Marvolo: Programmatic Data Augmentation for Deep Malware Detection

Mike Wong[1] (✉), Edward Raff[2], James Holt[3], and Ravi Netravali[1]

[1] Princeton University, Princeton, NJ, USA
{mikedwong, rnetravali}@cs.princeton.edu
[2] Booz Allen Hamilton, Annapolis Junction, MD, USA
Raff_Edward@bah.com
[3] Laboratory for Physical Sciences, College Park, MD, USA
holt@lps.umd.edu

Abstract. Data acquisition for ML-driven malware detection is challenging. While large commercial datasets exist, they are prohibitively expensive. On the other hand, an entity (e.g., a bank or government), may be targeted with unique malware, but the data samples available will never be sufficient to train a bespoke ML-based detector. While data augmentation has been a key component in improving deep learning models by providing requisite diversity for generalization, it has proven far more challenging for malware detection. The main challenges are that (1) determining the augmentations to make is not straightforward, (2) operations are on binaries rather than source code (which is not available), complicating correctness and understanding, and (3) labeling new files mandates expensive binary reverse engineering. We present Marvolo for creating realistic, semantics preserving transformations that mimic the code alterations made by malware authors in practice, allowing us to generate augmented data on raw binary files. This also enables Marvolo to safely propagate labels to newly-generated data. Across several malware datasets and recent ML-based detectors, Marvolo improves accuracy and AUC by up to 5% and 10% respectively, while boosting efficiency by 79x by avoiding redundant computation.

1 Introduction

Malware detection is a problem with real-world ramifications, and machine learning has been used in building malware detectors for decades which can be trained with large commercial datasets [12, 20] of malicious and benign binaries. Unfortunately, detection in the wild continues to fall short of expectations, with attacks regularly occurring [4]. The core issue is cost: large and comprehensive datasets generally require licensing costs that can reach $400k/year. Thus, it is often impractical to obtain sufficiently general and representative training datasets, and yet, these datasets govern the efficacy of these models. As a result, a victim may only be able to discover $1 - 50$ samples of a malware family [25]. Worse, targeted malware, such as banking and nation-state malware, which are

designed for a specific target for which data samples are limited makes detection even more difficult due to the lack of available training samples.

Data augmentation techniques have been proposed to mitigate these issues [24, 30], but they face several challenges that limit their utility. The main issue is that augmentation strategies are typically decoupled from the behavior of malware authors in the wild, and instead focus on random alterations to boost dataset heterogeneity. Further, they directly modify feature representations of raw binaries (since source code is unavailable), which further convolute the semantic understanding of the effects of those perturbations. This, in turn, also precludes correctness-preserving labeling of the newly created samples. Lastly, despite the focus on coarse alterations, the programmatic processing of binaries is costly, both resource- and time-wise.

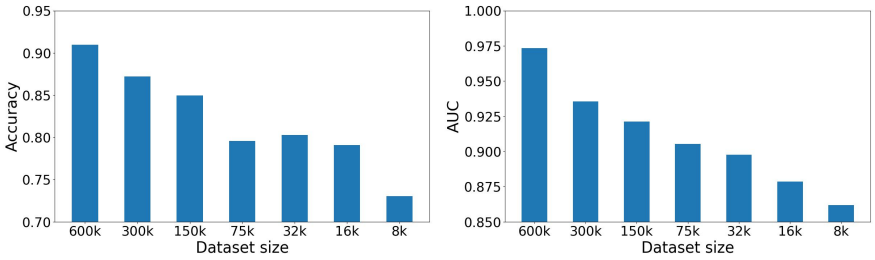
Analysis of malware over the years has revealed that malware authors typically use semantics preserving transformations [2, 48] to sidestep malware detectors and deter reverse engineering efforts. Our key insight is that the same observation can be used to enhance the efficacy of malware datasets through data augmentation. We introduce MARVOLO, a data augmentation engine for malware datasets. The key insight underlying MARVOLO is the use of semantics-preserving code transformations inspired by a study of real-world datasets we conducted in Section 3 that highlight the nature with which malware authors use code transformations. Building on this, MARVOLO embeds several key ideas. First, we use a 'lifter' to convert the files into a higher level representation, allowing us to perform code transformations on binaries and check for correctness. Second, we embed two complementary optimizations to collectively maximize the utility (i.e., number of realistic and diverse data samples) of the transformations within a time budget. Third, MARVOLO automatically labels newly-generated samples without mandating expensive binary reverse engineering.

We test MARVOLO using the state of the art MalConv2 [34] malware detector and multiple commercially-available large/small-scale datasets, i.e., the large-scale Ember [12] dataset, as well as a small-scale Brazilian dataset [16]. Overall, MARVOLO boosts detection accuracy by up to 5% and AUC by up to 10%, with most wins coming from detecting previously unseen novel families, which are intuitively more difficult to catch. MARVOLO also yields 2.35 – 3.8% higher accuracy and 8.4 – 9% higher AUC over prior augmentation approaches, which modify feature representations. Our optimizations provide a 79× speedup in contrast to the naive binary rewriting approach, making our approach tenable for generating large amounts of data samples. Further, we show that MARVOLO also yields accuracy and AUC improvements with non-deep baselines for detection.

We have open sourced MARVOLO at https://github.com/michaeldwong/marvolo.

2 Background and Related Work

Though prior attempts have been made in data augmentation for malware detection, they do not yet perform meaningful data augmentation. In [24, 30], programs are represented as sequences of opcodes and augmentation is performed



(a) Accuracy results.

(b) AUC results.

Fig. 1: Performance of MalConv2 [34] when training on different subsets of the Ember dataset [12]. All accuracy results were attained using a fixed threshold.

by replacing one opcode with another without necessarily preserving semantics. Further, [31] augments "images" generated from malware, which are known to be flawed representation [32]. These unrealistic augmentations exacerbate a common problem in malware research that labels are not always accurate [13]. In contrast to these efforts, MARVOLO's contributions lie in (1) a deep-dive analysis of large-scale malware datasets to uncover the usage patterns of semantics-preserving code transformations in malware, and (2) a system that leverages those insights to efficiently grow small datasets into larger ones with improved heterogeneity and realism that aid ML-based malware detection.

There exists a wide array of malware detection approaches with varying tradeoffs, ranging in amount of pre-processing done at prediction time from none (fast, less accurate) [9, 32] to full dynamic analysis (slow, more accurate) [23]. In this work we are focusing on small and incomprehensive datasets, where there is a need to triage files proactively especially when there is targeted malware. For this reason, signature-based methods are separate tools that capture what is known [38], where we still want a method to triage potential risk that are not known. In these situations it is common to use the probability of a classifier as a ranking for triage [29], meaning we often care about Area Under the Curve (AUC) as it corresponds to the quality of the detector at ranking correctly [33]. Put differently, the probability score from the classifier to rank is important for characterizing and ranking the files by maliciousness so that the most malicious files are identified and quarantined sooner rather than later. The goal of our modeling is thus to be good enough to triage for more expensive analysis (automated or human), as building an accurate detector standalone is not realistic given limited data.

We note that high quality labeled data is extraordinarily difficult to obtain for research purposes. The seminal EMBER [12] and SOREL-20M [20] require a VirusTotal license to obtain the original files, which can cost up to $400k/year. Consequently, groups must resort to far smaller datasets [42]. To show the importance of large datasets, we show the accuracy and AUC degradation of using progressively smaller Ember datasets in Figure 1. To contextualize these results,

Zenpak

```
inc eax
inc ecx
inc edx
inc ebx
inc esp
inc ebp
inc esi
inc edi
```

```
dec eax
dec ecx
dec edx
dec ebx
dec esp
dec ebp
dec esi
dec edi
```

Sivis

```
nop
nop
nop
xor eax, eax
inc ebx
dec ebx
inc ecx
dec ecx
```

```
inc eax
push edx
xor edx, edx
pop edx
inc eax
dec eax
cmp 0x17b8ef93, eax
jne 0x407033
```

Fig. 2: Code snippets from two malware families in the Ember dataset that exhibit semantics-preserving code transformations.

we note that the implications of detecting even a single additional malicious binary in the wild can be substantial, and that single-digit accuracy improvements are celebrated by malware analysts. For this reason our work uses only a subset of Ember with only several malware families as well as a Brazilian dataset [16], which will let us test the effectiveness of MARVOLO. This maintains relevance to our target use case as defenders can run honeypots to collect malware targeted at themselves [14]. Further background and related work is provided in Appendix A.

3 Approach

Our results from Section 2 highlight the inadequacies of small malware datasets relative to the large (commercial) datasets that have supported high accuracies for ML-driven malware detectors in practical settings. However, given the superior attainability of small datasets, our main goal is to determine whether they can be altered to more closely mimic the properties of their larger counterparts and deliver similar efficacy when used to train malware detectors. To do so, we programmatically analyzed the binaries in the large Ember dataset to identify their defining characteristics. We start with several representative case studies that illustrate our findings, before describing more general takeaways.

Case study I. Figure 2 shows code snippets from the Zenpak and Sivis malware families.[4] The Zenpak binary uses a code obfuscation technique called junk code insertion [48]. As its name suggests, junk code is comprised of instructions that are executed but do not affect the externalized output(s) of the program. Here, junk code manifests as a series of `inc` instructions (line 1-8) that each increment a register's value, immediately followed by a series of `dec` instructions (lines 9-16) that decrement them.

The Sivis binary also uses multiple forms of junk code insertion: (1) the `nop` instructions (lines 1-3) which do not trigger any computation or data movement, (2) the interleaved `inc` and `dec` that sequentially alter the same registers (lines 5-8, 13-14), and (3) lines 10-12 which `push` the value of `edx` onto the stack, set the value of `edx` to 0 using `xor`, and then pop the old value of `edx` from the stack and store it back into `edx` (rendering the `xor` operation useless). The Sivis

[4] x86 assembly code samples are written in Intel syntax.

Binary 1

```
push ebx
push esi
mov esi,DWORD PTR [ebp+0x8]
push edi
mov eax,ds:0x470208
push 0x7
pop ecx
lea edi,DWORD PTR [ebp-0x2c]
```

Binary 2

```
mov eax,ds:0x423e88
push ebx
push esi
mov esi,DWORD PTR [ebp+0x8]
push edi
push 0x7
pop ecx
lea edi,DWORD PTR [ebp-0x28]
```

Fig. 3: Snippets from two binaries in the same "InstallMonster" family that exhibit minor differences due to code obfuscations.

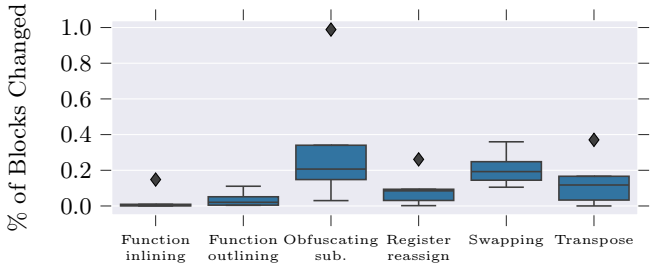


Fig. 4: Percentages of code blocks in Ember's binaries that are affected by different code transformations.

binary embeds another code obfuscation technique called opaque predicates [48], which are (typically) known a priori by a programmer to always evaluate to true or false. This manifests in relation to `eax`. At the start of the snippet, `eax` is definitively set to 0 after the `xor` instruction (line 4). However, at the point of the `cmp` instruction in line 15, the value stored in `eax` is definitively 1 due to the series of `inc` and `dec` operations in the preceding statements. In line 15, since `eax` $\neq$ `0x17b8ef93`, the jump in the following `jne` instruction is always taken.

Case study II. Figure 3 depicts snippets from two sample binaries from the Ember dataset that belong to the same family. Unsurprisingly, the two code snippets are similar at first glance. However, there exist minor differences due to two code obfuscation techniques that they embed. First, each binary uses a `mov` instruction to write data from the data segment into `eax`. However, the data is located in different memory locations across the two version; the two binaries retrieve the value from `ds:0x470208` and `ds:0x324e88`, respectively. This pattern is also seen in the `lea` instructions where the two binaries use different offsets from the stack base pointer, `ebp`, to retrieve their values. In addition, the two binaries use instruction swapping to reorder instructions (in this case, the `mov` instruction) in a manner that preserves overall semantics.

Takeaways. Our case studies highlight two main points (which we repeatedly observed across the Ember dataset):

(1) ***Semantics-preserving code transformations.*** Malware authors routinely alter prior versions of malicious programs using code obfuscation techniques that

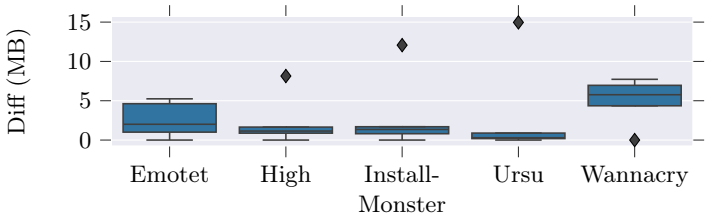


Fig. 5: Pairwise byte diffs between binaries in five representative malware families.

preserve program behavior. The reason is intuitive: generating malware involves a lot of manual labor and sophisticated code alteration. As malware detectors discern already-deployed malware by recognizing patterns in their code composition or execution regimes (§2), a far less challenging way for malware authors to continue deploying their malicious code is to perform semantics-preserving code transformations. These transformations alter that code minimally, to preserve its malicious behavior while deviating from the patterns used to detect its predecessor. Unsurprisingly, we did not observe any remnants of semantics-preserving code transformations in the benign samples that we analyzed.

(2) ***Combinations of transformations.*** To ensure sufficient differences from detected malware versions, malware authors often resort to performing semantics-preserving transformations, e.g., as in case study II above. This approach is fruitful as such transformations are often (logically) complementary, and the effect of each transformation depends on subtle interactions between the transformation logic and binary code (ranges shown in Figure 4). Additionally, we find that, to further boost diversity with multiple transformations, each obfuscation is not necessarily applied to all possible blocks in a binary, i.e., some binaries exhibited the effects of an obfuscation in all code blocks that it applied to, while others demonstrated the effects in only a fraction of those blocks.

Taking a step back, these observations lead to two implications about the large datasets that have been successfully used for ML-driven malware detection. First, there exist far fewer families of malicious binaries than malicious binaries themselves; the Ember dataset includes 300K malicious binary samples spread across only 332 families. There exist many binary versions per family: there are 287 and 13,951 binaries in the median and 99th percentile families, respectively. Second, the binaries within each family can differ quite substantially depending on the specific transformations that are applied across versions. Figure 5 highlights this property, showing that for subsets of five representative families, the constituent binaries exhibit median pairwise percent differences of 38-99% (which equates to raw differences of 0.8–5.4 MB).

Our approach. The results above motivate a new approach to bolstering the efficacy of the small datasets: data augmentation via semantics-preserving transformations. That is, we aim to grow small datasets by performing different combinations of semantics-preserving code transformations on varying numbers of blocks in the binaries. Doing so would mimic the techniques that malware authors use to sidestep malware detectors over time [7], and yield data simi-

lar to that in (proven) large datasets. We employ further code transformations done by optimizing compilers to generate new benign binaries. More importantly, semantics-preserving transformations provide a direct path to accurately labeling newly generated data without manual effort since pre- and post-transformation binaries will exhibit the same behavior (and thus can safely share labels). In §4, we describe our system, MARVOLO, that realizes this approach in a practical manner.

4 MARVOLO

4.1 Binary Rewriting Overview

Figure 6 illustrates MARVOLO's binary mutation process for performing semantics-preserving transformations on a single (malicious) binary. To begin mutation, MARVOLO decompiles existing PE32 binaries using Ddisasm [19] and internally represents the binary as a series of basic instruction (or code) *blocks*.

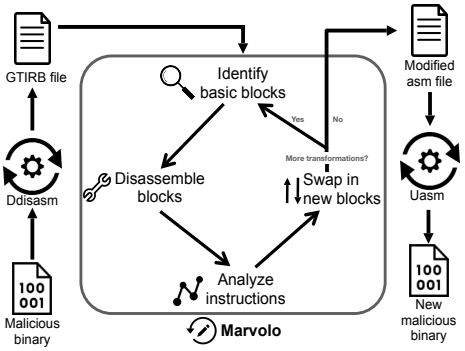


Fig. 6: MARVOLO workflow for mutating a malicious binary.

To operate on (i.e., mutate) instruction blocks, MARVOLO first disassembles each block. The resulting blocks are then passed into the MARVOLO code transformation engine, which (1) selects a set of semantics-preserving code transformations to apply to the binary during a given iteration, (2) analyzes all blocks to determine which blocks each considered transformation is applicable to, (3) selects the fraction of potential blocks to apply each transformation to, and (4) sequentially carries out the transformations on the selected blocks; §4.2 details this process. After code transformations are complete for a given iteration, MARVOLO then directly swaps out the corresponding (unmodified) blocks with their transformed counterparts and invokes an assembler to get the output binary. This binary is then added to the original dataset and tagged with the same label (i.e., malicious or benign) as the one used during its generation. This end-to-end process repeats multiple times for each binary in the dataset in accordance with a user-specified time or resource budget.

4.2 Code Transformations

MARVOLO currently supports a wide range of different semantics-preserving code transformations that cover the set of mutations we observed in our analysis of the popular Ember dataset (§3). To ensure that a modified code block is semantically equivalent to the original block, static analysis is performed after the code transformation is applied. This analysis tracks program reads and writes

and determines whether the reads from the registers and memory locations in that basic block would still return the same values after the modification. If a violation occurs from the code transformation, it is reverted and a new transformation is attempted. Appendix B provides an overview of the transformations supported.

MARVOLO's goal is to generate new versions of input binaries that differ in diverse ways from their originals while adhering to a user-specified time and/or resource budget (which dictates potential parallelism across mutation iterations). The main challenge is that it is difficult to determine, a priori, how a given transformation will alter a given binary. It depends on subtle interactions between the transformation logic and the binary instructions, which collectively dictate how many blocks are applicable for a transformation, and how many instructions will be modified, added, or deleted. Thus, during each mutation iteration, MARVOLO instead opts to randomly select multiple transformations for each mutation iteration and stochastically order them. This follows from our finding that malware authors typically employ multiple transformations together, and that binaries in the same family can differ by (largely) varying amounts (§3).

To further bolster variance across the transformed binaries, MARVOLO varies two parameters across the mutation iterations for each input binary. m specifies the number of transformation iterations to perform on each binary, and c governs the fraction of blocks to mutate in each iteration. MARVOLO maintains a running list of parameter values used for a given binary and selects subsequent values to maximize diversity, i.e., maximizing the distance from all previously used values. Note that the overarching time budget takes precedence over per-binary parameter values; to enforce this, MARVOLO round robins through the input binaries, performing one mutation iteration on each one, and circling back to fulfill the selected m per binary only if time permits. In practice, we find that 1 – 6 mutation iterations for each binary suffices in providing diversity in the amount of code that is perturbed while still being computationally feasible (keeping mutation times within several minutes).

Algorithm 1 MARVOLO data augmentation

Input: dataset S, number of new binaries k, set of supported transformations T
Output: augmented dataset S^*
$S^* \leftarrow \{\}$
for $i = 1$ **to** k **do**
  $\hat{x} \leftarrow SampleBinary(S)$
  for $j = 1$ **to** m **do**
    $t \leftarrow SelectNextTransformation(T)$
    $\hat{x} \leftarrow t(\hat{x})$
  end for
  $S^* \leftarrow S^* \cup \{\hat{x}\}$
end for
return S^*

4.3 Optimizations for Practicality

Sources of inefficiency. Binary mutation of a single executable with MARVOLO can be broken down into 3 stages: (1) invoking Ddisasm on the binary (*decompilation*), (2) carrying out semantics-preserving code transformations (*mutation*), and (3) generating the output binary (*reasssembly*). We profiled the runtime of each stage by passing 3K random binaries from Ember through MARVOLO. As shown in Figure 7, all three stages consume substantial time: median values for the three stages across binary sizes are 0.6–33, 0.1–585, and 0.1–34 seconds, respectively. We additionally observed that per-stage delays grow as binary sizes grow and span upwards of 460, 961, and 44 seconds. Accordingly, aiming to even perform a single mutation iteration on each binary in existing small datasets (which would not fully bridge the size gap with large datasets) could take up to several thousand hours! The associated resource costs would forego the savings that practitioners reap by not purchasing existing large datasets. Instead, MARVOLO embeds the following two optimizations to boost MARVOLO's utility for a given time budget; we evaluate their effectiveness §5, and provide more details in Appendix C.

(1) ***Code similarity clustering.*** A clustering strategy to group binaries based on their compositions. We make the key observation that many binaries within a malware family have equivalent code sections (i.e., the instructions are the same) and differ in other sections of the binary and leverage this insight to cluster binaries together with the same code section. Only a single binary per cluster is operated on, and the resulting code blocks are rapidly (but safely, from a semantics perspective) dropped into the other binaries in the same cluster. This approach circumvents costly operations for all-but-one binary per cluster, while preserving diverse interactions between code alterations and other sections in each binary.

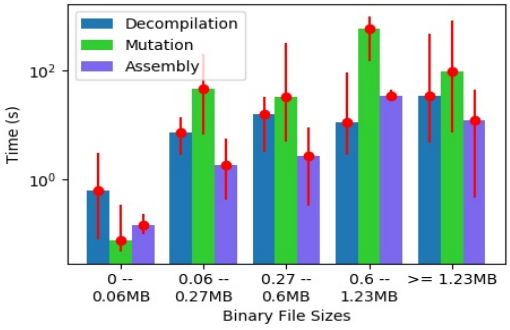


Fig. 7: Breakdown of time spent on each stage in MARVOLO's pipeline (Figure 6) for (a single run on) binaries in different size groups. Bars list medians, with error bars for 25-75th percentiles.

(2) ***Intermediate binary generation.*** A technique to increase the number of diverse binaries output from each pass through the pipeline. The main difficulty is that it is difficult to (efficiently) determine, a priori, the effects that a transformation will have on a given binary's code blocks. Thus, MARVOLO opts for a dynamic approach, whereby a lightweight runtime check determines the efficacy of outputting a binary – based on code discrepancies from the original and previously output versions – after each transformation that is performed in a pipeline pass.

5 Evaluation

5.1 Methodology

We focus on byte-based detectors that require no feature engineering and extraction to deploy. First, such methods are the fastest to run (naturally, they require no feature extractor to run) making them realistic for triage use. Second, manual human effort to understand a file can take days or weeks of work [44] and the needed features will change over time [7]. Byte-based models allow immediate adaption to new content. Thus, we use the state-of-the-art MalConv2 deep malware detector as our primary model [34]. We also include two non-deep approaches based on compression algorithms that are commonly used for malware detection, the Lempel Ziv Jaccard Distance (LZJD) [35] and Burrows Wheeler Markov Distance (BWMD) [37].

Our experiments consider two main datasets: (1) the commercial Ember dataset with 1.1M samples, and (2) the smaller-scale Brazilian malware dataset [16] with 50K samples. Given the realism of Ember observed by researchers and practitioners, we use its test set, which consists of 200K benign and malicious samples, to reflect malware detection scenarios in the wild. For training, we consider a subset of the 600K-sample Ember training dataset, as well as the Brazilian dataset; we train a separate MalConv2 model for each case. In contrast to the Ember subsets in Figure 1, we purposely constrain the number of families to realistically mimic the dataset compositions commonly used in smaller datasets [11, 42].

5.2 Overall Accuracy Improvements

Approach	Accuracy	AUC
Random Insertion	1.09	0.53
Dropout	0.27	0.69
Random Replacement	1.72	-0.10
Synonym Replacement	0.99	-3.90
MARVOLO	**4.07**	**9.06**

Table 1: Accuracy and AUC percentage improvements for opcode sequence augmentation and MARVOLO augmentation

For each dataset, we train MalConv2 to convergence. Training involves first collecting (converged) "pre-trained" weights on the original training dataset, and then running an additional training round (5 epochs) with the augmented dataset that MARVOLO generates. For all baselines, we use the default hyperparameters provided. Accuracy is reported as the percentage of correct labels (i.e., benign or malicious) output by MalConv2. We also measure AUC, which is an especially important metric for malware analysts because of the need to characterize and rank binaries to determine which ones should be analyzed, identified, and quarantined sooner rather than later [11, 32]. Thus, A high AUC is crucial since it corresponds to a successful ranking of most malicious files above benign files. A discussion on using MARVOLO following our experiments can be found in Appendix D.

We first compare how MARVOLO performs in contrast to prior malware augmentation approaches. We first compared MARVOLO using all code transformations across both benign and malicious files against prior approaches that

modify the opcode sequence representations of programs [24, 30]. We mimic the experimental setup used in [30] by using the Continuous Bag of Words (CBOW) word2vec algorithm [28] trained on opcode sequences from the binaries in our dataset to construct an embedding matrix with information that represents the semantic similarity (e.g., `add` and `adc`) between opcodes. Opcodes with similar embedding vectors tend to be semantically similar. Each opcode in the sequence is then replaced with its corresponding word2vec embedding vector and converted to a binary file to be ingested by MalConv2. We implement four core strategies featured in opcode sequence augmentation: (1) random insertion, (2) random deletion, (3) random replacement, and (4) synonym replacement. For each experiment, we generated 6K mutated binaries. Table 1 contains the results, with MARVOLO yielding 2.35 – 3.8% higher accuracy and 8.4% – 9% higher AUC over the opcode sequence augmentation strategies. Since MARVOLO performs *meaningful* data augmentation by mimicking the code alterations made in practice, we generate more realistic data samples. We also applied image-based augmentation [31] to our dataset by converting our binaries to RGB images and augmenting them with Gaussian, Poisson, and Laplace noise. Across multiple experiments, performance improvements did not exceed 1%, and thus significantly trails the wins delivered by MARVOLO and even led to occasional accuracy degradations.

We also evaluated MARVOLO using the subset of the Brazilian malware dataset which yielded accuracy gains of 1 – 2%. Unlike the Ember dataset, the Brazilian dataset does not contain family labels for its malware so we could not constrain the training set to several families of interest. Thus, the original training dataset exhibited more heterogeneity so adding additional augmented samples had a weaker effect. Table 2 shows our results for evaluating BWMD [37] and LZJD [35]. For each of these algorithms, we use both logistic regression and XGBoost [17] for malware classification. Overall, MARVOLO achieves up to a 1.18% improvement in accuracy and 1.61% improvement in AUC. Some of the effects of the mutations are lost after compression so the difference between the new embeddings and the embeddings from the unmodified binaries is less pronounced than the difference between the binaries without compression. Nonetheless, we note that even a 1% increase in accuracy is significant because of the sheer size and heterogeneity of our test dataset as well as the potential catastrophic consequences of misclassifying just a single file.

Model	Accuracy	AUC
LZJD + Logistic Regression	0.87	1.01
LZJD + XGBoost	1.14	1.14
BWMD + Logistic Regression	1.18	1.61
BWMD + XGBoost	1.01	-0.23

Table 2: Accuracy and AUC percentage improvements for non-deep baselines yielded by MARVOLO augmentation

5.3 Analyzing MARVOLO

Figure 8 shows the accuracy improvements that MARVOLO brings to MalConv2 when augmenting the Ember training dataset with different numbers of mutated samples (ranging from 3-12K). We run each experiment four times and report

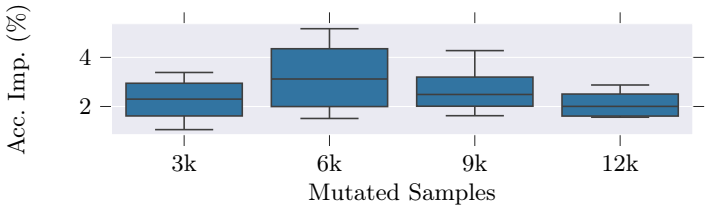


Fig. 8: Accuracy improvements (y-axis) when training MalConv2 on the Ember dataset augmented with different numbers of mutated samples (from MARVOLO).

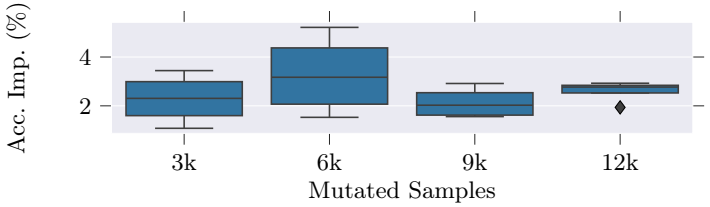


Fig. 9: MARVOLO's accuracy improvements (y-axis) when testing on only unseen (in the training data) malware families in the Ember test dataset. 8.

on the distributions. Accuracy improvements range from 1–5% atop the baseline accuracy of 61.3% and AUC improvements range from 5–10% atop the baseline AUC of 65.2% achieved when considering the unmodified Ember dataset alone. These results highlight that accuracy improvements typically come quickly, while operating on only a small number of binaries, e.g., adding only 3K and 6K mutated samples to the dataset delivers 3.5% and 5% of accuracy boosts, respectively. The reason is that MARVOLO's efficiency-centric optimizations promote rapid diversity amongst the generated samples, which in turn enable MalConv2 to quickly strike a desirable balance between (1) learning to detect obfuscation patterns, while (2) not overfitting to mutated samples. Results on the smaller Brazilian malware dataset [16] were comparable: adding 2K mutated files delivered median accuracy improvements of 2% (atop the 61% without MARVOLO).

Further analysis reveals that a key driver of the overall accuracy wins delivered by MARVOLO are improvements on test samples from *previously unseen* malware families, i.e., families that did not appear in the training dataset. Recall from §2 that such samples are the ones which static analysis and small-scale ML approaches typically struggle to generalize to. Figure 9 illustrates this, showing that MARVOLO's accuracy boosts on only the subset of test binaries that were not seen during training are on par with the wins on the complete test set (1–5%). The underlying reason for these improvements is that code transformations provide a discernible pattern for MalConv2 to link across diverse binaries in different families.

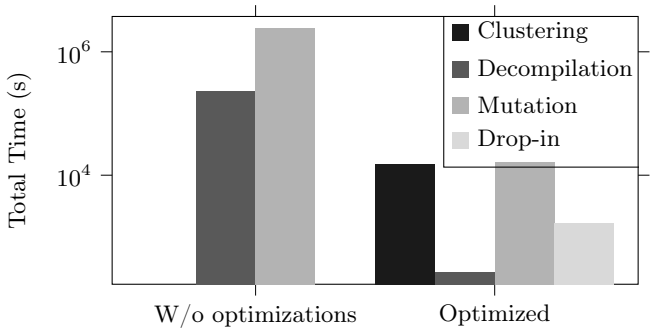


Fig. 10: Time spent on various stages of the mutation pipeline for two versions of Marvolo: one with both optimizations, and one without. Mutation and reassembly are combined into a single bar for ease of disposition. Results are aggregate times when generating 3K mutated samples.

Importance of number of binaries mutated. Figures 8 and 9 show Marvolo's performance as the number of added mutated binaries changes. As discussed, the benefits from Marvolo's mutations come early as most accuracy wins can be realized by using only a small fraction of the overall dataset as input. More generally, however, Marvolo's performance with regards to input size is collectively governed by two factors – (1) the overall dataset size, and (2) the number of input samples – that influence the relationship between the utility of malware detection insights from newly added (mutated) samples and the risk of overfitting. Intuitively, larger datasets require larger numbers of mutated samples to reap benefits because they already exhibit a sufficient amount of heterogeneity (as shown in Figure 1), and they are also far less susceptible to overfitting (as the weight of each added sample is relatively smaller).

Code transformation	Type	Accuracy
Junk code	Malware	2.62
Swapping	Malware	1.41
Obfuscating sub.	Malware	3.18
Register reassignment	Malware	1.80
Code transposition	Malware	2.47
Opaque predicates	Malware	1.00
Optimizing sub.	Benign	2.83
Function outlining	Benign	1.23
Function inlining	Benign	2.60
Function reordering	Benign	0.04

Table 3: Marvolo's accuracy improvements when using a version of Marvolo that only performs a single type of semantics-preserving code transformation during mutation.

Importance of different transformations. Table 3 shows the effect that each transformation has on accuracy improvement. In summary, we find that we generally reap more accuracy improvements when mutating malicious files over benign files. Intuitively, many datasets only consist of malicious files from several families that do not employ a diverse set of obfuscations. Delving further, we find that

obfuscating instruction substitution (replacing an instruction with an abstruse sequence of different instructions) yields the highest accuracy wins followed by junk code insertion and code transposition (reordering code blocks). These are commonly obfuscations that significantly change the file's appearance. Further, we attain significant improvements for mutations on benign files with optimizing instruction substitution and function inlining being the most prominent. These transformations improve the model by mimicking common types of optimizations that compilers employ in practice (which are not as widely used by malware authors, who opt to use their own toolchains and have fewer incentives to deploy optimized code). Function reordering, which changes the positions of functions in the file, shows that simply making arbitrary modifications that do not represent the transformations made in practice provide little benefit.

Importance of* MARVOLO*'s optimizations. Recall from §4 that MARVOLO embeds two optimizations to tackle the overheads in the mutation process. To uncover the effects of these optimizations, we profiled two runs of MARVOLO's mutation pipeline, one with the two optimizations enabled, and one without them. Each pipeline was used to generate 3K mutated samples, and we note that the MalConv2 models trained on these mutated samples (atop the Ember dataset) delivered accuracy within 1% of one another.

Figure 10 shows the aggregate time spent in each pipeline stage across these two variants. The optimized version runs 79× faster to generate mutated samples of similar efficacy (given the near-identical MalConv2 performance across the two cases noted above). Speedups are primarily from the lower decompilation and mutation/reassembly costs, which in turn are due to running only a single binary per cluster through the pipeline (85% fewer binaries), with each run yielding a larger number of mutated samples. These drops dwarf the drop-in overheads used to mix (altered) code and data blocks, and the slight (blocking) overhead of performing clustering prior to mutation; note that clustering overheads are paid once, and will thus steadily decrease in relative importance as the number of mutated samples grows.

6 Conclusion

MARVOLO is a data augmentation engine that boosts the efficacy of the malware datasets that practitioners commonly are restricted to by performing semantics-preserving code transformations on the constituent binaries. To the best of our knowledge, we are the first to leverage insights from a deep-dive analysis of existing malware datasets to apply meaningful data augmentation to the domain of malware detection. Key to MARVOLO's practicality are its ability to (safely) propagate labels across input and output binary samples, and its optimizations to boost the number of fruitful (i.e., diverse and representative) data samples generated within a fixed time budget. Experiments using commercial malware datasets and a recent ML-driven malware detector show that MARVOLO boosts accuracies by up to 5%, while operating on only 15% of the available binaries (mutation speedups of 79×).

Bibliography

[1] Malware detection using statistical analysis of byte-level file content. pp. 23–31. ACM Press (2009). https://doi.org/10.1145/1599272.1599278, http://portal.acm.org/citation.cfm?doid=1599272.1599278

[2] Code obfuscation. https://en.wikibooks.org/wiki/X86_Disassembly/Code_Obfuscation (2021), accessed: 2023-06-12

[3] Ghidra software reverse engineering framework. https://ghidra-sre.org/ (2021), accessed: 2023-06-12

[4] Global Ransomware Damage Costs Predicted To Reach $20 Billion (USD) By 2021. https://bit.ly/3j3bTEB (2021), accessed: 2021-10-06

[5] GTIRB Pretty Printer. https://github.com/GrammaTech/gtirb-pprinter (2021), accessed: 2023-06-12

[6] Gtirb rewriting api. https://github.com/GrammaTech/gtirb-rewriting (2021), accessed: 2021-08-07

[7] Labs Report at RSA: Evasive Malware's Gone Mainstream. https://bit.ly/3p2lH5G (2021), accessed: 2021-10-07

[8] VxSig. https://github.com/google/vxsig (2021), accessed: 2023-06-12

[9] Yara: The pattern matching swiss knife for malware researchers (and everyone else). http://virustotal.github.io/yara/ (2021), accessed: 2021-08-07

[10] Portable Executable. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format (2023), accessed: 2023-06-12

[11] Abedelaziz Mohaisen, O.A.: Unveiling Zeus: Automated Classification of Malware Samples. In: WWW Companion (2013)

[12] Anderson, H.S., Roth, P.: EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. In: arXiv 1804.04637 (2018)

[13] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K.: Dos and Don'ts of Machine Learning in Computer Security. In: USENIX Security Symposium (2022)

[14] Bhagat, N., Arora, B.: Intrusion detection using honeypots. In: PDGC (2018)

[15] Bozorgi, M., Saul, L.K., Savage, S., Voelker, G.M.: Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In: ACM SIGKDD. pp. 105–114 (2010). https://doi.org/10.1145/1835804.1835821

[16] Ceschin, F., Pinagé, F., Castilho, M., Menotti, D., Oliveira, L.S., Grégio, A.: The Need for Speed: An Analysis of Brazilian Malware Classifiers. In: IEEE Security & Privacy (2018)

[17] Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 785–794. KDD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2939672.2939785, https://doi.org/10.1145/2939672.2939785

[18] Fan, Y., Hou, S., Zhang, Y., Ye, Y., Abdulhayoglu, M.: Gotcha - Sly Malware! Scorpion: A Metagraph2vec Based Malware Detection System. In:

ACM SIGKDD. pp. 253–262 (2018). https://doi.org/10.1145/3219819.3219862

[19] Flores-Montoya, A., Schulte, E.: Datalog Disassembly. In: 29th USENIX Security Symposium (2020)

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

[21] Huang, W., and Stokes, J. W.: Mt-Net: A Multi-Task Neural Network for Dynamic Malware Classification. In: DIMVA. pp. 6528–6537 (2016)

[22] Ibrahim, A.H., Abdelhalim, M.B., Hussein, H., Fahmy, A.: Analysis of x86 instruction set usage for windows 7 applications. In: International Conference on Computer Technologies and Development (2011)

[23] Jamalpur, S., Sai Navya, Y., Raja, P., Tagore, G., Rama Koteswara Rao, G.: Dynamic malware analysis using cuckoo sandbox. In: IEEE ICICCT) (2018)

[24] Jason Wei, K.Z.: Eda: Easy data augmentation techniques for boosting performance on text classification tasks. In: IJCNLP (2020)

[25] Joyce, R.J., Amlani, D., Nicholas, C., Raff, E.: MOTIF: A Large Malware Reference Dataset with Ground Truth Family Labels. In: The AAAI-22 Workshop on Artificial Intelligence for Cyber Security (AICS) (2022). https://doi.org/10.48550/arXiv.2111.15031, https://github.com/boozallen/MOTIF

[26] Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM — Software Protection for the Masses. In: SPRO (2015)

[27] Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: ACM SIGKDD (2004). https://doi.org/10.1145/1014052.1014105

[28] Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient Estimation of Word Representations in Vector Space. In: arXiv:1301.3781v3

[29] Nguyen, A.T., Raff, E., Sant-Miller, A.: Would a file by any other name seem as malicious? In: IEEE Big Data (2019)

[30] Niall McLaughlin, J.M.d.R.: Data augmentation for opcode sequence based malware detection. In: arXiv 2106.11821 (2021)

[31] Ozgur, F., Catak, Ahmed, J., Sahinbas, K., Hussain Khand, Z.: Data augmentation based malware detection using convolutional neural networks. In: PeerJ Computer Science (2021)

[32] Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.: Malware Detection by Eating a Whole EXE. arXiv preprint arXiv:1710.09435 (oct 2017), http://arxiv.org/abs/1710.09435

[33] Raff, E., Filar, B., Holt, J.: Getting Passive Aggressive About False Positives: Patching Deployed Malware Detectors. In: 2020 International Conference on Data Mining Workshops (ICDMW). pp. 506–515. IEEE (nov 2020). https://doi.org/10.1109/ICDMW51313.2020.00074, https://ieeexplore.ieee.org/document/9346444/

[34] Raff, E., Fleshman, W., Zak, R., Anderson, H.S., Filar, B., McLean, M.: Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection. In: AAAI (2021)

[35] Raff, E., Nicholas, C.: An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance. In: ACM SIGKDD. pp. 1007–1015 (2017)
[36] Raff, E., Nicholas, C.: A survey of machine learning methods and challenges for windows malware classification. In: ML-RSA (2020)
[37] Raff, E., Nicholas, C., McLean, M.: A New Burrows Wheeler Transform Markov Distance. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence (2020), http://arxiv.org/abs/1912.13046
[38] Raff, E., Zak, R., Munoz, G.L., Fleming, W., Anderson, H.S., Filar, B., Nicholas, C., Holt, J.: Automatic yara rule generation using biclustering. In: ACM CCS AISec (2020)
[39] Ri Or-meir, Nir Nissim, Yuval Elovici, Lior Rokach: Dynamic malware analysis in the modern era—a state of the art survey. In: ACM Computing Survey 52, 5, Article 88) (2018)
[40] Ronen, R., Radu, M., Feurstein, C., Yom-Tov, E., Ahmadi, M.: Microsoft Malware Classification Challenge. In: arXiv 1802.10135 (2018)
[41] Schulte, E., Dorn, J., Flores-Montoya, A., Ballman, A., Johnson, T.: GTIRB: Intermediate Representation for Binaries. In: arXiv 1907.02859 (2019)
[42] Smith, M.R., Johnson, N.T., Ingram, J.B., Carbajal, A.J., Haus, B.I., Domschot, E., Ramyaa, R., Lamb, C.C., Verzi, S.J., Kegelmeyer, W.P.: Mind the Gap: On Bridging the Semantic Gap between Machine Learning and Malware Analysis. In: ACM CCS AISec (2020)
[43] Tamersoy, A., Roundy, K., Chau, D.H.: Guilt by Association: Large Scale Malware Detection by Mining File-relation Graphs. In: ACM SIGKDD. pp. 1524–1533 (2014). https://doi.org/10.1145/2623330.2623342
[44] Votipka, D., Rabin, S.M., Micinski, K., Foster, J.S., Mazurek, M.M.: An Observational Investigation of Reverse Engineers ' Processes. In: USENIX Security Symposium (2019)
[45] Wartell, R., Zhou, Y., Hanlen, K.W., Kantarcioglu, M., Thuraisingham, B.: Differentiating code from data in x86 binaries. In: ECML PKDD (2011)
[46] Ye, Y., Li, T., Chen, Y., Jiang, Q.: Automatic Malware Categorization Using Cluster Ensemble. In: ACM SIGKDD. pp. 95–104 (2010). https://doi.org/10.1145/1835804.1835820
[47] Ye, Y., Li, T., Zhu, S., Zhuang, W., Tas, E., Gupta, U., Abdulhayoglu, M.: Combining File Content and File Relations for Cloud Based Malware Detection. In: ACM SIGKDD. pp. 222–230 (2011). https://doi.org/10.1145/2020408.2020448
[48] You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: BWCCA (2010)

7 Ethical statement

While MARVOLO can be employed by malicious actors to obfuscate their code, they already have tools of greater functionality that run faster because malware authors have their source code. MARVOLO thus gives no new abilities to malware authors, but does enable researchers to better prepare and study such obfuscations impact on modeling and detectors — as researchers rarely have source code, let alone at an industry scale.

A Further Background and Related Work

Primer on PE format. Most approaches in malware detection analyze raw executables (or binaries) [3, 9, 23, 32]. Due to the widespread usage of Windows systems and the large amounts of malware targeting these systems, these malware detectors focus primarily on analyzing PE32 executables. As a brief primer, a PE file consists of a data structure that provides the OS with the necessary information to load the program into memory. It contains a series of headers and sections with different data. Most notably, the `.text` section contains the instructions to be executed, the `.data` section contains global variables, and the `.rsrc` section contains resources used by the program such as icons. We refer the interested reader to the PE specification for further details [10].

Programmatic malware detection. Early work in malware detection involved both static and dynamic analysis techniques [36]. Static analysis approaches primarily involved using a tool such as Yara [9] to generate specific rules or patterns for identifying malicious code; pattern searches may target only code sections [8], or general byte signatures across all sections of a binary [38]. However, static signatures fail to keep pace with the rapidly evolving space of deployed malware variants [7]. Further, generating such rules is time-consuming and requires substantial manual effort from malware analysts to uncover patterns that appear in malware. Automatic signature generation tools do exist [8, 38], but have witnessed limited adoption as the resulting signatures are often restricted to exact matches on byte sequences and strings observed in previously-seen malware samples.

Malware detectors rooted in dynamic analysis [39] typically execute a binary in a sandbox to observe its behavior while isolating (or at least, restricting) any potential damage. Although dynamic approaches step past the limitations of static analyses that are restricted to pre-determined signature searches, the required analysis can be computationally expensive because each file often must be executed multiple times to elicit harmful behavior. Worse, some malicious binaries embed checks to detect whether they are running a virtual (sandbox) environment based on VM properties such as the amount of available DRAM, the number of cores, the list of installed applications/tools, and even the temperature of the CPU [36]. If a sandbox is detected, these malicious programs dynamically alter their behavior to hide malintent (thereby evading detection).

Data-driven malware detection. To address the above limitations and deliver detection accuracy (and generalization), data-driven techniques using deep learning models have seen significant traction in recent years. These models typically consist of neural networks that determine whether or not a given binary is malicious or benign based on various, defining features of that binary. For instance, certain models run inference over PE header values, assembly code, network traffic, and even the names of binaries [29, 36]. Others follow a dynamic approach and perform manual feature engineering of API calls [21]. Most recently, the MalConv CNN [32] performs malware detection by operating directly over the raw bytes in a binary, thereby eschewing labor-intensive feature engineering and the need for domain expertise.

The Problem: Limited (Realistic) Data. As expected, the effectiveness of data-driven malware detectors heavily depends on the data used to train the corresponding neural networks. Unfortunately, to date, it is practically difficult for practitioners to obtain access to training datasets that are sufficiently representative of malware in the wild.

Commercial datasets that contain massive amounts of labeled data samples for malware detection do exist and have been used to train models that deliver excellent malware detection accuracy in the wild [12]. For instance, the popular Ember dataset contains 1.1 million samples and close to 3,000 distinct malware families. However, obtaining the raw executables in the Ember dataset mandates having a VirusTotal license, which can cost upwards of $400,000 per year![5] Consequently, many cost-constrained practitioners and research groups must resort to far smaller datasets that are publicly available, e.g., the Brazilian malware dataset contains 50K files [16], while the Microsoft malware dataset contains 20K files with only 9 malware families [40].

On the other hand, practitioners can opt to generate homegrown datasets using honeypots that attract malware binaries [14]. However, such approaches face three challenges. First, the type of malware that is gathered is dependent on the collection methodology set by the user, leading to biased datasets [36]. Second, collecting a sufficient number of benign data samples is difficult as benignware does not seek to replicate across machines (like malware does), and software is often closed-source and copyright-protected. Along these lines, many seminal works in malware detection have struggled to obtain benign executables, often collecting them from clean installations [1, 27], but this fails to obtain more than a few thousand samples. More recent works often rely on partnerships with anti-virus companies in order to obtain sufficient benign samples [15, 18, 35, 43, 46, 47]. This naturally results in unsharable data, causing reproducibility challenges [36], slows research by non-connected groups, and neglects the needs of niche and targeted malware[16, 38]. Finally, even if practitioners were to obtain a large number of samples, labeling them is not straightforward. Software reverse-engineering tools exist [3], but can consume many hours to reverse engineer a single executable, even for expert analysts [11, 36, 44].

Takeaway 1: small malware datasets lack heterogeneity, fail to generalize. Across the considered free, small datasets that are sized between 20-75k samples, MalConv's accuracy spanned only 60-71% relative to a training on the full Ember training dataset (600k).

Takeaway 2: large (proven) malware datasets have important diversity that detectors capitalize on. Figure 1 shows the diminishing accuracy and AUC of MalConv when trained on progressively fewer data samples from the Ember dataset. Starting with the full 600K Ember training dataset, accuracy is at 91%. However, accuracy

[5] Ember's free offering omits executables, and only presents a limited number of features per binary, e.g., size, library functions. These features are insufficient for most existing data-driven malware detectors, and cannot support long term development: analysts must avoid having adversaries learn about the used features, and cannot test new features without access to the binaries.

dips below 80% when trained on subsets sized similarly to existing free datasets, e.g., 75k samples and less. These results indicate the data-hungry nature of ML-based malware detectors, and highlight the heterogeneity in data samples in large datasets; we dig deeper into these aspects in the following section.

B MARVOLO's Code Transformations

Junk code insertion. Insert instructions into the binary that don't alter the output of the program upon being executed. These instructions may change the state of the program (e.g., register values and memory) but reverse the changes before progressing to subsequent instructions. The simplest form of this transformation that we implement is the insertion of `nop` instructions. We also generate semantic nops which consist of pushing values onto the stack, performing arithmetic and logical operations, and then popping the values off once they're completed. We augment this with additional instructions that also read and write to memory. In the following semantic nop

```
push eax
inc eax
or  eax, 0x1c
add eax, dword ptr [esp - 0x34]
not eax
pop eax
```

the `eax` register is first pushed to the stack. Then arithmetic and bitwise operations are performed on `eax`. Lastly, the old value of `eax` is popped from the stack and written back into `eax`; since the value of `eax` is not written elsewhere prior to `pop eax`, the computations are effectively useless.

Register reassignment. Changes the names of the variables or registers. Identify a live register, `rX`, within a basic block and replace it with a new register, `rY`, that is unused within the block. The value of `rY` is first pushed onto the stack and is then written with the value stored in `rX`. After computations are performed on `rY`, it is written to `rX` and the original value of `rY` is popped and written back to `rY`.

Function inlining. Identify functions and every time they are invoked, replace the `call` instructions with the bodies of the identified functions. In our implementation, we solely focus on functions with straight-line code. Function inlining is a common compiler optimization used to reduce the overhead of invoking a function and to make basic blocks more amenable to subsequent optimizations.

Function outlining. Identify straight-line instructions within the current basic block and generate a new function with those instructions. Replace the original instructions with a `call` instruction to the newly-generated function. This is a compiler optimization for reducing code size.

Obfuscating Instruction substitution. Replace an instruction with a semantically equivalent sequence of new instructions. We currently support over 30 substitutions. We add simple substitutions such as changing `add rX, 1` to `sub rX, -1`. We adopt further instruction substitutions, including many implemented in

LLVM Obfuscator [26]. These substitutions are mostly comprised of more complex bitwise and arithmetic instructions. For instance, MARVOLO would replace the instruction `or eax,0x4711` with

```
push esi
push edi
mov esi, eax
mov edi, 0x4711
and eax, edi
xor esi, edi
or  eax, esi
pop edi
pop esi
```

The transformation is effectively replacing $a = b|c$ with $a = (b \ \& \ c)|(b \oplus c)$.

Optimizing instruction substitution. Replace an instruction with an equivalent instruction that optimizing compilers often emit [2]. While these instructions are often times not as intuitive as their more straightforward counterparts, they are faster to execute. For instance, `mov rX, 0` is often times changed to `xor rX, rX`. Another instance is substituting arithmetic instructions, such as `add`, with `lea` instructions. Applying this transformation more broadly captures the range of programs that can be produced by different compiler toolchains and options.

Code transposition. This transformation reorders a sequence of instructions that changes the appearance of the code without altering the behavior [48]. MARVOLO implements code transformation by dividing a basic block into smaller slices. Then these slices are rearranged in a different order and are each appended with an unconditional `jmp` instruction to ensure that the original execution order of the initial basic block is preserved.

Instruction swapping. As another form of code transposition, we take 2 instructions and swap their positions. While this transformation does not significantly affect the readability of the code, it is used by malware authors to evade anti-virus scanners. To ensure that the transformation preserves semantics, analysis is performed to check that the swap doesn't violate any computational dependencies. We check that each of the destination registers for the instructions aren't used as a source register for other instructions. We also check that any source registers used by the two instructions aren't written to. Below we demonstrate an example; the left side shows the original program and the right side shows the modified program after the `add` and `sub` instructions had been swapped.

Original	**Mutated**
`add eax, ebx`	`sub ecx, 0x7c21`
`sub ecx, 0x7c21`	`add eax, ebx`
`ret`	`ret`

On the other hand, the program

```
mov eax, 0x1af3
add ecx, eax
```

is not amenable to swapping since the `add` instruction would not use the updated value in `eax` after the `mov` instruction.

Opaque predicate insertion. Opaque predicates are predicates that always evaluate to true or false and are known a priory the programmer. While opaque predicates evaluate to the same value under all inputs, they are still evaluated during runtime. To represent the instances where code and data are interleaved within a binary [45], we generate a sequence of randomly-generated bytes following the opaque predicate. An unconditional `jmp` instruction is inserted so that these generated bytes are not executed and the next instructions within the program are run. Opaque predicates are commonly inserted by code obfuscators. [26].

Function reordering. Functions are moved to different positions throughout the binary. This transformation drastically changes the appearance of the binary without adding new instructions or removing existing ones.

C MARVOLO Optimizations

Code similarity clustering. To reduce the number of binaries passed through the mutation pipeline, MARVOLO employs a clustering strategy to group binaries together based on their compositions (and thus, their interactions with the pipeline). Efficiency wins come from passing only a single binary per cluster through the pipeline. Intuitively, the goal for clustering is thus to maximize cluster sizes without masking differences between the binaries in the dataset.

Unfortunately, the straightforward clustering strategy of grouping binaries based on byte similarity (i.e., cluster binaries whose byte-level differences are smaller than a pre-determined threshold) are ill-suited for our task. The reason is that, even malicious binaries within the same family that exhibit identical `.text` and `.data` sections, which contain program instructions and global variables respectively, may have vast byte-level differences (upwards of tens of thousands of bytes). Though massive, these differences do not alter the overall behavior of the binary, and thus should not map binaries to different clusters. Yet unearthing such insights requires passing the binary through costly decompilation, foregoing many savings.

Instead, MARVOLO leverages our finding that, within a malware family, it is not uncommon for multiple binaries to have equivalent `.text` sections (i.e., the binaries' instructions are the same); note that these binaries commonly differ in their `.data` and `.rsrc` sections – we discuss this below. Since MARVOLO only performs code transformations, these are the only portions of the binary that MARVOLO modifies; it is thus redundant to send binaries with identical `.text` sections through MARVOLO's pipeline. Consequently, MARVOLO operates on only a single binary per observed code section. For each generated mutated version of the binary, MARVOLO performs drop-in replacement (i.e., avoiding

costly decompilation and reassembly) of the transformed code section with other binaries in the same cluster; memory location offsets are quickly updated in each affected binary. In effect, this rapidly simulates the process of passing all binaries in a cluster through the end-to-end pipeline.

Note that modifying `.data` and `.rsrc` sections in a binary may not deliver semantic equivalence. In contrast to semantics-preserving code transformations that guarantee equivalent behavior across program inputs, data-level modifications can alter the taken control flows in a program, resulting in different externalized values. In light of this, MARVOLO only performs drop-in replacement for binaries in the same cluster, i.e., that have identical code sections to the one which passed through the mutation pipeline. This ensures that code-data relationships are unchanged since the same control flows would be traversed during binary execution, which in turn ensures safety in propagating labels to newly generated binaries.

Intermediate binary generation. The goal of MARVOLO's second optimization is to maximize the useful binaries output during each pass through the mutation pipeline. Recall that, for each input binary, MARVOLO's pipeline (as described thus far) selects and performs a series of transformations to generate a single mutated binary. Thus, a simple approach to increase pipeline outputs for a given run would be to output a mutated binary after each successive transformation is performed. The issue is that the generated binaries will only differ by a single transformation pass and thus will likely fail to deliver the heterogeneity seen in large datasets; recall from §3 that malware authors commonly use multiple transformations to ensure substantial differences from the original malware binaries. Instead, we must ensure that the generated binaries diverge substantially from one another.

The challenge is that it is difficult to know a priori how many bytes a transformation will change in a given binary (§4.2). To handle this, MARVOLO employs a lightweight runtime check after each transformation is applied to determine whether the code changes performed up until that point are comprehensive enough to warrant a new binary generation (and thus assembly). Logically, the runtime checks compare byte-level diffs between the current binary version, the original, and those output after prior transformations; if all values exceed a preset threshold, MARVOLO deems the current binary worthy of costly assembly (and thus, a new sample in the dataset).[6] To ensure that discrepancies only pertain to behavior-affecting portions of the binary without requiring costly assembly and binary-wise diffs (which we find can consume tens of seconds), MARVOLO approximates this behavior by tracking the number of code blocks affected after each step (scaled based on the inherent intrusion level of the applied transformation [22]).

[6] While most binaries within a family have significant differences, some exhibit only minor differences between one another. Thus, MARVOLO occasionally (10% of the time, by default) outputs binary versions even if the diff threshold has not been exceeded.

D Using MARVOLO

Indeed MARVOLO is intended to complement existing ML-driven malware detectors and we do not propose changing hyperparameters but we recommend keeping the hyperparameter-tuning methodology the same after data augmentation. Beyond these hyperparameters, we note two additional considerations:

1. ***Input seclection.*** MARVOLO performs best when presented with inputs comprising a diverse set of binaries that differ (as the dataset allows) in family and composition, e.g., binaries with large fractions of differing code portions. Doing so aids malware detectors in identifying the underlying transformations (injected by MARVOLO) across wider-ranging contexts. Further, as noted above, MARVOLO must balance generating sufficient mutated samples to boost heterogeneity in training datasets, while avoiding overfitting to those samples. Our current implementation leverages that accuracy boosts come early (i.e., with few samples) and overfitting occurs soon after, motivating an iterative process starting with only a small number of samples.
2. ***Transformation selection.*** Our results in Section 3 highlight that malware authors not only use many different kinds of code transformations, but also diverse combinations of them. Thus, MARVOLO opts for a general randomized selection of transformations and combinations during mutation. However, to make the most use of (limited) compute resources, a practitioner could identify which code transformations are present in the samples that they already have, and focus the augmentation process on under-represented ones.

Extending MARVOLO ***with new transformations.*** MARVOLO currently supports the 10 binary transformations mentioned in B. Due to the modular design of MARVOLO and the availability of binary rewriting tools, a malware analyst can extend MARVOLO with newer transformations as they are encountered. MARVOLO uses the GTIRB rewriting framework [6], which not allows for x86 code modifications, but also supports numerous binary formats. Hence, MARVOLO can also augment datasets consisting of different binary types (e.g., PE64, ELF).

Using binaries that cannot be executed. Fundamentally, our binary rewriting approach should result in properly labeled, working binaries because we only employ semantics-preserving transformations. We observed that the Ddisasm rewriting framework [5, 6, 19, 41] has several bugs that resulted in binaries that did not execute, however we note that because many ML-based malware detectors use static approaches to learn the binaries, these binaries never need to be executed. Further, because the errors in the binaries are very insignificant, the models are invariant to them and as shown in Section 5, they still yield significant performance improvements.