Kipple: Towards accessible, robust malware classification

Andy Applebaum*

The MITRE Corporation†

Abstract

The past few decades have shown that machine learning (ML) can be a powerful tool for static malware detection, with papers today still purporting to eek out slight accuracy improvements. At the same time, researchers have noted that ML-based classifiers are susceptible to *adversarial ML*, whereby attackers can exploit underlying weaknesses in ML techniques to specifically tailor their malware to evade these classifiers. Defending against these kinds of attacks has proven challenging, particularly for those not steeped in the field.

To help tighten this gap, we have developed *Kipple*, a Windows Portable Executable (PE) malware classifier designed to detect attempts at evasion. Kipple uses a portfolio of classifiers, building on a primary classifier designed to detect "normal" malware by attaching classifiers designed to specific types of adversarial malware to it. While simplistic, this approach is able to make it significantly harder – though not impossible – to bypass the primary classifier.

This paper reports on our process developing Kipple, specifically highlighting our methodology and several notable conclusions, including how our ensemble approach outperforms one using simple adversarial retraining and other performance notes. Our hope with publishing this paper is to provide an example defense against adversarial malware, and to also more broadly make the field more accessible to newcomers; towards this larger goal, we include a set of "lessons learned" for newcomers to the field, as well as an open-source software release containing Kipple's models, the data it was built from, and various scripts used to help generate it.

1 Introduction

Static detection of malware files is an ongoing area of research within the security community. While traditionally an "old-school" approach, static detection offers many advantages, principally that by scanning files to see if they're malicious we can quarantine them before they execute and run malicious commands. Early malware detection approaches were simple, originally keying off of known details about a file, such as its MD5 hash or known malicious strings held in its binary representation. Today, more modern approaches use various machine learning (ML) approaches to perform more advanced static detection.

Unfortunately, ML algorithms and technologies can be susceptible to *adversarial examples*, wherein an adversary perturbs an input object in a way that keeps the object semantically consistent (i.e., it is still *the same*) but the classifier is no longer able to properly identify it. Much of the research in this space has focused on computer vision as the exemplar domain, but the research community has, over the past few years, also looked at how it can apply to malware detection. For example:

*aapplebaum@mitre.org

†The author's affiliation with The MITRE Corporation is provided for identification purposes only and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions, or viewpoints expressed by the author. Approved for Public Release; Distribution Unlimited. Public Release Case Number 21-3931.

Proceedings of the Conference on Applied Machine Learning for Information Security, 2021

- Rosenberg et al. [18] propose to generate adversarial malware by using explainable artificial intelligence (AI), identifying what features should be perturbed to make a sample evasive.
- Li et al. [13] generate adversarial Android malware by using generative adversarial networks (GANs), bypassing previously proposed defenses.
- Hu and Tan [11] show how to attack a recurrent neural-network (RNN) based malware classifier by substituting a victim classifier and generating adversarial examples against it.

Many others abound in the literature as well – including many with accompanying open source code.

Unfortunately, defenses against these attacks are limited: both conceptually, as in there not being any clear way to defend against adversarial example attacks, but also practically, in that there are more papers purporting to attack models than to defend them. Moreover, in browsing the open source availability, we find that there's been little code – or models – published to help defenders secure their systems against these kinds of attacks. This is problematic, and shows the asymmetric capabilities around and/or interest in defending ML-based classifiers.

In this paper, we seek to bridge this gap, walking through our own attempts to build a robust malware classifier we term "Kipple". While our approach is imperfect, we believe that increasing the amount of discourse in this space, publishing lessons learned for newcomers, and highlighting potential areas for future research, we can help advance the state of the art in defending against evasive malware. Additionally, we have publicly released our code, data, and models online: https://github.com/aapplebaum/kipple.

2 Overview

Kipple originated as a small personal project entered into the 2021 Machine Learning Security Evasion Competition (MLSEC)[1]. The competition features a "defender" and an "attacker" track, where in the former participants submit ML models designed to detect malicious Windows Portable Executable (PE) files, and in the latter participants seek to evade the user-submitted classifiers by leveraging obfuscation and adversarial ML techniques. For submission into the defender track, the goal is to build a classifier that achieves no more than 1% false positive rate with no more than 10% false negative rate on normal malware, trying to minimize the number of evasions in the second track.

Because Kipple was implemented as a side project, the initial effort was small, running only on a personal computer and having the ultimate goal of merely submitting an entry regardless of its performance. Originally the impetus for Kipple was to leverage ideas from differential privacy as a defense against adversarial examples [15], but this early approach was dropped after poor performance testing against normal malware[2].

Instead, we adopted a more straightforward hypothesis: many adversarial malware construction techniques are easily distinguishable from traditional benign *and* malicious Windows PE files, and if we create an ensemble classifier consisting of a traditional malware classifier and a set of classifiers designed to detect adversarial malware, we can greatly increase the efficacy of the initial classifier. Doing this, we need not specifically evade *our* initial classifier, but rather generate many instances of the adversarially generated examples and train a new classifier on those.

To implement this, our approach worked as follows:

1. Create an initial classifier designed to detect "normal" malware;
2. Create a set of adversarial malware, using a variety of approaches and resources to do so;

[1] https://mlsec.io/

[2] A longer write-up on this approach and its shortcomings is left for future work.

Table 1: Number of samples per data source for existing/external data sets.

Source	Format	Label	Training Size	Testing Size	Count
Ember	Feature Vector	Malware	300000	100000	400000
VirusShare	Binaries	Malware	N/A	N/A	7662
SoReL	Binaries	Malware	N/A	N/A	31914
MLSEC 2019	Binaries	Malware	N/A	50	50
MLSEC 2020	Binaries	Malware	N/A	50	50
MLSEC 2021	Binaries	Malware	N/A	50	50
Ember	Feature Vector	Unknown	200000	N/A	200000
Ember	Feature Vector	Benign	300000	100000	400000
Local	Binaries	Benign	2191	379	2570
MLSEC 2019	Binaries	Adversarial	N/A	544	544

3. Produce a separate classifier for each set of adversarial malware; and

4. Create an ensemble classifier leveraging the initial classifier and each adversarial classifier.

In the remainder of this paper, we walk through our process for each step, starting with data creation and identification, building and testing the initial classifier, and then creating the separate and ensemble classifiers. As part of this last step, we walk through the different performance results for each of the different options we investigated for our ensemble.

3 Data

This section covers the data used to generate and test Kipple, breaking it down by source. Broadly speaking, we consider two primary classes for data: samples obtained from external sources – termed "original samples" – and samples generated as part of adversarial example construction, frequently referred to as "variants". Within both sets, most of the samples were known to be either benign or malicious, although a small portion of the original sample data was unlabeled.

The rest of this section discusses the source, count, and methodology for generating the data, with Table 1 providing summary statistics for the normal/original/non-adversarial samples, and Table 2 providing summary statistics for the adversarially generated samples/variants.

3.1 Original Samples

We leverage the 2018 EMBER[3] data set [1] as the core data for our approach. This data includes feature vectors for 1 million PE files scanned in or before 2018, categorized into if the file was benign, malicious, or unknown. The EMBER data set provides a solid base to build a classifier from, but because it is provided as feature vectors and not as binaries themselves, our ability to construct a more robust classifier from the EMBER data is limited. To remediate this, we look to external sources to provide binaries which can be used to construct adversarial malware.

3.1.1 VirusShare

We first investigated the free VirusShare platform[4] as a source for original malware samples. While the platform offers bulk downloads, due to size limitations – discussed further in Section 7

[3] https://github.com/elastic/ember
[4] https://virusshare.com/

– we opted to use the website's API to obtain individual samples one at a time. This option required us to specify an MD5 or SHA256 value for the sample we wanted; to get a list of relevant hashes, we consulted the EMBER data set, resulting in us downloading the actual samples referenced by EMBER. Unfortunately, the API limits the number of requests you can make to one malware sample per 15 seconds; ultimately, after a period of a few weeks we were able to obtain roughly 7500 samples in total.

3.1.2 SoReL

The SoReL[5] data set [10] contains the features and metadata of roughly 20 million different PE files, half of which are malware and the other half benign. In addition to feature vectors, SoReL also provides several pretrained models, and most importantly for our work, access to the roughly 10 million malware instances for download. We do note, however, that as a precaution the SoReL authors modified the hosted binaries to be non-executable, potentially altering each instance's representation and features.

Unfortunately, due to local space constraints, downloading the feature vectors provided by SoReL was prohibitive – when compressed, the features take up roughly 72GB of disk space, with the malware instances themselves requiring 8TB of space. While the features themselves are not particularly prohibitive for consumer hardware, design decisions made when provisioning the Kipple training machine prevented its download at the time; moreover, training such a model that makes use of all of the SoReL data could prove difficult using consumer hardware.

Instead, we took a simple approach of randomly downloading malware instances one at a time. Through this process, we were about to acquire roughly 32000 malware instances, with delays from self-imposed and external factors including rate-limiting and download speeds.

3.1.3 Local Files

We were able to pool together roughly 2600 benign files from the author's personal computer that were accumulated over a period of ten or so years. These files would be used primarily for developing adversarial examples later in the next section, and were not used for actual training; the split between training and test here reflecting which files were used for adversarial example construction.

3.1.4 MLSEC Data

As an entry into MLSEC 2021, we obtained three small labeled data sets:

- MLSEC 2019 data, including 50 normal malware instances used during the 2019 challenge, and 545 malware variants submitted by contestants during the competition.
- MLSEC 2020 data, including 50 normal malware instances used during the 2020 challenge.
- MLSEC 2021 data, including 50 normal malware instances used for the 2021 challenge.

None of these files were used for training, with the 2019 data used as the primary metric for accuracy for Kipple during development.

3.2 Adversarially Generated Samples

We took three different approaches to generate adversarial malware samples; for two of these three, we leveraged public libraries designed to create adversarial examples against ML-based malware classifiers. For the third, we leveraged prior research looking at evading ML-classifiers using more traditional obfuscation techniques.

[5] https://github.com/sophos-ai/SOREL-20M

Table 2: Number of samples per data source and generation technique for adversarial malware. Note that all samples are binaries and malicious.

Source	Generation Technique	Training Size	Testing Size	Count
SoReL	MalwareRL	37053	500	37553
SoReL	GAMMA	4651	516	5167
SoReL	DOS Manipulation	2331	259	2590
SoReL	Small Pad	125	100	225
SoReL	Large Pad	177	100	277
VirusShare	MalwareRL	24081	500	24581
VirusShare	GAMMA	5066	563	5629
VirusShare	DOS Manipulation	2533	281	2814
VirusShare	Small Pad	2112	235	2347
VirusShare	Large Pad	2533	282	2815
msfvenom	No Added Code	5296	588	5884
msfvenom	Added SoReL	32633	1000	33633
msfvenom	Added VirusShare	6852	762	7614
MLSEC 2019-2021	MalwareRL	0	1433	1433
MLSEC 2019-2021	SecML Malware	0	746	746

3.2.1 MalwareRL

Our first approach leverages the *MalwareRL* [4] open source software package[6]. This package is an extension of prior work in [2], which originally sought to create a framework to develop reinforcement learning-based agents that can modify malware to make it evade an ML-based classifier; the updated MalwareRL package contains much of the original functionality of the original package, updating it to use a newer version of Python and newer libraries, adding actions, and providing better maintenance.

The core idea of MalwareRL is that we can chain together small modifications to PE files in such a way as make them evasive, with the goal being to create intelligent agents that are able to efficiently select action chains. Out of the box, MalwareRL includes an implementation of 16 file-modification actions, an agent that makes decisions randomly, and an implementation of the MalConv [17] detection model as specified in [1]. During a trial, the MalwareRL environment will randomly select a malware instance and apply modifications as per the agent's decision-mechanism, stopping if too many modifications are made or if the modified sample evades the classifier.

We ran MalwareRL off and on over a period of a month using the included random agent and MalConv classifier. As input, we provided the VirusShare and SoReL binaries downloaded in the prior section. Additionally, we leveraged the local benign "train" data as input to several of MalwareRL's actions (e.g. – the *add_section_strings* action used strings from our local data). During each trial, if the agent was able to successfully evade the classifier, we would save the file. To prevent some over-fitting, we recorded at most about 10 different evasive variants per sample[7]. The final counts for each of the VirusShare and SoReL data sets are listed in Table 2 with "MalwareRL" as the generation technique. Note that due to project constraints, we did not save the list of modifications used for each trial.

[6] https://github.com/bfilar/malware_rl

[7] This cutoff changed throughout the project, both to be higher and to be lower, but hovered around 10.

Table 3: Parameters used for implant generation; variation based on architecture not shown.

Field	Parameters
Platform	Windows
Architectures	x86, x64
Encoders	none, xor, xor_dynamic, shikata_ga_nai
Encryption	none, aes256, base64, rc4, xor
Payload	shell/bind_tcp, meterpreter/reverse_tcp, meterpreter/bind_tcp
Template	Local benign train data
Added Code	None, VirusShare, SoReL

3.2.2 SecML Malware

Like MalwareRL, the *SecML Malware* [6] package is designed to provide functionality to create adversarial attacks against Windows PE malware detectors[8]. SecML Malware differs, however, in that its attacks are end-to-end and are not intended to be chained together, providing a variety of different white-box and black-box attacks. To use the library, users need only provide the input malware and some benign samples to build from, with the package able to construct adversarial examples from there. By default, SecML Malware attacks an included instance of MalConv similar to MalwareRL.

We leverage three classes of attacks from the package:

- GAMMA attacks [8], including both the shift and section injection attacks;
- DOS manipulation [7], defaulting to partial as opposed to full; and
- Padding attacks [12], including a "smaller" version that pads a random number of bytes less than 100, and a "larger" version that pads a random number of bytes less than 1000.

We ran SecML Malware off and on over a period of a month using the above attacks against the default MalConv classifier. As input, we provided the VirusShare and SoReL binaries, leveraging the local benign "train" data as goodware, similar to for MalwareRL. Unlike MalwareRL, we saved each resulting sample *regardless* of it was evasive or not. Anecdotally, we saw that the padding attacks were not producing effective results, and stopped generating examples later in our generation process.

3.2.3 msfvenom

MalwareRL and SecML Malware are both designed to modify PE files to make them evade classifiers; while these approaches are effective, we also wanted to investigate hardening our model with more obfuscated malware instances and techniques. Towards this, we decided to build on prior work in [3] where they created evasive implants by leveraging the open source *msfvenom* utility[9]. We followed a similar approach, randomly choosing parameters based on the values in Table 3, with our local benign training examples as templates, and embedding either no code, a VirusShare instance, or a SoReL instance of malware. While prior work suggested the use of templates would make the resulting implants evasive, we believed that including the malware instances as "added-code" would make it easier for the classifier to detect the implant, and also more robust from a training perspective.

[8] https://github.com/pralab/secml_malware
[9] https://www.offensive-security.com/metasploit-unleashed/msfvenom/

Table 4: Accuracy for the base model against original samples at a $\tilde{1}$% false positive rate.

Source	Label	Number Correct	Total Samples	Accuracy
EMBER Test	Benign	99019	100000	99.0%
Local	Benign	2509	2570	97.6%
EMBER Test	Malware	96486	100000	98.5%
VirusShare	Malware	7651	7662	99.9%
SoReL	Malware	28809	31914	90.3%
MLSEC 2019	Malware	50	50	100%
MLSEC 2020	Malware	49	50	98%
MLSEC 2021	Malware	50	50	100%
MLSEC 2019	Adversarial	293	544	53.8%
MLSEC MalwareRL	Adversarial	811	1433	56.6%
MLSEC SecML Malware	Adversarial	570	746	76.4%

3.3 Building Test Sets

We originally intended to split each data set generated in Section 3.2 into a set of samples used for "training" and a set of samples used for "testing," with the latter being the primary performance metric to evaluate each model. This split is illustrated in Table 2.

Unfortunately, in practice the split was challenging to execute: as it turns out, almost all new models trained on the adversarially generated variants scored 100% on the test data. After some analysis we saw that this was due to overlap between the test and training data: while no sample itself was present in both sets, it was often the case that an original sample had distinct but similar variants in each, and thus many classifiers were indirectly exposed to some of the testing data during the training phase.

To remediate this, we created two additional data sets used specifically for testing, represented in the last two rows of Table 2. Both data sets leveraged the MLSEC 2019, 2020, and 2021 normal malware samples as the starting malware; because none of the classifiers were trained on this data, there was no chance for pre-identification of the variants. In the first set, termed *MLSEC MalwareRL*, we leveraged the MalwareRL framework to conduct a small set of 1433 variants of the original MLSEC samples. In the second set, termed *MLSEC SecML Malware*, we leveraged the SecML Malware library to build a set of 746 variants, including the GAMMA, DOS manipulation, and padding approaches.

4 Testing and Building a Base Model

We built a base model following the most recent EMBER code and publication [16], using a gradient boosted decision tree (GBDT) implemented in LightGBM[10] (LGBM). For our base model, we only used the EMBER malicious and benign training sets, choosing parameters similar to that [16][11]. We fix the threshold for our model to 0.85, using the EMBER benign test data to match the 1% false positive rate required for submissions to the MLSEC defender track.

Table 4 shows the accuracy of our model for each of the original sample data sets. Looking at the benign data, we see that we hit 99% accuracy for the EMBER data as expected given our threshold, but only 97.6% for the local benign data; we suspect this slight drop is due to the nature of the local binaries, as some of them included author-written code and other academic

[10] https://lightgbm.readthedocs.io/en/latest/Python-Intro.html

[11] Unfortunately the exact configuration was lost, but we believe it is the same as the source on slide 16 but with a lower number of leaves.

Table 5: Accuracy for the base model against the adversarially generated samples.

Source	Generation Technique	Number Correct	Total Samples	Accuracy
SoReL	MalwareRL	22102	37553	58.9%
SoReL	GAMMA	3079	5167	59.6%
SoReL	DOS Manipulation	2312	2590	89.2%
SoReL	Small Pad	214	225	95.1%
SoReL	Large Pad	260	277	93.9%
VirusShare	MalwareRL	20512	24581	83.4%
VirusShare	GAMMA	4547	5629	80.8%
VirusShare	DOS Manipulation	2803	2814	99.6%
VirusShare	Small Pad	2337	2347	99.6%
VirusShare	Large Pad	2805	2815	99.6%
msfvenom	No Added Code	639	5884	10.9%
msfvenom	Added SoReL	7643	33633	22.7%
msfvenom	Added VirusShare	1847	7614	24.3%

files. Looking at malware, we can see the classifier has fairly strong results, correctly classifying 98.5% of the EMBER test malware, 99.9% of the VirusShare malware, and missing only one out of the 150 MLSEC base malware files.

The SoReL malware proves much more interesting: even without any obfuscation, the classifier only is able to correctly identify 90.3% of the instances. While this meets our 10% false negative goal, it still falls short as opposed to the other sets. We posit this may be due to the model being poor in general, the SoReL malware being particularly hard to classify (i.e., significantly different than the EMBER training data due to it being more recent), or the modifications made to the SoReL binaries to make them inactive causing them to appear as benign.

The last data sets are the adversarial samples. Here, the classifier performs poorly on both the MLSEC 2019 adversarial malware and the MLSEC MalwareRL variants, correctly identifying 53.8% of the former and 56.6% of the latter. While the classifier performs better on the MLSEC SecML Malware set – identifying 76.4% of the samples – the poor performance on the other sets indicates that while the base classifier may be strong against “normal" malware, it’s likely susceptible to adversarially crafted ones. We note that the better performance in the SecML Malware set is likely due to the ease in which it can identify some of the individual SecML Malware attacks.

Indeed, this conclusion is reinforced by looking at Table 5, which reports the accuracy of our base classifier against the new samples we generated in the previous section. First, looking at the bottom of the table, we can see that the base classifier is very weak against the msfvenom samples, only correctly classifying 10.9% of the ones generated without added code. Contrary to our expectations, we don’t see that adding code makes it much easier for the classifier to identify the implant, scoring only 22.7% accuracy when a SoReL sample is added, and 24.3% accuracy when a VirusShare sample is added. While highlighting a significant gap in our classifier, this does align with the conclusion from prior work suggesting the msfvenom template option was particularly effective as an evasion technique.

Moving up the table, we see the classifier perform reasonably well on the VirusShare examples, identifying the DOS Manipulation and both padding techniques nearly at nearly 100%. The GAMMA attack is the strongest of the five, with the classifier correctly identifying 80.8% of the instances, with the MalwareRL approach as second-strongest with 83.4% accuracy.

The SoReL examples prove more problematic for the classifier. First, for the MalwareRL case, we see it only correctly identify 58.9% of the samples – a much lower value than for VirusShare.

Similarly, GAMMA is just about nearly as effective, with the classifier only identifying 59.6% of the samples. The DOS Manipulation case is not as dire, with the classifier spotting 89.2% of the cases, and the two padding cases both show improvement over the prior three, although not nearly as much as for the VirusShare case.

One interesting trend apparent from looking at the SoReL and VirusShare cases is how much better the classifier performs on the latter. This can be explained by examining the methodology behind gathering the samples: the SoReL data were selected randomly, whereas the VirusShare data was selected by pulling hashes from the EMBER data. Thus, it's likely that the classifier was indeed trained on the *original* malware sample used as part of the submission, and thus might be more famliar with the adversarial one. If this is the case, it shows an interesting potential gap in detection – we would expect the VirusShare GAMMA and MalwareRL accuracies to be higher if indeed the classifier was trained on the underlying non-modifying malware; 80% accuracy on adversarial examples perturbed from data the classifier was trained on seems rather low.

As one last stray obseravation, we note that the MalwareRL and SecML Malware samples were all generated targeting the MalConv classifier, one that uses a completely different architecture than our base case, though does use the same training data. Still, looking at the SoReL case, we find it interesting that e.g. 58.9% of the MalwareRL samples were transferable from the MalConv classifier to our GBDT one. We believe that this ease of transferability can have implications for future ML-based detection approaches.

5 Model Development

In this section we discuss how we build on the base model to make it more robust to adversarial attacks, focusing on the different options tested and the performance of each. This includes a model using adversarial retraining, individual models built from the EMBER data set and the new adversarial samples, and different portfolio options that leverage multiple models. Table 6 shows the final results for each model.

5.1 Retraining

Our first defensive approach is to try *adversarial retraining*. In this approach, we leverage a GBDT similar to as for Section 4, with 1000 iterations, 1024 leaves, a bagging and feature fraction of 0.5, and a learning rate of 0.005. We then train the classifier with the following data:

- The original EMBER benign data as benign;
- The original EMBER unknown data as discarded;
- The original EMBER malware data as malicious; and
- All new adversarial samples as malicious.

Similar to as in Section 4, we set a cutoff to ensure only 1% false positive on the EMBER test data, using a threshold of .75.

The second row (labeled *Retrained*) within Table 6 shows the results for the retrained model. Here, we see some initiall loss of performance as opposed to the base model: we have a lower accuracy for the local benign data set, a lower accuracy for the EMBER test malware set, and a lower accuracy for the MLSEC malware data. However, we do see a pronounced increase of $\tilde{2}0\%$ accuracy across each of the 2019, MalwareRl, and SecML Malware adversarial test sets. Whether this increase is worth the small decrease – $\tilde{4}\%$ – in normal malware is likely application-dependent, though in most cases it would seem that the additional coverage is likely worth the small performance hit.

Table 6: Accuracy for each setup in Section 5, with each row representing a model and column representing a data set. Note that each row/model has an accuracy of 99% on the EMBER benign test data.

Row	Model Name/Composition	Benign	Malware		Adversarial MLSEC		
		Local	EMBER	MLSEC	2019	Mal-RL	SecML
1	Base	97.6%	98.5%	99.3%	53.8%	56.6%	76.4%
2	Retrained	78.0%	94.4%	96.7%	76.7%	84.0%	86.6%
3	adversarial-all	41.7%	4.3%	16.0%	52.6%	60.3%	47.9%
4	adversarial-benign	40.1%	53.8%	77.3%	86.6%	84.3%	88.6%
5	variants-all	95.3%	9.3%	43.3%	78.3%	89.9%	71.6%
6	variants-benign	95.5%	60.6%	87.3%	88.2%	91.0%	94.4%
7	msf-all	29.3%	0.4%	4%	8.1%	4.9%	15.8%
8	msf-benign	24.5%	6.7%	50.7%	20.4%	35.5%	59.4%
9	undetect-all	21.6%	0.4%	46.7%	15.6%	39.4%	55.0%
10	undetect-benign	72.3%	0.6%	4.0%	0.6%	2.6%	8.7%
11	Base, adversarial-all	41.7%	96.0%	100%	83.1%	66.4%	94.6%
12	Base, adversarial-benign	41.4%	96.0%	100%	86.0%	70.6%	96.2%
13	Base, variants-all, msf-all	37.5%	94.7%	96.7%	88.0%	84.9%	95.0%
14	Base, variants-all, msf-benign	36.4%	95.3%	98.7%	88.6%	84.2%	95.7%
15	Base, variants-benign, msf-all	37.5%	95.6%	100%	88.6%	78.3%	95.0%
16	Base, variants-benign, msf-benign	28.5%	93.9%	98.7%	90.8%	81.1%	95.2%
17	Base, variants-all, undetect-all	28.8%	92.5%	93.3%	91.7%	89.0%	95.9%
18	Base, variants-all, undetect-benign	70.5%	92.7%	92.0%	89.3%	85.2%	95.0%
19	Base, variants-benign, undetect-all	28.8%	95.6%	100%	91.0%	84.5%	99.3%
20	Base, variants-benign, undetect-benign	70.5%	95.7%	100%	88.6%	78.8%	97.2%

5.2 Individual Models

Our second approach is to develop a set of individual models that do not label the EMBER set the same as the adversarial set. We specifically choose one of two paradigms for these models:

- **All**. Under this approach, we consider *all* EMBER data as benign, and all new adversarial samples as malicious. This approach tests/assumes that our new variants will have different signatures as opposed to the existing malware and benign data.
- **Benign**. Under this approach, we only consider the benign EMBER data as benign, ignoring all others (i.e., malicious, unknown). This approach specifically tries to construct a classifier that distinguishes between benign files and adversarial malware, without any training on normal malware.

Within these two categories, we create four different variations on the set of training data we want to use for adversarial samples:

- **Adversarial**. This set includes *all* adversarial samples.
- **Variants**. This set includes all samples generated with MalwareRL and SecML Malware.
- **msf**. This set includes all of the msfvenom samples.
- **undetect**. This set only includes the msfvenom samples that were flagged as benign by the *Base* classifier in Section 4.

This results in 8 different models, each of which is trained as a GBDT, with the *adversarial* and *variants* cases using the same parameters as the *Retrained* model and the *msf* and *undetect* cases modifying the number of leaves to 512 and number of iterations to 500. We further fix

each model's threshold to match the minimum needed to score a 1% false positive rate on the EMBER test data.

Rows 3-10 in Table 6 provide the results for each of the 8 models. First, we note that for each of the *adversarial*, *variants*, and *msf* models, the "all" case significantly under-performs versus the "benign" case. This is to some extent to be expected under this setup, as the classifier is unaware of what normal malware looks like; this is perhaps most evident with the *adversarial-all* model, where the classifier only correctly identified 4.3% of the EMBER test malware data. Interestingly, the *undetect* category does not follow this pattern, with the *all* case outperforming the *benign* case. We posit this may be due to the *undetect* samples appearing more like benignware – since they evaded the *Base* classifier – and thus the *all* case better discriminates between normal PE files and PE files designed *specifically* to look benign.

Looking at the best case for each model type – focusing on "benign" for *adversarial*, *variants*, and *msf*, and on "all" for *undetect* – we see that *msf* and *undetect* perform the worst, performing quite poorly on the EMBER malware test set (6.7% for *msf* and 0.4% for *undetect*) and under-performing the *Base* classifier for each adversarial set. Both classifiers also score poorly with the local benign data ($\tilde{2}$0% accuracy each) likely due to some of the local benign files having some variations present in both the test and train data[12] This idea is corroborated by looking at the *variants* and *adversarial* local benign scores as well: *variants* correctly identifies 95.5% of them, and *adversarial* falls in the middle at 40.1%.

On the other end of the spectrum, *variants-benign* is clearly the strongest, scoring highest for EMBER malware, MLSEC malware, and even the adversarial MLSEC data. Indeed, this classifier by itself is almost able to hit the 10% false negative threshold for that data. As runner-up, *adversarial-benign* is a little less performant, having a lower accuracy across each data set but falling within a reasonable 2% of *variants-benign* on the adversarial MLSEC 2019 data.

5.3 Portfolio Approaches

Moving past individual models, we investigate whether we can use a "portfolio" of models, focusing on a few combinations. Here, we set a threshold for each model in the portfolio; if at least one model flags a file as malicious, we record malicious. If none classify it as malicious, then we score it as benign. To set each model's threshold, we look at all threshold combinations that ensure that the total false positive rate is 1%[13]; then, we find the best pair (or, if a portfolio of three models, the best triple) to maximize the detection rate for the MLSEC data, ensuring we the thresholds score at least 90% for the EMBER malware set. Rows 11 through 20 of Table 6 show the results for our portfolio arrangement.

We only test two portfolios of size two: the *Base* model along with either *adversarial-all* or *adversarial-benign*. These two portfolios score roughly equally, with the *benign* configuration slightly outperforming the *all* configuration for the adversarial MLSEC samples. Notably, both portfolios offer benefits over their singleton counterparts: versus the *Base* profile by itself, each portfolio option provides much stronger coverage of the adversarial MLSEC data, at a cost of only a few percentage points off of the EMBER malware set. However, the portfolio does struggle with the local benign data, likely due to the included *msfvenom* data. Against the *adversarial* singletons, the portfolios perform much better across the board, with only the *adversarial-benign* case scoring near the portfolio for the adversarial MLSEC data.

The last 8 rows of Table 6 correspond to the eight portfolios of size three that we tested. Here, each portfolio includes the *Base* profile along with a version of the *variants* model and either a version of the *msf* model or a version of the *undetect* model. Interestingly, we when combined into a portfolio, we now see the *all* and *benign* cases each more competitive with their respective counterparts, scoring roughly similar across categories for each. Indeed, most of the

[12] As an example, the same binary might be present in both but with different filenames.

[13] Nothing that a lower false positive rate would indicate a higher true positive rate.

size three portfolios seem roughly on par with each other, and all of them largely outperform the size two portfolios, the *Retrained* model, and the *Base* model.

That said, there are some differences: even in the size three portfolio set the *msf* cases (rows 13 through 16) are not as strong as the *undetect* cases (rows 17-20). Within the latter set, two options stick out as the strongest. First, against the adversarial samples, {*Base*, *variants-all*, *undetect-all*} would appear the strongest, performing best on the 2019 and Malware RL data sets, and within a small distance of the best score for the SecML Malware data set. More well-rounded however is the {*Base*, *variants-benign*, *undetect-all*} portfolio. This option performs better by a few percentage points on the normal EMBER and MLSEC sets and still is the second-highest scorer for the adversarial samples.

6 MLSEC 2021 Performance

In this section we discuss Kipple's performance in the MLSEC 2021 event, covering what the final submission looked like and taking a critical analysis towards its performance.

6.1 Submission

We ultimately entered "Kipple" into the 2021 MLSEC competition using the {*Base*, *variants-all*, *undetect-benign*} configuration; this decision was made somewhat arbitrarily and was due to some miscalculations on performance before submission. Additionally, we made several changes to make the model more performant.

First, in local testing, we realized that Kipple struggled to correctly classify some benign binaries; specifically the local ones similar to the ones used to build it. To hedge against this, we first raised the *undetect-benign* model within the portfolio to have a much higher threshold; this largely stopped it from misclassifying these files. Still, to make the approach even more effective, we hardcoded in the specific MD5 values of the binaries it was trained on, bypassing the model entirely and returning "benign" if a file was submitted that matched the hash.

Second, we leverage the MLSEC-provided code[14] implementing a "stateful" detection system as described by Chen et al. [5]. This model works by keeping a "history" of prior queries that were flagged as malware; when a new query is sent to the classifier, if it is identified as malicious, the classifier stores it and returns malicious. If it is instead identified as benign, it is compared to the history of prior malicious queries; if within a certain distance of an existing prior query, it flags the submission as malicious.

Our submission followed the default implementation, but differed in that it added a query to the history if and only if the *variants-all* model flagged it as malicious, ignoring whether the submission was flagged by *Base* or *undetect-benign*. While including the other models would likely have been more accurate, we did not have adequate time to test this approach and did not want to increase our false positive rate.

6.2 Results

Kipple ultimately took third in the defensive track, with a score of 231 bypasses; the winner [9] had 162 bypasses, second place [14] had 193, and fourth, fifth, and sixth place had 714, 734, and 1840 respectively. While an overall strong performance, Kipple did have several shortcomings.

First, looking at the results in Table 6, Kipple is not perfect: it still misses a good portion of the MalwareRL and SecML Malware attacks. Indeed, during the competition the authors inadvertantly evaded Kipple by using MalwareRL during the attacker track.

[14] https://github.com/Azure/2021-machine-learning-security-evasion-competition/blob/main/defender/defender/models/ember_model.py#L114

Second, and perhaps more importantly, Kipple has very little knowledge of *traditional* malware obfuscation techniques. In self-testing, the authors found that many off-the-shelf packers and crypters were able to sufficiently obfuscate the MLSEC samples as to bypass Kipple's detection. While some of these approaches used some novelty, many of them were simple open-source scripts and easy to run. In fact, the winners of the attacker track [9] – who also won the defender track – recognized this, and were able to score the most evasions using a custom dropper they developed, sans any specific adversarial ML techniques (though some of their action space did overlap with the techniques in MalwareRL).

7 Lessons Learned

As a smaller side-effort, we encountered and overcame many barriers in developing Kipple that those more unfamiliar with the field would likely also run into. As part of our goal to enrich the research community, we report on several of these roadblocks, how we overcame them, and tips for others looking to conduct similar research. While many of these may seem obvious at first glance, for those just getting their feet wet these lessons may not be immediately intuitive.

7.1 Space Requirements

Kipple was initially to be built on a small (<30GB) Linux virtual machine run on a personal computer. This *quickly* became problematic – between the large size of EMBER/massive size of SoReL, the needed Python/other libraries, and space requirements to save newly generated variants, 30GB became too small fast. While we ultimately increased the space allotted to the VM developing Kipple, some of the early decisions influenced the course of the project; and even once the VM's space was enlarged, we still found ourselves running out of storage. Ultimately Kipple was allocated roughly 300GB of space, which appears sufficient for most operations.

Recommendation. We recommend researchers start with at least 200GB of space if possible when conducting adversarial malware research, with more preferred. Additionally, using VMs in a cloud environment will likely make storage less of a problem as opposed to local hardware.

7.2 Time Allocation

Time played a key role in the development of Kipple: many of the ideas of the project were straightforward, but were hard to test due to needed computation time and other difficulties. Indeed, many issues compounded as we intended to deliver Kipple as part of the 2021 MLSEC event which had a strict submission deadline. Some example issues include:

- Long-running code or scripts containing typos or mistakes at the end;
- Long computation time(s) for building new classifiers and models;
- Needing more time to construct adversarial examples; and
- Rate limiting for downloading malware samples.

Ultimately we tried to mitigate these as best we could, but often many of experiments we ran for Kipple would need to be configured late in the evening and run overnight, with some days lost due to typos or other issues.

Recommendation. Understand that adversarial malware research may be time intensive. Attempt to parallelize your work and research whenever possible; strongly consider using cloud infrastructure if you can. Setup pipelines to run consecutive jobs and let the code run on its own when you aren't needed. Review your code for bugs before running, and consider letting your long jobs run for a few minutes before stepping away.

One other recommendation to help save on time is to avoid duplicative computation. As an example, when training a new model, we would always convert each adversarial sample into its feature vector during training, requiring a significant amount of time once we generated many samples and had multiple models to test. Eventually we developed a storage system where upon generation we would store the feature vector for a new adversarial sample, thereby bypassing the need to convert the sample for each of the models we wanted to train. This approach itself was time consuming to implement, but saved many hours of computation time further down the road.

7.3 Record Keeping

We tried many different implementations for Kipple, ranging from choosing different LGBM parameters to training on different data sets. Unfortunately, we ran into issues keeping records of some of our models, parameters, and other training information. This ended up slightly hurting our efforts, as in some cases we did not have exact information regarding a model we built a week before, or we needed to reproduce something due to a small code mix-up. Much of this was the result of fighting time and storage issues, where we would implement something quickly without following good practices; in these cases, even when the outcome was positive, we still had to invest more time figuring out how to fix our implementation than if we would have done it correctly the first time.

Recommendation. Good coding practices help: use version control when possible, label variables well in your code, and label all files/models/others to be descriptive. Keep track of all important information, and backup often.

7.4 Data Robustness

Perhaps the most surprising result from our experiments was the accuracy of each model on the test data sets we created for the adversarial samples. This came as a shock initially, first indicating that the models were extremely effective – however, we ultimately realized that the accuracy came as a result of training/testing overlap, where variants of individual samples wound up in both the training and testing sets, thereby allowing the classifier to "see" example test data while training.

Recommendation. Identify training and test sets early, and try to identify any overlap between the two early in the process. Make sure to include at least one data set that appears disjoint from the training data.

8 Discussion and Future Work

We close out this paper with a discussion of some of our major results as well as identifying different areas for future work. One immediate noteworthy result was the high rate of evasion for the adversarial examples produced in Section 3. Even though these were produced to be evasive against MalConv – a classifier using a very different architecture – they still were effective against our GBDT classifier in Section 4. The transferrability of adversarial examples is an interesting area of study, and we hope that these results can help inform future areas of research; anecdotally,

we found that many of the samples produced locally also transferred to the other models as part of the MLSEC attacker challenge.

Another notable result from Kipple is the effectiveness of the portfolio classifier versus the adversarially retrained one. While the difference is perhaps not terribly large in performance, the portfolio did seem to do convincingly better on the MLSEC data set. The final Kipple model, while not bulletproof, was successful in stopping a large amount of evasion attacks. We believe that a future approach could leverage more data and additional training modules to make the solution even more robust.

Finally, we note that one of our original hypotheses was that the existing benign *and* malware would be sufficiently different than our own generated adversarial samples to provide effective training data in building a new classifier. Interestingly we found evidence both for and against this hypothesis: for the *adversarial*, *variants*, and *msf* models it was clear that *benign* outperformed *all*, but for the *undetect* model our hypothesis was indeed true and *all* outperformed *benign*. We believe that this, as well as the efficacy of training on strictly *evasive* samples, is an interesting area for further research.

9 Code Availability

As part of this project, we have released all of the code behind Kipple online: https://github.com/aapplebaum/kipple. This includes code to train and evaluate models, the adversarial variants we generated, and the models themselves. We hope that this release will help encourage future research in this area.

References

[1] H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, Apr. 2018.

[2] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[3] Andy Applebaum. Trying to make meterpreter into an adversarial example. URL: https://www.camlis.org/2019/talks/applebaum, 2019.

[4] Bobby Filar. MalwareRL, 2021. URL https://github.com/bfilar/malware_rl.

[5] S. Chen, N. Carlini, and D. Wagner. Stateful detection of black-box adversarial attacks. In *Proceedings of the 1st ACM Workshop on Security and Privacy on Artificial Intelligence*, pages 30–39, 2020.

[6] L. Demetrio and B. Biggio. secml-malware: A python library for adversarial robustness evaluation of windows malware classifiers, 2021.

[7] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Alessandro. Explaining vulnerabilities of deep learning to adversarial malware binaries. In *ITASEC19*, volume 2315, 2019.

[8] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 2021.

[9] Fabrício Ceschin and Marcus Botacin. Adversarial Machine Learning, Malware Detection, and the 2021's MLSEC Competition, Sept. 2021.

[10] R. Harang and E. M. Rudd. Sorel-20m: A large scale benchmark dataset for malicious pe detection, 2020.

[11] W. Hu and Y. Tan. Black-box attacks against rnn based malware detection algorithms. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[12] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[13] H. Li, S. Zhou, W. Yuan, J. Li, and H. Leung. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal*, 14(1):653–656, 2019.

[14] A. Mosquera. amsqr at MLSEC-2021: Thwarting Adversarial Malware Evasion with a Defense-in-Depth, Sept. 2021. URL https://doi.org/10.5281/zenodo.5534783.

[15] Nicholas Carlini. On evaluating adversarial robustness. URL: https://www.camlis.org/2019/keynotes/carlini, 2019.

[16] Phil Roth. Ember improvements. URL: https://www.camlis.org/2019/talks/roth, 2019.

[17] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[18] I. Rosenberg, S. Meir, J. Berrebi, I. Gordon, G. Sicard, and E. O. David. Generating end-to-end adversarial examples for malware classifiers using explainability. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. IEEE, 2020.