An Adversarial Malware Evasion Attack using Neural Machine Translation

***Abstract*—In recent decades, malware has become a significant problem for an increasingly digitized society. Amidst the explosion of machine learning (ML) and deep learning (DL), feasible malware detection at scale has become more effective than ever. However, ML and DL systems are notoriously vulnerable to adversarial attacks. In the case of an adversarial evasion attack against a malware detection system, an adversary would carefully perturb an executable such that it maintains its malicious functionality, but is able to evade the classifier. To understand this class of threats before malicious actors do, the cybersecurity community has made a significant effort to develop techniques to craft adversarial malware. By doing this, security experts can design defense protocols against adversarial attacks and hope to remain one step ahead of adversaries. In our work, we propose a novel adversarial evasion attack that operates by directly modifying the source code of malicious software. We believe that we can identify pieces of code within malware that detection systems find suspicious using DL explainability theory. We then intend to substitute these suspicious sections of code with semantically equivalent code that appears benign. To perform the replacement, we propose a sequence-to-sequence model based upon large-language-modeling (LLM) for assembly-level code. We argue that our attack can be conjoined with any other adversarial attack to result in a more powerful attack with an improved evasion rate. We propose a thorough experiment to evaluate the effectiveness of our attack against a variety of hardened detection systems.**

I. Background

The arms race between malware and the detection of malware is an ongoing struggle. Machine learning (ML) and deep learning (DL) detection systems have been shown to be an effective tool to identify malicious software statically, e.g., the DREBIN [1], MalConv [2] and EMBER [3] detectors. While able to perform well on unseen examples, these models are vulnerable to adversarial evasion attacks [4]. In the case of a practical adversarial malware evasion attack, the adversary modifies the malware binary such that it maintains the correct format, is executable, preserves its malicious functionality [5], yet cannot be identified by the detection model. It is important to discover the ways adversaries with ill intentions can conduct these attacks. To this end, the cybersecurity community has invested a great deal of effort to develop adversarial attacks [5]–[8]. Proposing such attacks allows researchers to devise countermeasures to defend against them [9].

Meanwhile, in the late 2010s, the field of natural language processing (NLP) was revolutionized by two important advances: the Transformer architecture for sequence processing [10] and task agnostic pre-training of large language models (LLMs) like ELMo [11] prior to task specific fine-tuning. These advances resulted in a series of models that achieved state of the art performance at a variety of NLP tasks, e.g., GPT [12] and BERT [13]. Since source code and natural language contain structural similarities, researchers applied these new strategies to tasks like automatic code documentation, e.g., CodeBERT [14], code generation, e.g., codex [15], and transcompilation [16]. Most of the research related to source code has been done for high-level languages like Python since there is more demand for code tools in these languages.

Recently, these advances have been applied to improve low level source code representations. Finding an effective representation for code at this level has long been a challenge. An effective representation of machine code is useful for a variety of applications, such as binary code similarity detection [17], function type signatures inference [18], value set analysis [19], and malware detection from raw bytes [2], [20]. Previous works have used simple raw byte encodings [19] or assembly-level encodings [17], neither of which can provide rich semantic knowledge about the meaning of the instructions. To achieve representations that do provide a more sophisticated understanding of the instructions, researchers have begun to start using LLMs from NLP. [21] and [22] respectively introduced PalmTree and BinBERT, LLMs for assembly code based on the BERT [13] architecture and unsupervised pre-training paradigm. [23] adopted the CodeBERT from [14] to generate assembly code from natural language descriptions.

II. Proposal

In this work, we propose a novel adversarial evasion attack to evade raw byte malware classifiers. Our attack uses explainability to identify suspicious sections of executable code within the binary and substitutes them with semantically equivalent code that appears harmless. To perform the substitutions, we propose two sequence to sequence (seq2seq) models based on techniques from NLP. We evaluate our attack on several malware detection systems and compare it with other adversarial attacks. Since our attack only modifies the code portion of malware, we argue that it should be used in conjunction with a complementary attack that modifies the headers and metadata of malware.

The remainder of this proposal is organized as follows: section II-A describes our proposed models to perform the substitution, section II-B describes how we identify which sections of malware a detector flags as suspicious, section II-C describes our process to gather data for training our models, section II-D describes how we launch our attack, section II-E describes our experiments, and section II-F evaluates our approach as a whole.

A. Model Architectures

Our first model is based upon unsupervised neural machine translation (NMT) models. Supervised NMT generally requires substantial parallel training corpora to learn high-performing models, which has been an ongoing challenge in this field [24]. Since such corpora are often unavailable and expensive to create, a variety of unsupervised techniques have been used to perform translation with only monolingual corpora. Many of the most successful NMT techniques rely on two training objectives. First, denoising auto encoding (DAE) [25] is used as a form of language modeling to teach the model the structure of language. DAE adds noise to source examples, from which the model attempts to reconstruct the original source input. Second, backtranslation [26] is used to actually translate between a source and target language. Backtranslation trains a target-to-source model in conjunction with the primary source-to-target model. The target-to-source model generates noisy source examples, from which the source-to-target model attempts to construct the target from. As an alternative to backtranslation, so-called "on-the-fly" backtranslation has been proposed as a learning objective that forgoes the target-to-source model entirely [27], [28]. This training objective uses the source-to-target model from the previous epoch to generate noisy target predictions, then tasks the model to reconstruct the source from the target predictions. Recently, the TransCoder model, built on DAE and backtranslation, has been used to translate code from one language to another in a completely unsupervised fashion [16], [29], [30]. We experiment with several TransCoder-inspired architectures using these different variations of the backtranslation training objectives in our model development.

Our second model uses a more creative approach to perform the substitution, namely, it frames the substitution as a dual summarization/generation task, rather than a translation task. First, a code summarization model creates precise natural language (NL) descriptions from input code. Second, a code generation model takes NL descriptions and generates code from them. To ensure the code it produces appears benign, our generation model is trained to match the statistical properties of benign executables, discussed in II-B. This model is supported by the related work from [31] and [23]. Although they used the technique as part of backtranslation within a broader NMT framework, [31] demonstrated that a dual code summarization/generation model could help translate between Java and Python code. [23] demonstrated that assembly code could be generated from NL. To our knowledge there has not been any research that summarizes assembly level source code, so our assembly code summarization model based on {FIXME: citation please!} is the first of its kind. We pair this with a code generation model similar to [23] based on CodeBERT [14] to complete the code-to-NL-to-code pipeline.

By incorporating nondeterminism into each model, we can generate multiple candidate substitutions for any given input. This gives us many possible options for substitution, thereby improving our chances that one of them will successfully deceive the classifier. Previous works using code translation and generation generally deal with shorter sequences {FIXME: citation please!}, which makes standard sequence processing architectures feasible. Our models are expected to handle long sequences of assembly level code, which is much less concise than typical programming languages like Python, Java, or C. For this reason, we use the Reformer [35], a Transformer architecture shown to be successful at processing sequences of up to 64,000 units, for processing our sequences.

B. Identification and Explainability

To identify which parts of a malware executable are most influential to the classifier, we monitor the magnitude of the classifier's gradients as was done in [36]–[38]. In [38], the authors found that the header and metadata of PE malware was most influential in the classifiers decision, while [37] found that the classifier made its decision based upon all parts of a PE file. In our work, we seek to clarify this discrepancy in the literature. Although we can identify what code within the malware appears malicious to the classifier, we still need a qualitative understanding of what makes this code appear malicious to train our second proposed model. Several works have shown that opcode frequencies can be indicative of malicious software [32]–[34]. This is admittedly a weak qualitative understanding of what makes individual chunks of code appear suspicious, but we leave developing a deeper understanding for future work.

C. Creating Datasets

To train our first model, we need to construct two monolingual corpora of malicious and benign code snippets. We begin with a labeled dataset of malicious and benign executables. We propose two methods to build our corpora. The first (method 1) naively builds the malicious corpora from the malicious executables only and the benign corpora from the benign executables only. First, the executables are disassembled into an intermediate representation. Next, we extract only the portions of the executable containing the code itself. Finally, we use a sliding-window approach to generate fixed-size snippets of code. We aggressively oversample by using multiple window sizes of different powers of 2. The second (method 2) does not use the class labels of the executables themselves when building the corpora. Instead, it uses the raw byte classifier targeted to attack to decide which chunks of code should belong in the malicious corpora vs the benign corpora. Chunks of code the classifier finds suspicious are placed in the malicious corpora, while chunks of code the classifier does not find suspicious are placed in the benign corpora. Both of these methods result in the two desired monolingual corpora of malicious and benign code snippets.

Our second model relies on the statistical properties of malicious vs benign source code, which we derive from the malicious and benign corpora created to train the translation model. Unlike the NMT-based model, this model takes a supervised learning approach and requires labeled data of as-

sembly code and corresponding natural language descriptions. {FIXME: develop this idea}.

D. Attack Strategy

Once we have a model that can modify malicious source to make it appear benign, we can use it to launch a white-box adversarial attack. We propose an attack designed against raw byte CNN classifiers, since existing works demonstrate how to analyze these models to determine what regions of the malware they find most suspicious. Theoretically, variations our attack could be used against any classifier as long as specific portions of the malware's code can be identified as being significant to the classifier's decision. Our attack works quite simply. We use the explainability methods described in section II-B to identify and rank regions of the malware code most suitable for substitution. Then, using the model described in II-A, we continually substitute these regions until the adversarial example is labeled as benign.

E. Experiment

When evaluating our adversarial attack, we seek to answer the following questions:

- RQ1: How well does our attack perform against a specific classifier in a white-box scenario?
- RQ2: Will the adversarial examples generated in an attempt to evade one classifier evade a radically different type of classifier?
- RQ3: Is our attack more effective against classifiers that are restricted to learning from only the code portions of malware?
- RQ4: Does our attack outperform other adversarial attacks?
- RQ5: Can our attack be combined with other adversarial attacks to create a more powerful attack?

For all experiments, we report the evasion rate of our attacks, as was done in {FIXME: citation please!}. Researchers concerned with generating practical adversarial malware {FIXME: citation please!} frequently use Cuckoo Sandbox {FIXME: citation please!} to verify the attack did not compromise the functionality of the malware. We verify the malicious functionality was preserved for examples that evade the classifier.

To answer RQ1, we launch white-box attacks against two state of the art raw byte malware classifiers: MalConv2 and MalConvGCG [20]. To answer RQ2, we use the adversarial examples generated from our attack and attempt to evade the EMBER classifier [3], a Gaussian Boosted Decision Tree model that uses manually extracted features as opposed to raw bytes. To answer RQ3, we modify the MalConv2, MalConvGCG, and EMBER classifiers to only use features extracted from the code sections of the malware and repeat the experiments from RQ1 and RQ2.

To answer RQ4, we introduce a second widely used attack, {FIXME: citation please!}, and compare its evasion rate against the classification systems introduced in previous experiments. Since this attack primarily modifies the header of an executable and our attack alters the code, we believe that the attacks combined will be more effective than either attack separately. To answer RQ5, we evaluate the combined attack against all our classifiers.

F. Discussion

There are several aspects of this proposal that are either unexplored or worthy of further consideration:

- How can we ensure the code generation model produces executable code? Furthermore, how can we ensure the model produces semantically correct code? Can we use unit tests in some way? For example, can we use existing datasets containing source code and corresponding unit tests, then compile-disassemble these datasets to attain assembly level unit-tested code, then leverage this in some sort of pretraining fashion? Alternatively, can we use automatic unit test generation tools like EvoSuit? We could use such tools to ensure generated target code matches the semantics of source code.
- How can we make the attack work in a single shot? Is there a way we can incrementally modify the source code of the malware while monitoring the classifier's gradients? Alternatively, we could simple use a brute force approach, and repeatedly modify the malware until it deceives the detector. We could produce 100 adversarial examples and report what percentage of them evade the classifier, similar to the codex experiments [15].
- When building the corpora, should we experiment with some form of structured segmentation, such as using function identification techniques [39]? One pass with a single-sized sliding window or multiple passes with sliding windows of multiple sizes? Should we use the sliding window approach at all, or simply use fixed-size chunk segmentation?

References

[1] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

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

[3] H. S. Anderson and P. Roth, "Ember: an open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

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

[5] X. Ling, L. Wu, J. Zhang, Z. Qu, W. Deng, X. Chen, C. Wu, S. Ji, T. Luo, J. Wu *et al.*, "Adversarial attacks against windows pe malware detection: A survey of the state-of-the-art," *arXiv preprint arXiv:2112.12310*, 2021.

[6] D. Maiorca, B. Biggio, and G. Giacinto, "Towards adversarial malware detection: Lessons learned from pdf-based attacks," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–36, 2019.

[7] D. Park and B. Yener, "A survey on practical adversarial examples for malware classifiers," in *Reversing and Offensive-oriented Trends Symposium*, 2020, pp. 23–35.

[8] D. Li, Q. Li, Y. Ye, and S. Xu, "Arms race in adversarial malware detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–35, 2021.

[9] ——, "A framework for enhancing deep neural networks against adversarial malware," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 1, pp. 736–750, 2021.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[11] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," 2018. [Online]. Available: https://arxiv.org/abs/1802.05365

[12] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[16] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 601–20 611, 2020.

[17] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[18] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99–116.

[19] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, "{DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1787–1804.

[20] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar, and M. McLean, "Classifying sequences of extreme length with constant memory applied to malware detection," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 11, 2021, pp. 9386–9394.

[21] X. Li, Y. Qu, and H. Yin, "Palmtree: learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[22] F. Artuso, M. Mormando, G. A. Di Luna, and L. Querzoni, "Binbert: Binary code understanding with a fine-tunable and execution-aware transformer," *arXiv preprint arXiv:2208.06692*, 2022.

[23] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Can we generate shellcodes via natural language? an empirical study," *Automated Software Engineering*, vol. 29, no. 1, pp. 1–34, 2022.

[24] F. Guzmán, P.-J. Chen, M. Ott, J. Pino, G. Lample, P. Koehn, V. Chaudhary, and M. Ranzato, "The flores evaluation datasets for low-resource machine translation: Nepali-english and sinhala-english," *arXiv preprint arXiv:1902.01382*, 2019.

[25] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." *Journal of machine learning research*, vol. 11, no. 12, 2010.

[26] R. Sennrich, B. Haddow, and A. Birch, "Improving neural machine translation models with monolingual data," *arXiv preprint arXiv:1511.06709*, 2015.

[27] G. Lample, A. Conneau, L. Denoyer, and M. Ranzato, "Unsupervised machine translation using monolingual corpora only," *arXiv preprint arXiv:1711.00043*, 2017.

[28] M. Artetxe, G. Labaka, E. Agirre, and K. Cho, "Unsupervised neural machine translation," *arXiv preprint arXiv:1710.11041*, 2017.

[29] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample, "Dobf: A deobfuscation pre-training objective for programming languages," *Advances in Neural Information Processing Systems*, vol. 34, pp. 14 967–14 979, 2021.

[30] B. Roziere, J. M. Zhang, F. Charton, M. Harman, G. Synnaeve, and G. Lample, "Leveraging automated unit tests for unsupervised code translation," *arXiv preprint arXiv:2110.06773*, 2021.

[31] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Summarize and generate to back-translate: Unsupervised translation of programming languages," *arXiv preprint arXiv:2205.11116*, 2022.

[32] A. Yewale and M. Singh, "Malware detection based on opcode frequency," in *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*. IEEE, 2016, pp. 646–649.

[33] Y. Kucuk and G. Yan, "Deceiving portable executable malware classifiers into targeted misclassification with practical adversarial examples," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 341–352.

[34] X. Li, K. Qiu, C. Qian, and G. Zhao, "An adversarial machine learning method based on opcode n-grams feature in malware detection," in *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*. IEEE, 2020, pp. 380–387.

[35] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," *arXiv preprint arXiv:2001.04451*, 2020.

[36] S. E. Coull and C. Gardner, "Activation analysis of a byte-based deep neural network for malware classification," in *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2019, pp. 21–27.

[37] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," *arXiv preprint arXiv:1901.03583*, 2019.

[38] S. Bose, T. Barao, and X. Liu, "Explaining ai for malware detection: Analysis of mechanisms of malconv," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[39] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX security symposium (USENIX Security 15)*, 2015, pp. 611–626.