Bolstering Binary Datasets for Malware Detection Through Programmatic Data Augmentation

Michael D. Wong[1], Edward Raff[2], James Holt[3], Ravi Netravali[1]
[1]*Princeton University* [2]*Booz Allen Hamilton* [3]*Laboratory for Physical Sciences*

ABSTRACT

Malware continues to plague large organizations and is becoming increasingly difficult to detect as malware authors are constantly looking for new ways to obfuscate and diversify their malicious code. While neural networks for malware detection have shown significant improvements over traditional signature-based detectors, security researchers and practitioners often struggle to obtain sufficiently large and comprehensive datasets for training these models.This is especially challenging for institutions like banks and governments that receive targeted malware, and can thus can not collect large scale malware. Diverse benign applications are also uniquely challenging due to copyright and licensing restrictions. We present Marvolo, a binary mutator that programmatically grows malware (and benign) datasets in a manner that boosts the accuracy of ML-driven malware detection. To do this, Marvolo employs semantics-preserving code transformations that mimic the alterations that malware authors and defensive benign developers routinely make in practice to sidestep advances in detectors or protect considered "trade secret" code. Crucially, semantics-preserving transformations also enable Marvolo to safely propagate labels from original to newly-generated data samples without mandating expensive reverse engineering of binaries. Experiments using recent ML-driven malware detector show that Marvolo boosts accuracies by up to 5% and AUC up to 10%.

ACM Reference Format:
Michael D. Wong[1], Edward Raff[2], James Holt[3], Ravi Netravali[1]. 2022. Bolstering Binary Datasets for Malware Detection Through Programmatic Data Augmentation. In *AI4Cyber/MLHat '22*. ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cybersecurity is more important than ever as malware authors continue to devise not only new sophisticated attack methods, but also obfuscations to hide their malicious code. Indeed, malware now causes billions of dollars in damages every year [6]. Detecting malware is notoriously difficult – e.g., in 2020, it took 280 days on average to identify and contain a data breach [2] – and many of the affected systems are crucial to the operation of hospitals, schools, governments, universities, and other critical infrastructure [3].

Many approaches have been developed to aid practitioners in distinguishing malicious files (i.e., malware) from benign ones, which we will review further in §2. Early solutions centered on employing static or dynamic analyses of binaries to identify indicators of malicious behavior, e.g., stealing user credentials [33]. However, each faces significant drawbacks: static analysis relies on manually-specified signatures that struggle to generalize to newer malware variants, while dynamic analyses bring high computational costs and virtual environments that are detectable by malicious programs (which can then fly under the radar) [27]. More recently, a slew of data-driven strategies have been developed to sidestep the above issues by training ML models to distinguish between benign and malicious executables [1, 12, 21, 22, 25, 27].

Despite their promise, ML-based solutions face a significant practical challenge: obtaining representative and labeled training data is infeasible for many organizations. On the one hand, commercial datasets with these properties exist, but are unattainable for many due to financial constraints [12, 18], with the licensing needed can cost $400k/year. On the other hand, home-grown datasets face scaling and labeling challenges, e.g., benign samples are often closed-source or copyright-protected, and labeling involves error-prone manual analysis to reverse engineer each binary. Consequently, many security practitioners and researchers only have access to small datasets that lack the heterogeneity seen in the wild. For example, we find that recent ML detectors achieve accuracies of only 60-71% when trained on small public datasets [15, 31] versus their large, commercial counterparts; such degradations are unacceptable given that single-digit accuracy improvements (and any new detected malware) are celebrated by malware analysts [7]. We note that these small datasets are also realistic in representing the challenge applying ML to targeted or otherwise unique malware families of interst (e.g., targeted banking malware) rather than broad and indiscriminate malware.

We present Marvolo, a binary mutator that programmatically grows (accessible) malware datasets in a manner that directly boosts the accuracy of ML-driven malware detectors. The driving insight behind Marvolo's data augmentation strategy is drawn from our analysis of binaries in high-accuracy (but difficult to access) malware datasets (§3). In particular, we observe that these datasets routinely contain multiple versions of a given malware file that differ based on the effects of semantics-preserving code transformations, i.e., alterations to the code that change aesthetics, but not externalized behavior [39]. The reason is intuitive: producing malware requires significant effort, and once a malware binary becomes detectable, code transformations are a quick way for malware authors to preserve malicious behavior while sidestepping discernible patterns.

Building on the above observation, Marvolo performs a wide range of semantics-preserving code transformations on existing binaries in an input dataset. Crucially, this approach naturally results in automatic (accurate) labeling of the augmented data samples.

AI4Cyber/MLHat '22, August 15, 2022, Washington, DC

https://doi.org/XXXXXXX.XXXXXXX

The reason is that semantics-preserving transformations inherently preserve overall code behavior. Thus, labels of pre-transformed binaries can be safely carried over to the transformed versions.

We evaluated Marvolo using the recent MalConv2 [29] malware detector and multiple commercially-available large/small-scale datasets, i.e., the large-scale Ember [12] dataset, as well as the small-scale Brazilian and Microsoft datasets [15, 31]. Overall, we find that Marvolo boosts MalConv2's accuracies by up to 5%, with most wins coming from accurately detecting previously unseen binary families – such scenarios are intuitively more difficult to catch, but are the primary goal of any malware detection system, highlighting the practical utility of Marvolo.

2 BACKGROUND AND RELATED WORK

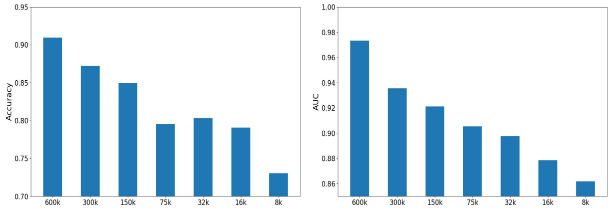
Though prior attempts have been made in data augmentation for malware detection, they do not do so in a meaningful way. In [19, 24], programs are represented as sequences of opcodes and augmentation involves replacing one opcode with another without preserving semantics. Further, [17] augments images generated from malware, which are known to be flawed representation [25]. In contrast, 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 by malware authors, and (2) a system that leverages those insights to efficiently grow small datasets into larger ones with improved heterogeneity and realism that aid end-to-end ML-based malware detection.

Malware detection involves both static and dynamic analysis techniques [27]. Static analysis approaches primarily involved using a tool such as Yara [9] to generate specific rules or patterns for identifying malicious files. However, static signatures fail to keep pace with the rapidly evolving space of deployed malware variants [7] and can take days of manual effort [28, 34]. Malware detectors rooted in dynamic analysis [30] execute a binary in a sandbox to observe its behavior while restricting potential damage. However, this 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 dynamically alter their behavior to evade detection [27].

2.1 The Problem: Limited (Realistic) Data

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. However, these malware detectors heavily rely on the data used to train the corresponding neural networks. Unfortunately, to date, it is 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. For instance, the popular Ember dataset [12] 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![1] 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 [15], while the Microsoft malware dataset contains 20K files with only 9 malware families [31].



(a) Accuracy results.

(b) AUC results.

Figure 1: Performance of MalConv2 [29] when training on different subsets of the Ember dataset [12]

On the other hand, practitioners can generate homegrown datasets using honeypots that attract malware binaries [13]. 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 [27]. Second, collecting benign data samples is difficult since software is often closed-source and copyright-protected, resulting in datasets with a few thousand benign samples [1, 21]. More recent works often rely on partnerships with anti-virus companies in order to obtain sufficient benign samples [14, 16, 26, 32, 37, 38]. This naturally results in unsharable data, causing reproducibility challenges [27], slows research by non-connected groups, and neglects the needs of niche and targeted malware[15, 28]. Finally, even if practitioners were to obtain a large number of samples, labeling them is not straightforward. Software reverse-engineering tools exist [5], but can consume many hours to reverse engineer a single executable, even for experts [10, 27, 35].

To demonstrate the sensitivity of ML-based malware detectors to dataset composition and size, we ran experiments comparing the efficacy of models trained with commercial large-scale (Ember) and small-scale datasets. Results use the recent MalConv2 detector [29], and follow the setup described in §5 (testing is done on the 200K Ember test set). To contextualize these results, we note that the implications of detecting even a single additional malicious binary in the wild can be substantial (§1), and that single-digit accuracy improvements are celebrated by malware analysts [7].

Takeaway 1: small malware datasets lack heterogeneity, fail to generalize. Across the considered free, small datasets that are sized between 20-75k samples, MalConv2'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. Figures 1a and 1b show the diminishing accuracy and AUC respectively of MalConv2 when trained on progressively fewer data samples from the Ember dataset. In these results, samples chosen at random were removed to create progressively shrink the dataset. As shown, accuracy starts at 91% but dips below 80% when trained on subsets sized similarly

[1] 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.

```
Zenpak                          Sivis
inc eax      dec eax       nop              inc eax
inc ecx      dec ecx       nop              push edx
inc edx      dec edx       nop              xor edx, edx
inc ebx      dec ebx       xor eax, eax     pop edx
inc esp      dec esp       inc ebx          inc eax
inc ebp      dec ebp       dec ebx          dec eax
inc esi      dec esi       inc ecx          cmp 0x17b8ef93, eax
inc edi      dec edi       dec ecx          jne 0x407033
```

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

to existing free datasets, e.g., 75k samples and less. Similarly AUC starts at 97% but dips below 86%. These results indicate the data-hungry nature of ML-based malware detectors, and highlight the heterogeneity in data samples in large datasets.

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 representative case studies that illustrate our findings, before describing more general takeaways.

```
Binary 1                              Binary 2
push ebx                              mov eax,ds:0x423e88
push esi                              push ebx
mov esi,DWORD PTR [ebp+0x8]           push esi
push edi                              mov esi,DWORD PTR [ebp+0x8]
mov eax,ds:0x470208                   push edi
push 0x7                              push 0x7
pop ecx                               pop ecx
lea edi,DWORD PTR [ebp-0x2c]          lea edi,DWORD PTR [ebp-0x28]
```

Figure 3: Snippets from two binaries in the same "Install-Monster" family that exhibit minor differences due to code obfuscations.

Case Studies. Figure 2 shows code snippets from two different malware families in the Ember dataset: the Zenpak malware family, and the Sivis malware family. These snippets exhibit instructions that alter the state of the program and later reverse those modifications. Figure 3 depicts snippets from two binaries from the Ember dataset that belong to the same family but have minor differences from obfuscations.

Case study I. Figure 2 shows code snippets from two different malware families in the Ember dataset: the Zenpak malware family, and the Sivis malware family.[2] The first binary from Zenpak uses a code obfuscation technique called junk code insertion [39]. 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 `dec` instructions (lines 9-16) that decrement them.

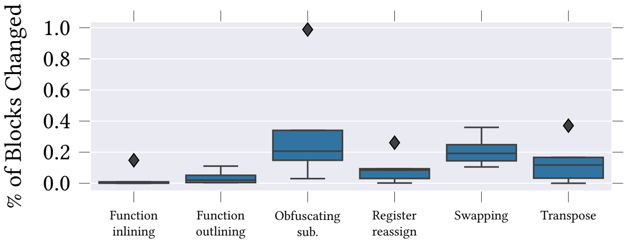


Figure 4: Percentages of code blocks in Ember's binaries that are affected by different code transformations.

The binary from Sivis 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 binary embeds another code obfuscation technique called opaque predicates [39], 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` ≠ `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.

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

(1) ***Semantics-preserving code transformations.*** Malware authors routinely alter their malicious programs using code obfuscation techniques that preserve program behavior. The reason is intuitive: 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 to preserve its malicious behavior while deviating from the patterns used in existing malware detectors.

(2) ***Combinations of transformations.*** Performing code transfor-

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

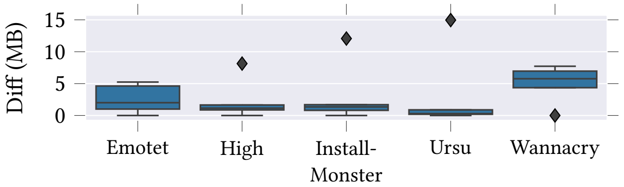


Figure 5: Pairwise byte diff results between binaries in five representative malware families.

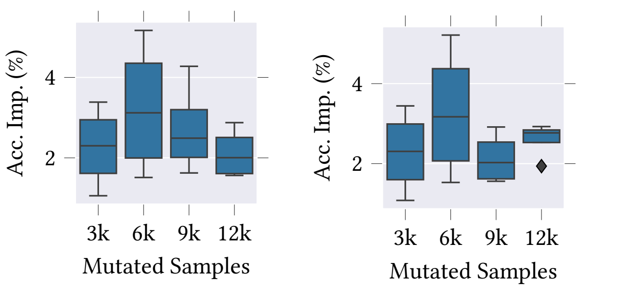
mations is fruitful as such transformations are often (logically) complementary, and the effect of each transformation depends on interactions between the transformation logic and binary code (ranges shown in Figure 4). Additionally, we find that, to further boost diversity with multiple transformations, 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, 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. 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 small malware datasets: data augmentation via semantics-preserving transformations. In other words, we aim to grow small datasets by performing different combinations of semantics-preserving code transformations on varying numbers of blocks in the constituent binaries. Doing so mimics the techniques that malware authors use to sidestep malware detectors over time [7], and yield data similar to that in (proven) large datasets. We employ further code transformations done by optimizing compilers to generate new benign binaries. Perhaps 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). §4 describes how our system, Marvolo, practically realizes this approach.

4 MARVOLO

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. 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.



(a) Accuracy on in-distribution. **(b) Accuracy on novel malware families only.**

Figure 6: Results of Marvolo augmented training when testing on in-distribution data from Ember (a), novel malware families not seen in training (b).

Marvolo currently supports 10 different semantics-preserving code transformations that cover the set of mutations we observed in our analysis of the popular Ember dataset (§3), as well as commonly used code obfuscations [8, 20, 39] and transformation techniques employed by off-the-shelf optimizing compilers [4, 11]. Supported transformations include junk code insertion and instruction swapping (both described in §3), as well as instruction substitution which replaces an instruction with a (more complex) sequence of instructions that is semantically equivalent. 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 A provides a comprehensive overview of the transformations that Marvolo supports, as well as the logistics to carrying out each one.

Limitations. Despite the promise of Marvolo, there are several limitations. First, new code alterations used by malware authors are only caught once detectors are presented with data samples or signatures that identify them. Marvolo does not address this problem, and instead aims to maximize the utility of the data samples that practitioners have access to at any time. Second, non-trivial engineering is required for extensions to new platforms (e.g., Android).

5 EVALUATION

To evaluate Marvolo, we used the recent MalConv2 CNN-powered malware detector [29]. For context, MalConv2's model spans 5 layers, with an embedding layer that maps bytes to vectors, and then a series of convolutional and recurrent layers. Our experiments consider 2 main datasets: (1) the high-accuracy commercial Ember dataset that includes 1.1M samples (800K after removing ill-formed binaries), and (2) the small-scale (free) Brazilian malware dataset [15] 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, directly 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 full Brazilian dataset; we train a separate MalConv2 model for each case. Our subsets consist of 10-30K samples with malicious samples coming from only several different malware families, which is consistent with the dataset size and composition that many malware research groups currently work with [23]. While the Ember training subset and test set contain binaries of the same family, we also perform experiments with a smaller test set that only consists of malware families that were not present in the training set to evaluate how well Marvolo gets MalConv2 to generalize. While we would have preferred to experiment with augmenting more datasets, we are constrained since many existing datasets do not contain raw binaries as mentioned in 2.

For each dataset, we train MalConv2 to convergence, routinely around 5 epochs. 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. Unless otherwise noted, Marvolo employs combinations of all 10 of its supported transformations and generates a set of mutated binaries (split evenly across malicious and benign files); the description of each experiment specifies the number of those mutated samples considered during retraining. 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 by their perceived maliciousness [10, 25]. Malware that is perceived to be more dangerous than others are then quickly identified and quarantined. Thus, A high AUC is crucial since it corresponds to a successful ranking of most malicious files above benign files. We run each experiment four times and report on the distributions.

5.1 Overall Accuracy Improvements

Figure 6a 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). Accuracy improvements range from 1–5% atop the baseline accuracy of 61.3% achieved when considering the unmodified Ember dataset alone. Perhaps more importantly, 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. Additionally, we find that our augmented datasets deliver AUC improvements of 5% – 10%. 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 [15] 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 and AUC 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 6b 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%). Marvolo also improves AUC on unseen malware families by 5–10%. The underlying reason for these improvements is that code transformations provide a discernible pattern for MalConv2 to link across diverse binaries in different families. In light of these results, we provide further analysis of Marvolo in Appendix B.

6 CONCLUSION

Marvolo is a data augmentation engine that leverages 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 is its ability to (safely) propagate labels across input and output binary samples. Experiments using commercial malware datasets and a recent ML-driven malware detector show that Marvolo boosts accuracies by up to 5% and AUC by up to 10%.

REFERENCES

[1] 2009. Malware detection using statistical analysis of byte-level file content. ACM Press, 23–31. https://doi.org/10.1145/1599272.1599278

[2] 2020. Cost of a Data Breach Report 2020). https://www.ibm.com/security/digital-assets/cost-data-breach-report/. Accessed: 2021-08-07.

[3] 2020. Malwarebytes State of Malware Report. https://www.malwarebytes.com/resources/files/2020/02/2020_state-of-malware-report.pdf. Accessed: 2021-09-28.

[4] 2021. Code Obfuscation. https://en.wikibooks.org/wiki/X86_Disassembly/Code_Obfuscation. Accessed: 2021-08-20.

[5] 2021. Ghidra Software Reverse Engineering Framework. https://ghidra-sre.org/. Accessed: 2021-09-28.

[6] 2021. Global Ransomware Damage Costs Predicted To Reach $20 Billion (USD) By 2021. https://bit.ly/3j3bTEB. Accessed: 2021-10-06.

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

[8] 2021. The Tigress diversifying C virtualizer. https://tigress.wtf/. Accessed: 2021-08-28.

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

[10] O. A. Abedelaziz Mohaisen. 2013. Unveiling Zeus: Automated Classification of Malware Samples. In *WWW Companion*.

[11] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 1986. Compilers: Principles, Techniques, and Tools.

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

[13] N. Bhagat and B. Arora. 2018. Intrusion Detection Using Honeypots. In *PDGC*.

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

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

[16] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu. 2018. Gotcha - Sly Malware! Scorpion: A Metagraph2vec Based Malware Detection System. In *ACM SIGKDD*. 253–262. https://doi.org/10.1145/3219819.3219862

[17] Ferhat Ozgur Catak, Javed Ahmed, Kevser Sahinbas, Zahid Hussain Khand. 2021. Data augmentation based malware detection using convolutional neural networks. In *PeerJ Computer Science*.

[18] R. Harang and E. M. Rudd. 2020. SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection. arXiv:2012.07634 [cs.CR]

[19] K. Z. Jason Wei. 2020. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. In *IJCNLP*.

[20] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. 2015. Obfuscator-LLVM — Software Protection for the Masses. In *SPRO*.

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

[22] M. Krcal, O. Svec, O. Jasek, and M. Balek. 2018. Deep Convolutional Malware Classifiers Can Learn From Raw Executables And Labels Only. In *ICLRW*.

[23] Michael R. Smith, Nicholas T. Johnson, Joe B. Ingram, Armida J. Carbajal, Bridget I. Haus, Eva Domschot, Ramyaa Ramyaa, Christopher C. Lamb, Stephen J. Verzi, W. Philip Kegelmeyer. 2020. Mind the Gap: On Bridging the Semantic Gap between Machine Learning and Malware Analysis. In *ACM CCS AISec.*

[24] J. M. d. R. Niall McLaughlin. 2021. Data Augmentation for Opcode Sequence Based Malware Detection. In *arXiv 2106.11821*.

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

[26] E. Raff and C. Nicholas. 2017. An Alternative to NCD for Large Sequences, Lempel-Ziv Jaccard Distance. In *ACM SIGKDD*. 1007–1015.

[27] E. Raff and C. Nicholas. 2020. A Survey of Machine Learning Methods and Challenges for Windows Malware Classification. In *ML-RSA*.

[28] E. Raff, R. Zak, G. L. Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt. 2020. Automatic Yara Rule Generation Using Biclustering. In *ACM CCS AISec*.

[29] Raff, Edward and Fleshman, William and Zak, Richard and Anderson, Hyrum S. and Filar, Bobby and McLean,Mark. 2021. Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection. In *AAAI*.

[30] Ri Or-meir, Nir Nissim, Yuval Elovici, Lior Rokach. 2018. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. In *ACM Computing Survey 52, 5, Article 88)*.

[31] R. Ronen, M. Radu, C. Feurstein, E. Yom-Tov, and M. Ahmadi. 2018. Microsoft Malware Classification Challenge. In *arXiv 1802.10135*.

[32] A. Tamersoy, K. Roundy, and D. H. Chau. 2014. Guilt by Association: Large Scale Malware Detection by Mining File-relation Graphs. In *ACM SIGKDD*. 1524–1533. https://doi.org/10.1145/2623330.2623342

[33] T. Z. Tebogo Mokoena. 2017. Malware Analysis and Detection in Enterprise Systems. In *IPSA/IUCC*.

[34] D. Votipka, S. M. Rabin, K. Micinski, J. S. Foster, and M. M. Mazurek. 2019. An Observational Investigation of Reverse Engineers ' Processes. In *USENIX Security Symposium*.

[35] D. Votipka, S. M. Rabin, K. Micinski, J. S. Foster, and M. M. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *USENIX Security*.

[36] R. Wartell, Y. Zhou, K. W. Hanlen, M. Kantarcioglu, and B. Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *ECML PKDD*.

[37] Y. Ye, T. Li, Y. Chen, and Q. Jiang. 2010. Automatic Malware Categorization Using Cluster Ensemble. In *ACM SIGKDD*. 95–104. https://doi.org/10.1145/1835804.1835820

[38] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu. 2011. Combining File Content and File Relations for Cloud Based Malware Detection. In *ACM SIGKDD*. 222–230. https://doi.org/10.1145/2020408.2020448

[39] I. You and K. Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *BWCCA*.

A 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 [20]. 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 $a = (b \,\&\, c)|(b\hat{}c)$.

Optimizing instruction substitution. Replace an instruction with an equivalent instruction that optimizing compilers often emit [4]. 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 [39]. 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.

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 [36], 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. [20].

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.

B ANALYZING MARVOLO

Importance of number of binaries mutated. Figures 6a and 6b show MARVOLO's performance as the number of added mutated binaries changes. As discussed, the benefits from MARVOLO's mutations come early from the perspective that most accuracy wins can be

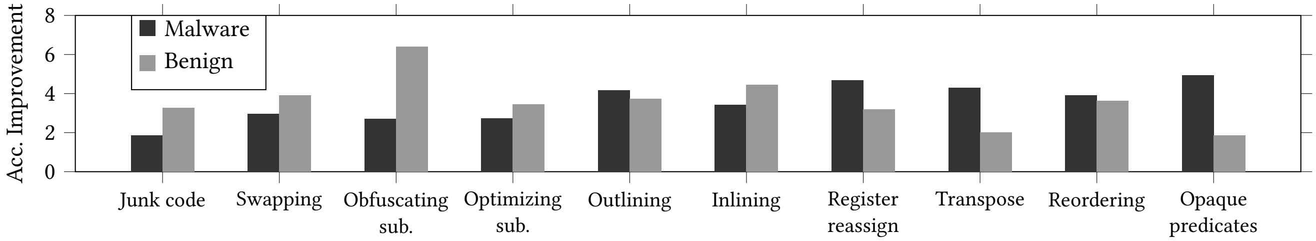


Figure 7: MalConv2's accuracy improvements when using a version of Marvolo that only performs a single type of semantics-preserving code transformation during mutation. Results are for adding 1K mutated samples in each case to the Ember dataset.

Original

```
add eax, ebx
sub ecx, 0x7c21
ret
```

Mutated

```
sub ecx, 0x7c21
add eax, ebx
ret
```

realized by using only a small fraction of the overall dataset as input; we observe this trend over multiple datasets of different sizes. 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).

B.1 Analyzing Marvolo

Importance of different transformations. To study the effect that each of Marvolo's ten code transformations have on accuracy improvements, for each transformation (in isolation), we generated two sets of 1K mutated samples: one where all mutated samples were benign, and one where all mutated samples were malicious. Figure 7 shows the accuracy improvements for MalConv2 running on the Ember dataset plus each of the 20 mutated datasets (one at a time). For benign files, instruction swapping, obfuscating substitutions, and function inlining yielded the largest accuracy wins, with 4%, 6%, and 4% performance gains, respectively. For malicious files, register reassignment, code transposition, and opaque predicate insertion were the most fruitful with 5%, 4%, and 5% performance gains, respectively. The reason is that the latter trio of transformations are more invasive (i.e., they lead to larger code alterations and resultant diffs), and are hence more often applied by malware authors to circumvent recently employed detection patterns.

Further, 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 currently 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.

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.