prediction function over the trained model. In clustering, the algorithm first performs feature extraction and normalization. Using a distance metric, the algorithm groups the samples into clusters that reflect the similarity between them.

3.2 From Architecture to Attack Surface

We now discuss how attacks on each component in Figure 1 may impact the overall functioning of the system. Table 1 summarizes the vectors and impact of attacks against the system components. A successful attack against one component may have ripple effects to others, either directly by transferring corrupted outputs to inputs, or indirectly via in-memory data structure corruption.

Feature extraction. Feature extraction is the backbone for the integrity of the system. Every attack from an external source must exploit vulnerabilities in this component as it is the sole communication port between the internal components and the external environment. An attack targeting the feature extraction component results in a corruption of the information passed downstream.

Within the feature extraction component itself, an attack can target the input parsing algorithms and/or the integrity checks performed on the feature representation. As shown in Section 4, such an exploit could result in memory corruption, arbitrary code execution, DoS or divergence. It is not always straightforward to define what is allowable input, or an allowable representation thereof. For example, in an image classification setting, a reasonable assumption would be to consider any renderable image as legitimate inputs. However, as detailed in Section 4.1, we found that most images that cause crashes in the OpenCV library are actually valid from a rendering perspective.

Prediction. Attacks are also possible against the prediction component, directly influencing the labels predicted by the algorithm. This could occur in both the training and the testing stages. For example, the attack could exploit bugs related to floating point overflow, floating point underflow, or the use of not-a-number (`NaN`) values. ML implementa-

Component	Exploitation Techniques	Impact
Feature extraction	Insufficient integrity checks	Misprediction, Memory corruption, Code execution, Divergence, DoS
Prediction	Overflow, Underflow, NaN, Loss of Precision	Misprediction, Divergence
Cost Function	Overflow, Underflow, NaN, Loss of Precision	Misprediction, DoS, Divergence
Minimization Algorithm	Overflow, Underflow, NaN, Loss of Precision	Misprediction, DoS, Divergence
Model Representation	Loss of Precision	Misprediction, Divergence
Clustering	Overflow, Underflow, NaN, Loss of Precision	Misclustering

Table 1: Attack surface of ML algorithms.

tions typically compute logarithms and square roots. This makes them particularly susceptible to bugs caused by NaNs (potentially the result of an overflow or insufficient consistency checks). The effects of a NaN propagate throughout the remainder of the computation and can result in a denial of service or model divergence.

Cost function and minimization algorithm. The cost function computation and the minimization algorithm are iteratively applied in the training phase. A bug could result in incorrect cost estimates or model updates that cause the model to diverge from the optimal value. Additionally, a denial of service could be obtained if the model update does not trigger the termination condition in the iterative algorithm. If the cost function consistently results in a NaN for the training examples, the minimization algorithm stagnates indefinitely without updating the model.

Model representation. The model representation could cause a model divergence through loss of precision. Since the training and the testing phases of algorithms are typically performed separately, the model has to be stored and transferred from the one to the other. As discussed in Section 4, casting between *float* and *long* types can skew the model away, resulting in inaccurate predictions for the unlabeled samples.

Clustering. In clustering, the algorithm itself or the distance metric can be manipulated using the same attack vectors as for the supervised learning. This could result in a denial of service or misclustering. In complete misclustering, the clusters are completely misrepresented, while in selective misclustering the attack might result in a particular sample being placed in a different cluster.

3.3 Discovery Methods

Fuzzing [20] is a popular method for bug discovery. A fuzzing tool tests a program using randomly generated inputs, which are often invalid or unexpected by the implementation, and records program exceptions or failures. In security, fuzzing has been employed to identify crashes that are indicative of memory safety errors in application. This technique has obvious applications to discovering one class of bug in machine learning systems—crashes—but can we use fuzzing to find bugs that silently corrupt the system's outputs? In this section, we use OpenCV as a running example while describing our bug discovery methodology.

Our use case is, in one sense, a natural fit for general purpose fuzzing because we can have a single program that runs on some input (i.e. an image) to produce some output (i.e. a text classification of that image). However, we have to ensure that we separate and identify both bug types of interest: crashes and silent corruption. To do this we introduce a technique we call *guided fuzzing*.

We use American Fuzzy Lop (AFL) [36] to instrument and fuzz-test machine learning programs. AFL was designed and is commonly used for finding crashes due to parsing failures, so the AFL loop involves running an application on multiple inputs and creating a report if an input causes a crash. AFL utilizes a genetic algorithm to generate inputs while maximizing the code coverage and has heuristics to discriminate between unique crashes and duplicates. We want to capitalize on AFL's ability to maximize code coverage while also finding crashing inputs.

A *guided fuzzing* workflow begins with a test case with a known outcome; for example, when analyzing OpenCV, we start with an image that contains a human face. The three outputs from the program under test might be: *crash*, *negative prediction*, (e.g. no face found) or *positive prediction* (e.g. face found). The default behavior with the initial test case is to find a face. Our fuzzing should mutate the image to change the output of the program under test to *negative prediction* while avoiding *crash*.

When we are searching for such logical failures, In this case, we do not care about inputs that produce crashes when OpenCV attempts to parse the image (although there is a disturbingly large number of these inputs). The first part of our *guided fuzzing* technique brackets the parsing regions of the program in a handler for the `SIGSEV` signal. The handler simply `exit`'s the program when a segmentation violation occurs. This prevents the crash and obscures it from AFL, which then believes that the application exited normally.

We then re-enable crashes in the application and check the outcome of important decision points in the ML algorithm. For example, we check the result of the face detection step, which corresponds to the outcome of the prediction phase from Figure 1. If the system failed to find a face, we induce a crash by manually dereferencing an invalid pointer. In this way, AFL recognizes when it has changed the output of the program to *no face found* without any change to AFL itself. Similarly, we can instrument the outputs of each of the components described in Section 3.2, to check for the presence of exploitable ML bugs.

4. RESULTS

We search for exploitable ML bugs in the OpenCV [4] computer vision library and in the Malheur [31] library for analyzing and clustering malware behavior. We select these

libraries because they are open source and they are widely adopted.

OpenCV provides its users with a common framework for computer vision applications and can process still imagery, live streaming video, and previously recorded video clips. For example, businesses can use computer vision and machine learning to reinforce physical security systems [32]. In such a scenario, an adversary may wish to thwart physical security through attacking the machine learning application itself.

Malheur is a security tool that performs analysis on malware reports that were recorded within sandboxed environments. Malheur can cluster the reports to determine which samples likely belong to the same malware family; these malware reports can be raw text files or compressed file archives. Malheur relies on `libarchive` to extract the malware reports from the file archives. An adversary that desires to delay analysis of their malware may target Malheur through crafted file archives and corrupt in-memory data. Data corruption will cause misclustering and allow the adversary to accomplish their goal.

As a result of our research, we responsibly disclosed five vulnerabilities (to which three were assigned CVE-IDs). The `libarchive` and Malheur system maintainers patched two of the vulnerabilities; as of January 27, 2016, the OpenCV maintainers acknowledged three vulnerabilities and would address the issues in future releases. These vulnerabilities still exist in the current version of OpenCV. Table 2 summarizes the vulnerabilities we found and their impact.

4.1 Discovery Results

OpenCV. We discovered bugs in OpenCV's image processing library, and we identified various conditions under which a valid JPG would cause an algorithm to terminate. Two vulnerabilities (CVE-2016-1516 and CVE-2016-1517) exist in the *feature extraction / selection* portion of the ML attack surface in Figure 1 and cause memory corruption when freeing a matrix allocated for image processing. In both CVEs, heap corruptions overflow fields in the matrix object and allow illegal access to memory locations when matrix objects are deallocated. Many examples exist in which an adversary can exploit similar vulnerabilities in image processing code and achieve remote code execution on a victim's system [14, 8, 21]. Our third vulnerability exists in OpenCV's custom image rendering library. Its improper handling of file artifacts and partial rendering of particular JPG images prevent consistent image classification.

During the guided fuzzing phase, these vulnerabilities and inconsistencies served as the basis for crafting legitimate input images that evade facial recognition detection. When used as-is, these images induce denial-of-service (DoS) crashes against OpenCV. DoS crashes in such an application require an administrator or operator to manually intervene to bring the system back online. Proof-of-concept SQL injection attacks already exist against video monitoring software that law enforcement organizations use to read license plates and issuing fines [24]; one can understand the ramifications of a DoS attack against similar applications or even autonomous driving vehicles using similar computer vision software.

A potentially viable defense against these crash-inducing images starts with first filtering input based on a render-check using the Python Image Library (PIL) [6] and the code snippet in Listing 1:

Listing 1: Image render-check using PIL

```
from PIL import Image
def is_image_ok(filename):
    try:
        Image.open(filename).load()
        return True
    except:
        return False
```

Of the 3197 images we found that induce crashes in OpenCV, PIL only allows 7 images to bypass this filter, resulting in a 0.0022% false negative rate. Yahoo! Flickr's proprietary image rendering solution allows 6 crash-inducing images through. These crash-inducing images are publicly available for viewing.[1]

Malheur. We discovered a critical bug within `libarchive` as used by Malheur. This vulnerability was issued CVE-2016-1541 [9] and was patched in `libarchive 3.2.0` on May 1, 2016. This vulnerability affected *every version of Linux and OS X*, given that `libarchive` is pre-packaged in these operating systems for handling various file archives. This vulnerability would allow an attacker to achieve arbitrary code execution by exploiting the inconsistent handling of `.tar.gz` compressed archives. This vulnerability occurs within the *feature extraction* block within Figure 1, given that the function inherently relies upon `libarchive` for attaining data. Once an attacker achieves arbitrary code execution, they have unlimited influence over the classification of the ML application. This bug could trigger another bug in Malheur's feature matrix extraction/selection and was patched on March 6, 2016 [30]. Section 4.2 explores the impact of corrupting the feature matrix in greater depth.

4.2 Guided Fuzzing Results

OpenCV. An attacker can exploit OpenCV's inconsistent rendering of images to induce silent failures and thwart facial detection within the prediction block of the ML attack surface in Figure 1. To begin, AFL utilizes a seed image with a shoulder-up picture of a person. The source code snippet that performs facial recognition[2] is a prime candidate for injecting the the logic branch (Listing 2) which allows us to induce a crash when the picture of the face is no longer detected.

Listing 2: Logic branch injection for facial detection

```
if(faces.size() == 0) {
```

[1] https://www.flickr.com/gp/138669175@N07/L53K8e

[2] https://github.com/Itseez/opencv/blob/master/samples/cpp/facedetect.cpp#L202

Vulnerability	Application	CVE-ID	Exploited	Impact
Heap Corruption	OpenCV	CVE-2016-1516	✓	Arbitrary code execution via double_free
Inconsistent rendering	OpenCV	n/a	✓	Partial rendering of `JPG` files affects classification results
Heap Corruption	OpenCV	CVE-2016-1517	✓	Denial of service attack via corrupt_chunks and segfault
Heap Corruption	Malheur via libarchive	CVE-2016-1541	✓	Arbitrary code execution on all Linux and OS X systems via corrupted archive
Heap Corruption	Malheur	GitHub patch	✓	Memory corruption via unsafe bounds checking
Loss of precision	Malheur	n/a		Loss of precision results in mispredictions
Loss of precision	Scikit-Learn	n/a	✓	Loss of precision results in model divergence

Table 2: Summary of ML Hunter findings.

```
    *((int *) 0xdeadbea7) =
    0xdeadbeef;
}
```

After 10.1 million permutations of the seed image, AFL crafted Figure 2. This image is incorrectly rendered by OpenCV, as seen on the left, but is clearly renderable by Google Photos, as seen on the right. In five out of five trials, this method successfully recreated photos that exercise this rendering bug. As these images are correctly formatted JPEG files, they bypass the PIL render-check described in Listing 1. In contrast to existing techniques for crafting adversarial samples that evade detection [34, 3, 2, 19, 1], our *attack does not depend on the learned model and succeeds from the first attempt.* This represents a new attack vector against machine learning, illustrating how bugs in ML code can provide a substantial advantage to the attacker.

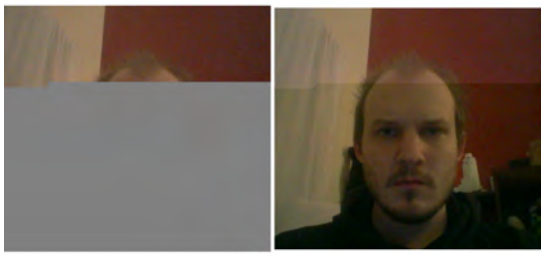

Figure 2: OpenCV incorrectly rendering a picture.

Malheur. Building on Malheur's inability to handle corrupted archive files, discussed previously, the guided fuzzing technique can corrupt Malheur's feature matrix and induce silent failures in prediction results. Thus, this vulnerability impacts all aspects of *clustering* within the generalized attack surface in Figure 1; an attacked can corrupt the in-memory data for unlabeled samples, tamper with the in-memory feature matrix, and affect the clustering results based on degree of induced skew. The vulnerable line of code[3] uses the variable `j` which is dependent on user-provided input. Thus, guided fuzzing can craft a corrupted archive file to traverse the heap and stomp over existing values in the feature matrix as shown in Listing 3.

[3]https://github.com/rieck/malheur/blob/master/src/fvec.c#L382!

Listing 3: Example of directed heap corruption

```
if (((unsigned long)&t[j -1] >
(unsigned long)&fv->val[0]) &&
((unsigned long)&t[j-1] <
(unsigned long)
((unsigned long)&fv->val[0]+
(unsigned long)fv->mem))){
    *((int *) 0xdeadbea7) =
    0xdeadbeef;
}
```

As this is a heap corruption vulnerability, we performed our proof-of-concept (PoC) exploit with address space layout randomization turned off. An attacker can couple our PoC exploit with ASLR bypass [11] techniques using another information disclosure exploit to find the desired offset. Additionally, our exploit uses a file archive; the exploitation success varies among operating systems and architectures as expected.

Expanding upon this example, an adversary can inject additional logic branches to control the degree in which the corrupted file impacts the feature matrix. Given enough time, AFL can generate inputs that increasingly skew the clustering of benign files the adversary did not craft.

This represents a second attack vector that provides new capabilities for adversaries. Unlike prior attacks proposed in the adversarial machine learning literature, this attack introduces the ability to *manipulate the in-memory representation of inputs not provided by the adversary.* From an adversarial perspective, this attack provides the opportunity to miscluster benign samples, or other malicious samples, to obfuscate the attacker's own malicious sample. An adversary can achieve this by inducing false negatives (more stealthy and desired) or false positives (junk reports). Again, this attack requires only one malicious sample and succeeds from the first attempt, owing to the bug. The `libarchive 3.2.0` and Malheur GitHub patch rendered this bug unexploitable, as corrupted archives are rejected on ingest.

We also discovered a bug in Malheur that we could not exploit. During feature normalization in Malheur, both functions[4] `fvec_norm1()` and `fvec_norm2()` return a value of type `double` but it is then normalized to a `float`. Using guided fuzzing, an attacker can discover instances where type casting from `double` to `float` yields a discrepancy.

Listing 4: Example of discovering precision loss

```
if (abs((double) (f->val[i] / s) -
(float) (f->val[i] / s)) > epsilon){
    *((int *) 0xdeadbea7) = 0xdeadbeef;
}
```

The `epsilon` in Listing 4 represents the loss of precision an adversary wishes to induce. In this particular instance, guided fuzzing did not discover any value of `epsilon` that caused misclustering. While our attack was unsuccessful, this approach could be a viable attack vector elsewhere.

We discovered that such a vulnerability is present within the Scikit-Learn machine learning library for Python and its underlying reliance on NumPy. When defining `ndarray` objects from Python lists without explicitly specifying a data type, the library infers the resulting data type according to undocumented heuristics. The NumPy arrays are used by Scikit-learn during both training and testing of the Linear Regression algorithm. This attack forces NumPy to set the `ndarray` data type as `object`, which preserves the underlying data type of each element. The Scikit-learn sanity checks ensure that the training and testing data types match. Because both the training and the testing arrays are of type *object*, the arrays pass the Scikit-learn checks. Our proof-of-concept (PoC) code places Python `float` and `long` values in the arrays before that data is ingested by the Scikit-learn module. When the input numbers are very large, this results in a loss of precision from casting. Specifically, the PoC shows how the regression model coefficients are drastically changed when using `float` instead of `long` in the training dataset; this results in a *diverged* model and inaccurate predictions. In absence of a fuzzing tool for python, we discovered bug by manually inspecting the Scikit source code, guided by the attack surface guidelines. These bugs illustrate a third attack vector that potentially enables new adversarial capabilities.

5. RELATED WORK

This section presents prior work on fuzzing and adversarial machine learning. Adversarial machine learning research focuses on crafting adversarial samples. The key distinction in our work is that we exploit bugs in machine learning code that give the adversary an advantage in conducting these attacks.

Insufficient input sanitization is a common cause of exploitable bugs [17]. Fuzzing is an automated technique that allows developers to test how gracefully their application handle various valid and invalid input [20, 23]. Fuzzers assist developers with isolating potentially buggy code and can play a critical role in identifying locations in need of input sanitization.

[4]https://github.com/rieck/malheur/blob/75ffd2498e964aa7d09782bf5a0d31afde36585f/src/fmath.c#L37

The field of adversarial machine learning has developed several methods for attacking ML systems, typically by querying ML models. Barreno et al. [1] proposed a general classification system for these attacks. Integrity attacks allow hostile input into a system and availability attacks prevent benign input from entering a system. Concept drift [35] is a phenomenon that occurs within machine learning systems as the prediction becomes less accurate over time due to unforeseen changes. Identifying concept drift, whether sudden or gradual, can be difficult in the presence of noise. Ideally, machine learning systems should combine robustness to noise and sensitivity to concept drift. Adversarial drift [19] describes intentionally induced concept drift in an effort to decrease the classification accuracy. Biggio et al. [3] described a threat model in which an attacker desires conceal malicious input in an effort to evade detection without negatively impacting the classification of legitimate samples. According to Biggio, an attacker may wish to inject malicious input to subvert the clustering process, rendering the resulting knowledge useless. The adversarial classifier reverse engineering [16] describes techniques for learning sufficient information about a classifier to instrument adversarial attacks. This information provides attackers and defenders with an understanding of how susceptible their system is to external adversarial influence. Newsome et al. [22] introduce a delusive adversary that provides malicious input in an attempt to obstruct the ML training phase; the attacker assumes full control over the input and its order.

In an analysis of a neural network trained for image processing tasks, Szegedy et al. [34] identified that an adversary can apply a perturbation to an image that is imperceptible to humans yet it changes the network's prediction. Research from Cha et al. [5] explores automated generation of such perturbations. Utilizing a well-formed seed input, a mutational fuzzer iteratively manipulates the seed to achieve maximum path traversal in a target program. This technique can isolate particular sets of input that cause the program to enter a state that might be of interest for an attacker.

Cretu et al. [7] discuss the process and importance of "casting out demons," sanitizing ML training datasets for anomaly detection (AD) sensors. AD systems inherently receive malicious input and anomalous events that may drastically impact the system's tuning and instrumentation. Accounting for data that may negatively impact the accuracy of the system's classifier can enhance its overall robustness.

6. DISCUSSION AND FUTURE WORK

In this paper, we focus on attacks against machine learning systems. However, our threat model has a broader applicability. For example, nation-state adversaries might be interested in attacking long-running simulations on supercomputers, with the aim of subtly skewing their results. In high performance computing, outputs are generally difficult to validate and expensive to re-compute, so it is difficult to defend against such attacks. Other data analytics systems may also be susceptible to such attacks.

For some of the bugs that we discovered, it is unclear who is responsible for fixing them. Should the Malheur maintainers have to worry about bugs in `libarchive` in order to preserve the integrity of their application? Should the architects of

OpenCV sacrifice performance for the sake of handling invalid input that developers did not filter? As more and more everyday devices begin to incorporate ML processing, this ambiguity must be explicitly resolved in order to provide secure systems.

Section 3 describes a semi-automated approach for discovering bugs in machine learning platforms through categorizing the backtrace of crash-inducing results. Tools such as `!exploitable` [18] provide researchers with automated crash analysis and the likelihood that the crash is exploitable. Overlaying the findings from such a tool on top of our generalized attack surface could expedite the discovery phase.

Section 4 explores many techniques that, at first glance, are only feasible because the targeted source code is publicly available. Recently, Papernot et al. [26] proposed model extraction attacks, by building surrogate classifiers that approximate black-box ML models. A logical next step in expanding our research would be understanding the overlap between building substitution models of proprietary classifiers and unique edge cases that result in bugs in the black box system.

An adversary discovering the possibility of "linchpin values" that appear during feature matrix construction would be another decisive shift towards an attacker's influence on ML systems. Linchpin values are consistent ranges of values within a feature matrix, that when present, result in a specific classification. Ribeiro et al. [29] proposed a technique for model explanation by building locally optimal classifiers around points of interest. Building upon their work, researchers may apply various analytic techniques to determine if there are common values or thresholds within a feature matrix that, when present, always result in a certain classification. With this information, an attacker could use guided fuzzing to craft arbitrary input that would guarantee a misclassification in the targeted system.

7. CONCLUSIONS

Entities that choose to trust data from unvetted sources subject themselves to a plethora of potential attacks in which a miscreant only requires minimal control over the entire dataset. For an attacker that wishes to control the decision-making process of its competitors or adversaries, this represents a powerful paradigm shift in attack vectors. We discovered several vulnerabilities within OpenCV and Malheur that allow an attacker to exploit bugs in underlying dependencies and the applications themselves to gain a marked advantage in influencing or out-right controlling the output of ML applications.

8. REFERENCES

[1] M. Barreno, B. Nelson, A. D. Joseph, and J. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[2] B. Biggio, B. Nelson, and P. Laskov. Support vector machines under adversarial label noise. In *ACML*, pages 97–112, 2011.

[3] B. Biggio, I. Pillai, S. Rota Bul'o, D. Ariu, M. Pelillo, and F. Roli. Is data clustering in adversarial settings secure? In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 87–98. ACM, 2013.

[4] G. Bradski. OpenCV. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. 2015.

[6] A. Clark. Python PILLOW, 2015.

[7] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 81–95. IEEE, 2008.

[8] CVEdetails. CVE-2015-4493: Heap-based buffer overflow in the stagefright::ESDS::parseESDescriptor function in libstagefright in mozilla firefox bef, 2015.

[9] CVEdetails. Vulnerability note VU#862384, 2016.

[10] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 3422–3426, 2013.

[11] T. Durden. Bypassing PAX ASLR protection. *Phrack Magazine*, 59(9):9–9, 2002.

[12] Environmental Protection Agency. EPA's cross-agency data analytics and visualization program | toxics release inventory (TRI) program | us epa, 2015.

[13] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[14] Google Project Zero. Project zero: Hack the galaxy: Hunting bugs in the samsung galaxy s6 edge, 2015.

[15] IBM. IBM social sentiment analysis powered by IBM analytics – india, 2015.

[16] D. Lowd and C. Meek. Adversarial learning. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 641–647. ACM, 2005.

[17] G. McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[18] Microsoft. !exploitable crash analyzer - msec debugger extensions, 2016.

[19] B. Miller, A. Kantchelian, S. Afroz, R. Bachwani, E. Dauber, L. Huang, M. C. Tschantz, A. D. Joseph, and J. Tygar. Adversarial active learning. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 3–14. ACM, 2014.

[20] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[21] National Institute of Standards and Technology. Nvd - detail, 2015.

[22] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *Recent advances in intrusion detection*, pages 81–105. Springer, 2006.

[23] P. Oehlert. Violating assumptions with fuzzing. *Security & Privacy, IEEE*, 3(2):58–62, 2005.

[24] G. Ollmann. SQL Injection in the Wild, 2013.

[25] OpenDNS. Cyber threat intelligence | OpenDNS, 2015.

[26] N. Papernot, P. D. McDaniel, I. J. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against deep learning systems using adversarial examples. *CoRR*, abs/1602.02697, 2016.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[28] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 391–404, 2010.

[29] M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?": Explaining the predictions of any classifier. *arXiv preprint arXiv:1602.04938*, 2016.

[30] K. Rieck. Fix for problem with corrupt archives., 2016.

[31] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(3), 2011.

[32] J. Sandhu. Machine Learning for Smart Home Security Systems, 2016.

[33] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*, pages 38–49, 2001.

[34] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[35] A. Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106, 2004.

[36] M. Zalewski. American Fuzzy Lop, 2015.