The Power of Words: Generating PowerShell Attacks from Natural Language

Pietro Liguori, Christian Marescalco, Roberto Natella, Vittorio Orbinato, and Luciano Pianese, *DIETI, Università degli Studi di Napoli Federico II*

https://www.usenix.org/conference/woot24/presentation/liguori

This paper is included in the Proceedings of the 18th USENIX WOOT Conference on Offensive Technologies.

August 12–13, 2024 • Philadelphia, PA, USA

ISBN 978-1-939133-43-4

Open access to the Proceedings of the 18th USENIX WOOT Conference on Offensive Technologies is sponsored by USENIX.

The Power of Words: Generating PowerShell Attacks from Natural Language

Pietro Liguori*, Christian Marescalco**, Roberto Natella*, Vittorio Orbinato*, Luciano Pianese*
DIETI, Università degli Studi di Napoli Federico II, Naples, Italy
**{pietro.liguori, roberto.natella, vittorio.orbinato, luciano.pianese}@unina.it*
***c.marescalco@studenti.unina.it*

Abstract

As the Windows OS stands out as one of the most targeted systems, the *PowerShell* language has become a key tool for malicious actors and cybersecurity professionals (e.g., for penetration testing). This work explores an uncharted domain in AI code generation by automatically generating offensive PowerShell code from natural language descriptions using Neural Machine Translation (NMT). For training and evaluation purposes, we propose two novel datasets with PowerShell code samples, one with manually curated descriptions in natural language and another code-only dataset for reinforcing the training. We present an extensive evaluation of state-of-the-art NMT models and analyze the generated code both statically and dynamically. Results indicate that tuning NMT using our dataset is effective at generating offensive PowerShell code. Comparative analysis against the most widely used LLM service ChatGPT reveals the specialized strengths of our fine-tuned models.

1 Introduction

Offensive security practices, such as red teaming and adversary emulation, play a crucial role by helping us to understand how attackers take advantage of vulnerabilities and how to mitigate attacks [1, 2]. In these attacks, cybersecurity professionals emulate malicious post-exploitation actions, such as credential stealing, lateral movement across accounts and machines, data obfuscation and exfiltration, and more [3].

As Windows stands out as one of the most targeted OS [4], the *PowerShell* language has become a key tool for both malicious actors and cybersecurity professionals. This language is widely used to perform attacks since it can perform complex actions, such as establishing connections and accessing OS services and APIs without the need to deliver a malicious binary executable or payload on the target machine (e.g., "fileless" malware), making them harder to detect [5–8].

Unfortunately, writing offensive code demands a high degree of expertise and effort, restricting the adoption of offensive security practices. Therefore, the rise of automatic *AI code generators* represents an appealing solution to unlock these practices to a broader spectrum of users [9].

AI code generators leverage ML models for Neural Machine Translation (NMT) to produce (offensive) code starting from inputs in Natural Language (NL), e.g., in the English language. The usage of NMT models is widespread across diverse software engineering tasks [10], yet their application in security-related scenarios is infrequent and not widely explored. This gap stems primarily from the lack of suitable corpora for training and evaluating code generators. The shortage of corpora for offensive code generation is an evident limitation: existing benchmarks [11–13] are derived from programming competitions and software interview questions (e.g., about algorithms and mathematics), or they focus on programs and languages that are not related to security (e.g., web applications in Python). Only a few security-oriented datasets are publicly available, targeting shellcodes in low-level programming languages [14]. As a result, there is a significant gap in the literature on offensive PowerShell code generation.

This work presents an assessment of AI code generators for PowerShell offensive code, a novel application of NMT. Given that generative models are predominantly trained on mainstream programming languages like Python and Java, we investigate strategies to repurpose these models for the PowerShell domain. To this aim, we adopt a combination of unlabeled and labeled datasets to train and evaluate models. Specifically, we first use a large collection of unlabeled (i.e., code only) samples of general-purpose PowerShell from various online repositories to pre-train ML models and refine their capabilities to comprehend and generate PowerShell code. Then, we build from scratch a manually annotated labeled dataset consisting of PowerShell code samples specifically crafted for security applications, which we pair with curated NL descriptions in English. We use this dataset to fine-tune three state-of-the-art NMT models (CodeT5+ [15], CodeGPT [16], and CodeGen [17]) to generate offensive PowerShell code. The dataset also serves as a ground truth for

the evaluation. We publicly share code, models [1] and datasets as open data[2] to encourage further experimentation on this topic.

To perform our experiments, we formulate four key research questions (RQs) aimed at evaluating the models' capabilities and the impact of the training strategies, performing static and execution analysis to assess the generated code, and comparing privately fine-tuned models with ChatGPT, the most widely used LLM service from OpenAI [18]. Table 1 summarizes the key findings of our analysis. To the best of our knowledge, this is the first work on the automatic generation of offensive PowerShell code from NL descriptions.

In the following, Section 2 discusses related work; Section 3 describes the research study; Section 4 shows the experimental results; Section 5 discusses the threats to validity; Section 6 discusses the ethical considerations; Section 7 concludes the paper.

2 Related Work

This work focuses on offensive code generation, involving machine translation techniques applied to the security domain for PowerShell code generation. Thus, we reviewed related literature in these areas.

ML for security-related PowerShell. Li *et al.* [19] designed a subtree-based de-obfuscation method and a semantic-aware PowerShell attack detection system. This work also demonstrates how the presented de-obfuscation method improves the performance of detection systems such as Windows Defender and Virus-Total. PowerDP [20] is a solution that aims to automatically identify malicious PowerShell commands through character distribution features and obfuscation multi-label classification also proposing a de-obfuscator method for recovering obfuscated commands. Even ML-based methodologies have arisen for detection purposes, as shown by Hendler *et al.* [21], who proposed several ML-based detectors demonstrating their effectiveness on malicious scripts. The authors also devised another solution [22] to achieve the same objective by retrieving information from Microsoft's AMSI interface. Mimura and Tajiri [23] presented a lighter methodology, restricting detection only to word embeddings. Mezawa *et al.* [24] proposed an evaluation methodology for ML-based detectors based on a word-level machine learning model. Given the effectiveness of Abstract Syntax Trees (ASTs) in detecting obfuscated PowerShell scripts, Rusak *et al.* [25] proposed a hybrid approach that combines ASTs and deep learning to enhance detection methods for high-level obfuscation PowerShell malicious programs. We remark that research of ML for PowerShell focuses on *defensive* uses (i.e., detecting and de-obfuscating attacks), but none of these studies analyzed the *offensive* uses of ML (i.e., generating attacks), which are also relevant for red teaming and adversary emulation purposes, and which are in the scope of this paper.

[1] HuggingFace repo
[2] GitHub repo

Analysis	Main Findings
Capability Assessment	• Models without fine-tuning (*zero-shot learning*) showed a limited ability to generate PowerShell code, often defaulting to Python syntax or incorrect PowerShell code. • The fine-tuning phase significantly enhanced the models' ability to generate syntactically correct and semantically relevant PowerShell code. Among the models, CodeT5+ and CodeGPT demonstrated notable improvements in generating offensive PowerShell code. • Pre-training on a large PowerShell corpus had a varying impact on different models. While pre-training generally improved CodeT5+ and CodeGPT, especially with a limited number of epochs for fine-tuning, CodeGen did not consistently benefit from pre-training.
Static and Execution Analysis	• All models achieved high syntax accuracy, indicating their strong capability to generate syntactically correct code. However, a significant number of warnings were identified, suggesting potential issues or suboptimal coding practices. • The execution analysis showed that, despite textual differences between the ground truth and the generated code, the models are still able to generate offensive PowerShell code closely aligned with the intended malicious activities, in terms of events occurring in the system (e.g., on the filesystem, network, registry).
Comparison with public AI model	• Our fine-tuned models outperform ChatGPT across all the metrics, showing that specializing the models on our fine-tuning dataset provides an advantage in the offensive PowerShell code generation task.

Table 1: Main findings.

Offensive Code Generation. Research on AI code generators for offensive security is still at an early stage. Gupta *et al.* [26] presented an outlook of the possibilities opened by ChatGPT for generating various types of cyber attacks, such as social engineering, phishing attacks, and malware creation. For each attack scenario, the paper shows qualitative examples of prompts submitted to ChatGPT, and the attack payloads generated as a result, including some snippets of PowerShell code. Similarly, Charan *et al.* [27] presented qualitative examples with ChatGPT and Google BARD to generate malicious

scripts (mainly in Python, Bash, and PowerShell) for the top 10 prevalent MITRE Techniques of 2022, showing the potential of these AI models for security applications. However, none of these studies systematically analyzed AI code generators, lacking in several aspects: (i) the evaluation was limited to a few examples, while systematic evaluation requires much larger datasets; (ii) the study lacked a ground truth for evaluating the correctness of generated code; (iii) they did not yet explore the potential of fine-tuning ML models for security-related code generation. The few studies in this direction focused on generating *exploits* in low-level languages (e.g., to attack memory management vulnerabilities). However, exploitation is only a limited part of the cyber kill chain, overlooking several more types of malicious code. Among these studies, Liguori *et al.* [28] proposed a dataset and approach for training and evaluating AI code generators for code security, by generating shellcodes in Assembly language. EVIL [29] automatically generates exploits for conducting code injection attacks via NMT by targeting both the generation of shellcodes in Assembly language and related Python code for encoding and obfuscating the shellcodes. DualSC [30] formalizes the automatic generation and summarization of shellcodes via a "Shallow" Transformer inspired by the T5 model and dual learning using the corpus provided by Liguori *et al.* [28]. ExploitGen [31] is an approach for generating exploit code in Python and Assembly based on the CodeBERT model. Differently from these studies, we presented a dedicated model for generating offensive PowerShell code, covering the entire cyber kill chain (e.g., including credential stealing, lateral movement, data exfiltration, and more tactics from the MITRE ATT&CK taxonomy). Moreover, we systematically analyzed the quality of generated PowerShell code by introducing a manually curated dataset to serve as a ground truth and evaluating the code statically and dynamically.

3 Research Study

The main objective of our research study is to understand whether NMT models can translate NL descriptions into code that accurately replicates the complexities of cyber attacks in PowerShell. This aspect is crucial as it explores the models' understanding of the unique syntax and semantics of this programming language.

Figure 1 provides an overview of this research study. We analyze various deep learning strategies to accurately generate code and introduce datasets to train and evaluate them. We study several state-of-the-art NMT models and introduce various approaches to evaluating the generated code, including the similarity of the generated code to ground truth and static and dynamic analysis of the code.

To help NMT models in the novel and ambitious task of generating PowerShell code from NL, we adopt a two-step process consisting of **pre-training** and **fine-tuning**. The pre-training phase aims to tailor NMT models (already pre-trained on other programming languages) in the generation of PowerShell code. Armed with the pre-trained models, we proceed to the fine-tuning phase. This iterative process refines the models' capabilities, enabling them to generate offensive PowerShell code from NL descriptions.

The main problem in using NMT models is to have a sufficient set of data and to use them effectively to train the models themselves. Recognizing the lack of suitable datasets for offensive PowerShell code generation, in this study, we collect a large set of PowerShell programs used for penetration testing and adversary emulation. In addition to the code, we create descriptions of these programs in English to allow the model to translate English into PowerShell code. This dataset was created manually to verify that the programs were related to security and to ensure that the English language descriptions were complete and consistent with the code. The dataset is labeled since each sample includes both the text to translate into code and the code expected to be produced by the model (ground truth).

The creation of labeled datasets is inevitably limited by the availability of PowerShell security programs and the need to manually create English language descriptions for each program. To increase the amount of training data, in this study, we investigate an additional strategy, fully automated, to build an extended dataset of PowerShell programs, collecting PowerShell programs and the related text from the web (for example, comments in the code or description accompanying the code). As the collection is fully automated, this second dataset is non-labeled. The dataset includes programs not strictly related to security but includes, in general, PowerShell code used for various purposes. This dataset still contributes to the ability to generate security code since it allows the model to learn from further examples how to generate syntactically valid PowerShell code and to correlate the PowerShell code with the English language. We use this dataset to pre-train the NMT models, carrying out additional unsupervised training rounds.

Table 2 reports the statistics of both datasets, in terms of size, unique number of tokens, and average number of tokens for NL descriptions (only for fine-tuning data) and code.

Finally, we evaluate the models as follows:

- *Capability Assessment*: We compare the textual similarity of the code generated by the models with a ground-truth reference through automatic metrics. These metrics are an appealing solution to estimate the generated code since they are easy to tune and time-saving, hence overcoming the limit of human evaluation, which poses practical challenges for large-scale assessments.

- *Static analysis*: We assess the generated code to ensure that it adheres to PowerShell programming conventions and does not contain syntax errors.

- *Execution analysis*: We evaluate the capability of the

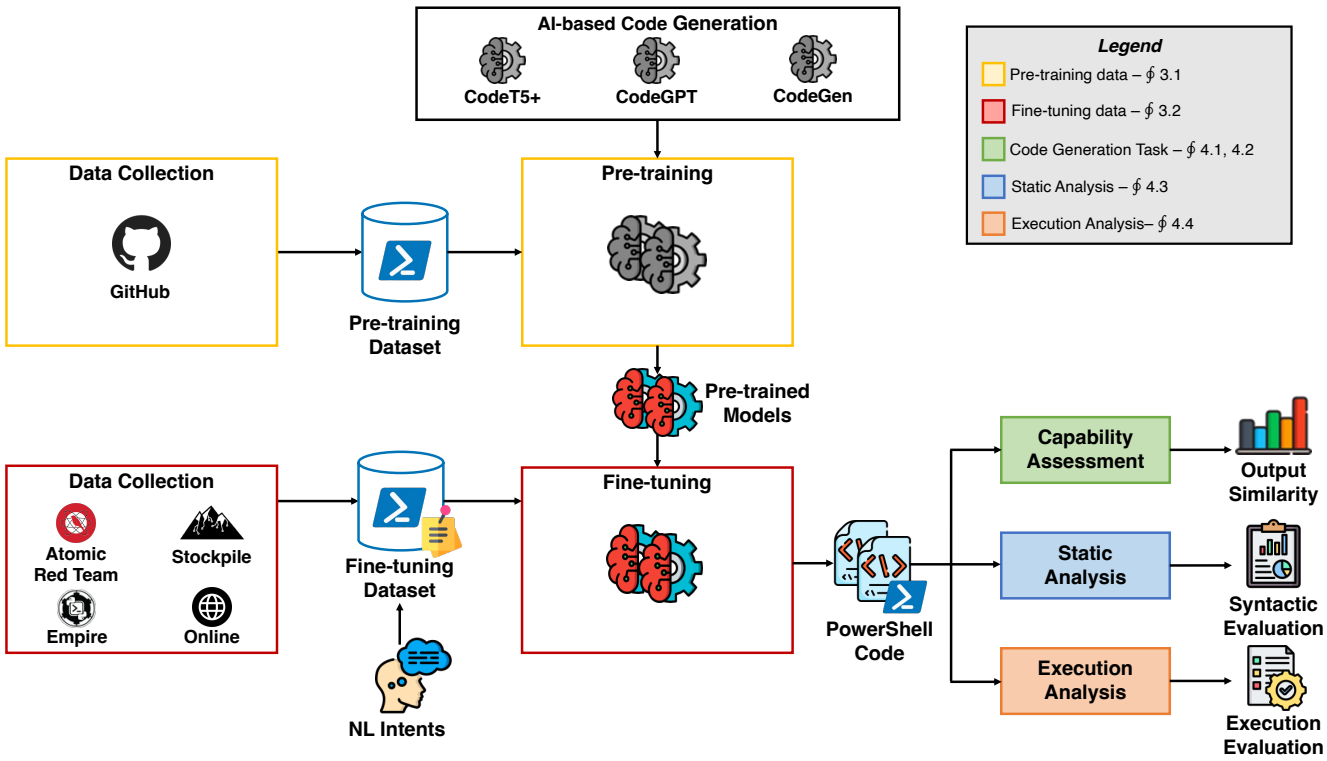

Figure 1: Overview of our research study.

generated offensive PowerShell code in executing malicious actions, replicating the behavior of the ground truth commands.

In the following of this section, we detail the pre-training (§ 3.1) and the fine-tuning data (§ 3.2), and the code generation task (§ 3.3).

3.1 Pre-training data (unlabeled)

Pre-training involves training the model on a large corpus of text data to learn general language representations before fine-tuning it for specific downstream tasks [32]. In other words, the parameters obtained from this step serve as a starting point for the later supervised training. Unsupervised or self-supervised pre-training is particularly attractive in the NMT context since large unlabeled data is available on the Internet. In this work, we leverage domain-adaptive pre-training (DAPT) [33]: given an NMT model pre-trained on massive, heterogeneous corpora, we perform additional rounds of unsupervised training with domain-specific data. Specifically, we leverage general-purpose PowerShell code for pre-training. The pre-training dataset aims to provide a valuable resource to enable the models' understanding of general-purpose PowerShell code. This dataset encompasses $\sim 90k$ samples extracted through the GitHub API. Specifically, we queried all the repositories containing PowerShell code from the last decade (2013-2023) to encompass a broad spectrum of PowerShell code, then parsed the extracted data to remove unnecessary information, such as duplicates (inside the same repository), and logging and echo commands. In addition, we filtered out all the PowerShell commands with sizes greater than 1024, ensuring the dataset maintains a balanced representation of code complexities. This collection encompasses a diverse array of PowerShell scripts, spanning various application domains such as system administration, automation, and network management. Including a wide range of scripts reflects the versatility of PowerShell as a scripting language and provides models with exposure to the diverse ways PowerShell is used across different use cases.

The pre-training process depends on the model architecture. For decoder-only models, i.e., CodeGPT and CodeGen, we chose *Causal Language Modeling (CLM)*, also referred to as Language Modeling, as the pre-training objective. CLM has been extensively used as a pre-training task for transformer-based decoder-only models [34], such as in the GPT series [35–37]. CLM refers to language models that predict the next token or sequence of tokens in a sentence in a causal or autoregressive manner, where the prediction for each token depends only on the preceding tokens. By using masking, the model only attends to the left context in a unidirec-

Statistic	Pre-training Dataset	Fine-tuning Dataset
Dataset size	89,814	1,127
Unique Intents	-	1,077
Unique Commands	79,410	1,121
Unique tokens (Intents)	-	2,273
Unique tokens (Commands)	85,342	17,463
Avg. tokens per Intent	-	15.97
Avg. tokens per Command	12.71	15.49

Table 2: Statistics of the pre-training and fine-tuning datasets. The pre-training dataset does not contain NL descriptions (intents).

tional manner, ensuring that it cannot see "into the future". In the probabilistic framework, starting from the text sequence $x = (x_1, x_2, x_3, \ldots, x_T)$, where x is the original sentence and x_t $(t = 1, 2, \ldots, T)$ is the t-th token, and T is the sequence length, an autoregressive model factorizes the likelihood of the input text sequence as $p(x) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$, where p is the likelihood of the input text sequence [38]. Finally, models are evaluated by token-level accuracy. For CodeT5+, the pre-training objective is *Masked Language Modeling (MLM)*, as recent works show its effectiveness in code understanding tasks [39]. MLM refers to the prediction of missing tokens in a sentence based on the context provided by the surrounding tokens. Unlike the left-to-right language model pre-training, MLM considers both the left and right context. The approach is inspired by BERT [40], where 15% of the tokens in the encoder inputs are randomly replaced with sentinel token [MASK], and the decoder is tasked with recovering these tokens to reconstruct the complete snippet. The model is evaluated by token level accuracy only on the masked-out tokens.

3.2 Fine-tuning data (labeled)

The overarching purpose of this dataset is to serve as a comprehensive resource for training models in the translation of NL intents, i.e., descriptions of code snippets, into executable security-oriented PowerShell commands. Specifically, we focus on offensive PowerShell code, a key resource for cybersecurity exercises since Microsoft Windows represents the most targeted OS. By encompassing a wide array of sources, the dataset aims to expose models to the intricacies of real-world cybersecurity scenarios, enabling them to understand and generate PowerShell commands that align with those typical of cybersecurity operations. This holistic approach strives to ensure that models trained on this dataset are well-equipped to handle the complexities of real-world tasks and contribute meaningfully to offensive code generation, specifically PowerShell commands.

The dataset, consisting of 1,127 samples of PowerShell commands, is meticulously curated from the following sources:

- *Atomic Red Team* [41]: renowned for its library of tests mapped to the MITRE ATT&CK framework[3] [42], serves the purpose of replicating real-world adversarial tactics, techniques, and procedures (TTPs). This inclusion provides the dataset with a foundation rooted in a standardized and widely accepted framework, ensuring that the PowerShell commands align with recognized cybersecurity methodologies.

- *Stockpile* [43]: is a plugin for the CALDERA cybersecurity framework [1, 44] developed by MITRE and introduces a layer of sophistication by incorporating structured data integral for adversary emulation. Therefore, the dataset does not encompass raw PowerShell commands only but also captures the contextual information and relationships between commands within the broader context of adversarial scenarios.

- *Empire* [45]: a post-exploitation and adversary emulation framework integrated with MITRE ATT&CK, provides PowerShell commands representative of advanced malicious techniques, further enriching the dataset with nuanced and intricate scenarios.

- *Online sources*: we manually verified and selected additional offensive samples from several security-related online sources. We gathered samples from *HackTricks* [46], *Red Team Recipe* [47], and *Infosec Matter* [48], community-driven cybersecurity wikis about ethical hacking, penetration testing, and information security. By including diverse examples specific to the offensive PowerShell dataset, the model acquires a more profound understanding of the conventions and best practices unique to PowerShell security commands.

We manually curated the dataset to cover the highest number of tactics in the MITRE ATT&CK framework. In particular, the dataset covers 12 out of 14 tactics from the MITRE ATT&CK framework, the *de facto* standard for adversarial techniques representation, with varying numbers of techniques and sub-techniques per tactic. Figure 2 illustrates the number of entries for each ATT&CK tactic. Each entry in the dataset is annotated with an NL description extracted from the respective source. We manually annotated every sample that did not come with a predefined description. Moreover, we enriched all those descriptions that did not provide enough information about the specific PowerShell command. For instance, in the case of Atomic Red Team, the PowerShell commands represent implementations of the techniques in the ATT&CK framework. Consequently, these commands are

[3]The ATT&CK framework is a comprehensive knowledge base of the tactics, techniques, and procedures (TTPs) that adversaries leverage during cyberattacks, developed by MITRE.

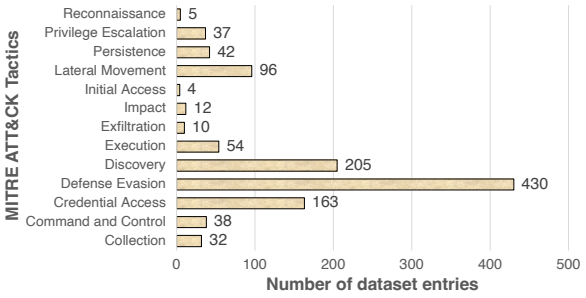

Figure 2: Mapping of fine-tuning dataset samples on the MITRE ATT&CK tactics.

often labeled with the technique name, which provides informative content about the technique itself rather than what the command does. To better understand how programmers and security experts describe PowerShell scripts and how to deal with ambiguities in natural language, we referred to popular books and manuals [49–51].

Finally, we notice that the size of our dataset is in line with other state-of-the-art corpora used to fine-tune ML models. In fact, in state-of-the-art code generation, the datasets for fine-tuning are relatively limited, in the order of one thousand samples [52].

3.3 Code Generation Task

To ensure the robustness of our study, we adopt the following state-of-the-art NMT models:

- **CodeT5+** [15] is a new family of Transformer models pre-trained with a diverse set of pretraining tasks to learn rich representations from both unimodal code data and bimodal code-text data. We utilize the variant with model size 220*M*, trained from scratch following T5's architecture [53]. It has an encoder-decoder architecture with 12 decoder layers, each with 12 attention heads and hidden layer dimension of 768, and 512 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 16, and beam size = 10.

- **CodeGPT** [16], a Transformer-based language model pre-trained on millions of Python functions and Java methods. The model architecture consists of 12 layers of Transformer decoders. We followed previous work for the implementation [54].

- **CodeGen** [17], an autoregressive language model for program synthesis with an architecture that follows a standard transformer decoder with left-to-right causal masking. The family of CodeGen models is trained in various sizes, including 350M, 2.7B, 6.1B, and 16.1B, and utilizes various datasets. Specifically, we leverage CodeGen-Multi, initialized from CodeGen-NL and further pre-trained on BigQuery [17], a large-scale dataset of multiple programming languages from GitHub repositories, which consists of 119.2B tokens and includes C, C++, Go, Java, JavaScript, and Python.

In our experiments, we randomly split the fine-tuning dataset into training (the set of examples used to fit the parameters), validation (the set used to tune the hyperparameters of the models), and test (the set used for the evaluation of the models) sets using a typical 80%/10%/10% ratio.

To assess the performance of the models in generating offensive PowerShell code from NL descriptions, we used *output similarity metrics*, which compare the generated code with the code from the ground truth. This type of metrics is widely used to assess the performance of AI generators in many code generation tasks [55], including the generation of code for security contexts [28–31, 56]. The metrics are:

- **Bilingual Evaluation Understudy (BLEU) score** [57]. It measures the degree of n-gram overlapping between the string of each code snippet produced by the model and the reference, for values of n usually ranging between 1 and 4 [58, 59]. We implemented BLEU-4 score (i.e., with $n = 4$) computation employing the `bleu_score` module contained in the open-source Natural Language Toolkit (NLTK) Python suite [60].

- **Edit Distance (ED)**. It measures the *edit distance* between two strings, i.e., the minimum number of operations on single characters required to make each code snippet produced by the model equal to the reference. For the edit distance, we adopted the Python library `pylcs` [61].

- **METEOR** [62]. It measures the *alignment* between each code snippet produced by the model and the reference. The alignment is defined as a mapping between unigrams (i.e., 1-gram), such that every unigram in each string maps to zero or one unigram in the other string and no unigrams in the same string. To calculate the METEOR metric, we relied on the Python library `evaluate` by HuggingFace [63].

- **ROUGE-L**. It is a metric based on the longest common subsequence (LCS) between the model output and the reference, i.e., the longest sequence of words (not necessarily consecutive, but still in order) shared between both. We computed the ROUGE-L metric using the Python package `rouge` [64].

All metrics range between 0 and 1, with higher scores corresponding to a better quality of the generated code. To evaluate the generated PowerShell code, we also introduce additional evaluation metrics based on static and dynamic analysis that are specific to our context. These metrics will be introduced in the following sections.

3.4 Research Questions

We designed this research study to answer the following research questions (RQs):

▷ **RQ1:** *To what extent can NMT models effectively generate offensive PowerShell code for security applications from NL descriptions?*

RQ1 aims to establish a preliminary assessment of NMT models in generating PowerShell code for offensive security applications. This investigation seeks to shed light on the models' efficacy in translating NL descriptions into offensive code.

▷ **RQ2:** *What is the influence of the training strategies on NMT models' performance in offensive PowerShell code generation?*

RQ2 focuses on the impact of pre-training and fine-tuning on the quality of generated code. We analyze the influence of these training strategies by considering different configurations of the NMT models and their impact on their performance.

▷ **RQ3:** *How good is the generated code in terms of code quality and dynamic behavior?*

RQ3 aims to evaluate the generated PowerShell code in a deeper way than output similarity metrics, in terms of syntactic correctness and capability of executing malicious actions realistically, through behavioral comparison with the ground truth.

▷ **RQ4:** *How do fine-tuned NMT models, leveraging security-oriented training data, compared to a publicly available, closed-source model?*

RQ4 introduces a comparative analysis, evaluating the performance of the fine-tuned models against a publicly available general-purpose language model, specifically ChatGPT 3.5. This investigation strives to evaluate whether specialization on security-focused data provides an advantage in the offensive PowerShell code generation domain.

4 Experimental Results

This section presents an extensive evaluation of NMT models (CodeT5+, CodeGPT, and CodeGen) on the generation of offensive PowerShell code. First, we assess the models' capability of generating PowerShell code in their original configuration (§ 4.1) without further training. Then, we evaluate the impact of different training strategies, i.e., domain-adaptive pre-training and fine-tuning, on the performance of such models (§ 4.2). To provide further insight into the PowerShell code generation, we analyze the quality of the generated code in terms of syntactic correctness (§ 4.3) and dynamic behavior (§ 4.4), i.e., its ability to replicate the behavior of the ground truth code. Finally, we compare the fine-tuned models with a public AI model (ChatGPT) for all the previous analyses (§ 4.5) to benchmark their performance against a publicly available, closed-source model.

Model	Pre-training	BLEU-4 (%)	ED (%)	METEOR (%)	ROUGE-L (%)
CodeT5+	✗	0.04	8.87	4.69	1.08
	✓	0.01	6.96	1.86	2.68
CodeGPT	✗	0.23	12.31	4.08	1.19
	✓	0.28	15.67	2.55	3.41
CodeGen	✗	0.06	7.58	2.88	0.21
	✓	0.00	0.43	0.09	0.00

Table 3: Performance of models with and without pre-training on zero-shot.

4.1 Zero-shot Learning

To establish a baseline for the evaluation, we initially used the NMT models in their original configuration, asking them to generate PowerShell code. This is a *zero-shot learning task*, where an NMT model is applied for a different scenario than the one for which it was trained. In this way, we evaluate the current gap of existing models in generating PowerShell code. Table 3 shows the results of this analysis. In this task, the models are tested without any gradient updates, relying only on the intent provided by the test set for inference [36,37]. The non-pre-trained versions of the models tend to generate Python code, but their performance is generally low for the downstream task of generating offensive PowerShell code. Pre-training the models with general-purpose PowerShell code slightly improves the accuracy but is still not high. Among the pre-trained versions, CodeGPT is the only one that provides output close to valid PowerShell code, although it does not align well with the expected code indicated by the intent in natural language. In summary, regardless of pre-training, all models demonstrate the need for fine-tuning on a tailored dataset for optimal performance in generating offensive PowerShell code.

4.2 Impact of Training Strategies

The evaluation of CodeT5+, CodeGPT, and CodeGen involved a meticulously designed test plan. More precisely, the models underwent three distinct fine-tuning scenarios: 3 Epochs, 10 Epochs, and 30 Epochs. This deliberate choice allowed us to assess the impact of prolonged fine-tuning on the models' ability to generate PowerShell code for offensive security tasks. In each scenario, we considered two training configurations: one with pre-training and the other without. This test plan allowed us to systematically explore the models' capabilities under varying conditions, providing a comprehensive understanding of their strengths and limitations. Table 4 shows the results.

In the 3 epochs setting, CodeT5+ exhibits low performance, regardless of pre-training, with a BLEU-4 score lower than 10%. In contrast, CodeGPT and CodeGen demonstrate notable performance even after a short fine-tuning period,

Model	Epochs	Pre-train. (%)	BLEU-4 (%)	ED (%)	METEOR (%)	ROUGE-L (%)
CodeT5+	3	✗	4.22	35.11	28.83	22.26
		✓	4.57	35.96	30.57	23.99
	10	✗	12.64	46.72	44.76	37.65
		✓	11.88	49.10	46.11	37.17
	30	✗	17.40	**50.92**	47.61	**39.05**
		✓	18.50	50.23	**47.87**	38.86
CodeGPT	3	✗	10.28	40.71	31.21	25.60
		✓	12.80	42.54	35.14	30.35
	10	✗	16.22	46.39	40.50	33.52
		✓	17.93	49.88	45.12	37.12
	30	✗	**21.71**	50.17	45.34	38.63
		✓	19.94	49.20	45.45	38.06
CodeGen	3	✗	16.20	47.68	42.27	35.97
		✓	14.75	45.88	39.86	34.69
	10	✗	19.15	50.52	46.76	37.63
		✓	19.04	48.45	43.25	35.25
	30	✗	18.23	47.53	44.10	35.48
		✓	18.53	48.67	44.14	35.45

Table 4: Performance of models with and without pre-training and different number of epochs. Best results for each metric are **blue/bold**.

achieving a BLEU-4 score higher than 10% and an ED over 40%. Notably, after 3 epochs, CodeGen demonstrates superior performance compared to the other two models. In the 10 epochs experiment, CodeT5+ shows significant improvement, with BLEU-4 tripling to 12%. Moreover, ED, METEOR, and ROUGE-L experience a rise of 12-16%. CodeGPT also enhances its performance, surpassing CodeT5+ in terms of BLEU-4 score, although it faces challenges in achieving the same level of overall improvement. CodeGen remains ahead of the other models, even reaching an ED over 50%. For a more in-depth assessment of the models' adaptability, the training duration is extended to 30 epochs. CodeT5+ demonstrates superior performance over CodeGPT in ED, METEOR, and ROUGE-L metrics, while CodeGPT exhibits a higher BLEU-4 score surpassing 20%. Notably, both models achieve a high ED value of around 50%. CodeGen establishes its performance without further improvement compared to the 10 epochs versions.

To provide an estimate of the goodness of the results, we compared the results of the models with the performance of the state-of-the-art (SOTA). Since the task of generating PowerShell using NMT models is a task never addressed before, we compared the results with recent work investigating the effectiveness of existing models in the generation of different languages from NL, specifically, Python code [65] and in shell language [66]. We found that the best performance is 21% for BLEU-4 and 38% for METEOR in the case of the Python language, and 25% for BLEU-4 and 44% for ED in the case of shell language. We notice that our results are in line with the ones of the SOTA. Even better, our best performance, represented by CodeT5+ without pre-training and 30 fine-tuning epochs, overcomes the SOTA over all the metrics.

We also assessed the impact of varying the number of epochs on fine-tuning time, with distinct differences observed between 3, 10, and 30 epochs for each model. For both CodeT5+ and CodeGPT, fine-tuning over 3 epochs takes approximately 20 minutes, whereas CodeGen requires double that time (40 minutes). Extending to 10 epochs, CodeT5+ and CodeGPT need around 35 and 39 minutes, respectively, while CodeGen's training time increases to 90 minutes. For the 30-epoch extension, CodeT5+ takes about 80 minutes, CodeGPT requires 110 minutes, and CodeGen extends its training time to 270 minutes. Finally, the comparison between the fine-tuning times of pre-trained and non-pre-trained models did not reveal evident differences, suggesting that the pre-training process does not introduce a significant computational overhead during the subsequent fine-tuning phase.

RQ1: To what extent can state-of-the-art NMT models effectively generate offensive PowerShell code for security applications from NL descriptions?

The evaluation of CodeT5+, CodeGPT, and CodeGen underscores their remarkable effectiveness in generating offensive PowerShell code for security applications from NL descriptions. CodeGen surpasses the other models in the 3 and 10 epochs settings according to all metrics. CodeT5+, designed with a specialized architecture for code generation tasks, consistently outperforms CodeGPT and CodeGen across various metrics in the 30 epochs setting. Particularly noteworthy is the comparison with SOTA performance in code generation tasks for different languages, such as Python and shell languages. Our best-performing model, CodeT5+ without pre-training and 30 fine-tuning epochs, surpasses the SOTA results, exhibiting superior performance across all metrics.

Considering the impact of pre-training further enriched our evaluation. Focusing on the 3-epoch experiments, CodeT5+ exhibits a slight improvement across all metrics, and CodeGPT extends the improvement to 2%-4% across all metrics. Conversely, CodeGen appears to have better performance without pre-training. Training the models for 10 epochs reveals a more pronounced distinction between the two versions. CodeT5+ pre-training results in a 2% increase in both Edit Distance (ED) and METEOR metrics. CodeGPT, on the other hand, shows a substantial displacement of 1.7%, 3.5%, 4.6%, and 3.6% for BLEU-4, ED, METEOR, and ROUGE-L, respectively. CodeGen maintains a negative displacement between the versions even with the extended training duration. When extending the fine-tuning duration to 30 epochs, pre-training

Natural Language Intent	Ground Truth	Model Output
Start a process with DirLister, wait for 4 seconds, and stop the DirLister process.	`Start-Process ${WebBrowserPassViewPath} ; Start-Sleep -Second 4 ; Stop-Process -Name "WebBrowserPassView"`	`Start-Process ${WebBrowserPassViewPath} ; Start-Sleep -Second 4 ; Stop-Process -Name "WebBrowserPassView"`
Invoke-Mimikatz cmdlet with bypassed execution policy.	`powershell.exe -exec bypass -c " Invoke-Mimikatz "`	`powershell.exe -ExecutionPolicy Bypass -Command " Invoke-Mimikatz "`
Invoke compiled HTML Help (CHM) file with specific storage handler, HH file path, and CHM file path.	`Invoke-ATHCompiledHelp -InfoTechStorageHandler $ { infotech_storage_handler } -HHFilePath $ { hh_file_path } -CHMFilePath $ { chm_file_path }`	`Invoke-ATHCompiled`**`HTML`**`Help` ~~**`-InfoTechStorageHandler`**~~ **`$ { infotech_storage_handler }`** ~~`-HHFilePath $ { hh_file_path }`~~ `-CHMFilePath $ { chm_file_path }`
Get the wininit process and perform token manipulation to create a new process for 'cmd.exe'.	`Get-Process wininit | Invoke-TokenManipulation -CreateProcess 'cmd.exe'`	**`$wininit =`** `Get-Process wininit | Invoke-TokenManipulation -CreateProcess 'cmd.exe'`

Table 5: Illustrative examples of model output. The prediction errors are **red/bold**. ~~Slashed~~ text refers to omitted predictions.

did not consistently yield superior results. In this case, the performance of pre-trained models is comparable to non-pre-trained counterparts.

RQ2: What is the influence of the training strategies on NMT models' performance in offensive PowerShell code generation?

As the fine-tuning period extends, such as with 10 and 30 epochs, the benefits of pre-training diminish or even become counterproductive. In these cases, the performance of pre-trained models consistently falls below that of their non-pre-trained counterparts. This highlights the variable effectiveness of pre-training, dependent on the duration of fine-tuning. These findings underscore the interplay between the duration of training epochs and the usage of pre-training, emphasizing the importance of carefully considering these factors in model development.

Table 5 illustrates four cases of model predictions. They are examples from our test sets to highlight both successful and failed prediction cases. Row # 1 demonstrates the models' ability to generate a PowerShell snippet composed of multiple commands (separated by semicolons) without errors. The model correctly predicts the correct variables, e.g., `WebBrowserPassViewPath`, and command names, such as `Start-Process`, `Start-Sleep`. Row # 2 is indicative of the concept of implicit model knowledge. Indeed, the model can generate a correct command by leveraging alternative equivalent versions of PowerShell's option flags (e.g., `-ExecutionPolicy` instead of `-exec`). Row # 3 shows a relevant example of a failure case. It is possible to notice how the model correctly predicts the variable names and values except for one not referenced in the intent (`-InfoTechStorageHandler`). In addition, the model fails to predict the correct command name, generating an additional word (`HTML`) based on the NL description. Finally, row # 4 illustrates another incorrect example in which the model is capable of generating the ground truth code, except for introducing an additional variable to save the output of the command (`$wininit =`).

Overall, we can conclude that these examples indicate the model's ability to generate complex PowerShell snippets, even though there is still some error margin, specifically related to omissions (e.g., variable names).

4.3 Static Analysis

We evaluated the generated code through *static analysis* to ensure that the code adheres to PowerShell conventions and does not contain syntax errors. The analysis was conducted on the top-performing models identified in the previous evaluation, namely the 30-epoch versions of CodeT5+ with pre-training, CodeGPT without pre-training, and CodeGen with pre-training. The static analysis leverages *PSScriptAnalyzer* [67], a static code checker for PowerShell modules and scripts. The primary purpose of PSScriptAnalyzer is to assess the quality of PowerShell code by analyzing its syntax, structure, and adherence to best practices. The rules are based on PowerShell best practices identified by the PowerShell Team and the community, organized into categories such as Cmdlet Design, Script Functions, Error Handling, Scripting Style, and Script Security. The severity levels (ParseError, Error, Warning, Information) associated with each rule indicate the importance and impact of adhering to the specific guideline. In this work, we focused on *parse errors*, which occur during the parsing phase of a program's execution, *errors*, occurring when code

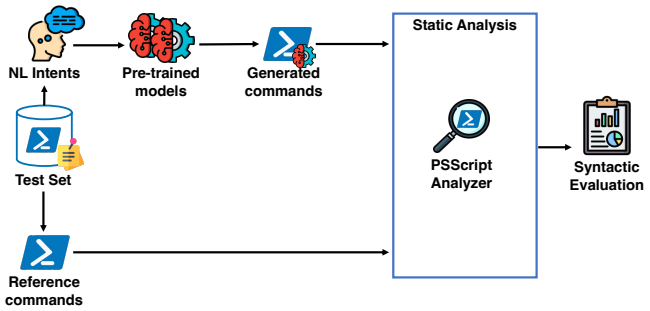

Figure 3: Static analysis workflow.

Model	Single Accuracy (%)	Comparative Accuracy (%)
CodeT5+	91.15	92.04
CodeGPT	98.23	98.23
CodeGen	98.23	98.23

Table 6: Syntactic evaluation for the best models.

does not meet specific high-severity rules (e.g., hardcoding computer names, using plain text passwords), and *warnings*, which typically highlight potential issues or coding practices that might lead to errors or security concerns.

We developed a syntactic analysis tool to streamline the process of detecting *parse errors*, *errors*, and *warnings* in PowerShell scripts. This tool automatically feeds PSScriptAnalyzer with PowerShell commands generated by the models during the testing phase. By doing so, our tool identifies errors and warnings in the generated code, assessing the overall syntactic quality of the models.

The syntactic analysis process begins with our test set, which consists of NL intents paired with reference PowerShell commands. These NL intents are fed into fine-tuned models to produce the PowerShell code. Both the generated commands and their corresponding references are then subjected to the syntax analyzer.

To assess the syntactic quality of the generated commands, we introduce two distinct metrics: *Single Syntax Accuracy* and *Comparative Syntax Accuracy*. The metrics are defined as follows:

- **Single Syntax Accuracy**: evaluates the percentage of commands without parse errors. This evaluation is independent of the reference commands from the ground truth.
- **Comparative Syntax Accuracy**: assesses the syntactic correctness of the generated commands by considering the results alongside the reference commands. When both commands present common parse errors, these are excluded from the counting process. Given that some reference commands include stub templates such as `<code>` or `<command>`, the analysis filters out parse errors associated with these templates, specifically the *RedirectionNotSupported* and *MissingFileSpecification* errors.

Test Set	ParseError (%)	Error (%)	Warning (%)
CodeT5+	8.85	1.94	35.92
CodeGPT	1.77	2.70	29.73
CodeGen	1.77	1.80	31.53
Ground Truth	2.65	0.00	39.09

Table 7: Summary of ParseError, Error, and Warning percentages for models and ground truth on the test set.

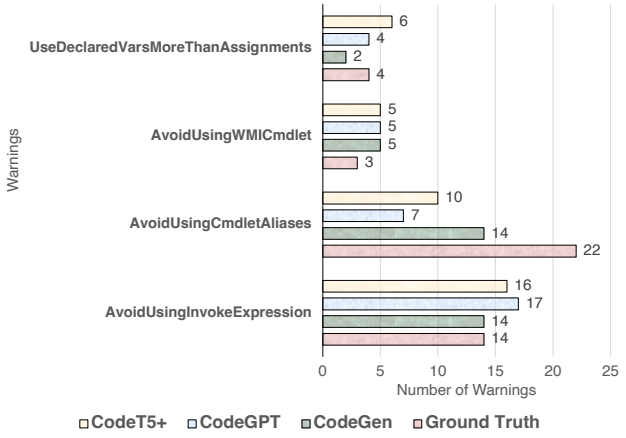

Figure 4: Counts for different warning types in each test set.

The workflow for the syntactic analysis is depicted in Figure 3. Looking at the results in Table 6, it is possible to notice that all the models achieved a score greater than 90%, assessing their strong capability to generate syntactically correct code. CodeGPT and CodeGen, in general, demonstrate high performance across both syntax metrics. Table 7 summarizes the percentages for various severity types in the test set. Given that warning frequencies are consistently above 30% for all models, including the ground truth, Figure 4 enumerates the various warning types within each set.

4.4 Execution Analysis

The execution analysis aims to evaluate the generated offensive PowerShell code when running in an actual system. This involves assessing the ability of the code to behave as intended in terms of effects caused on the system. Therefore, we run both code from the ground truth and generated code, monitor their behavior at runtime, and compare the behavioral events

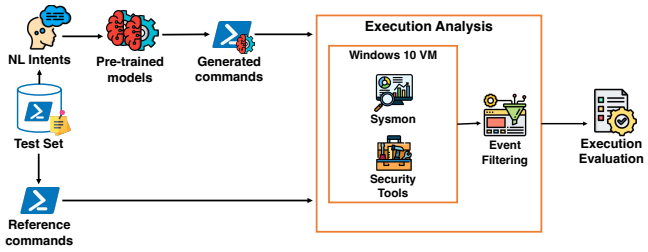

Figure 5: Execution analysis workflow.

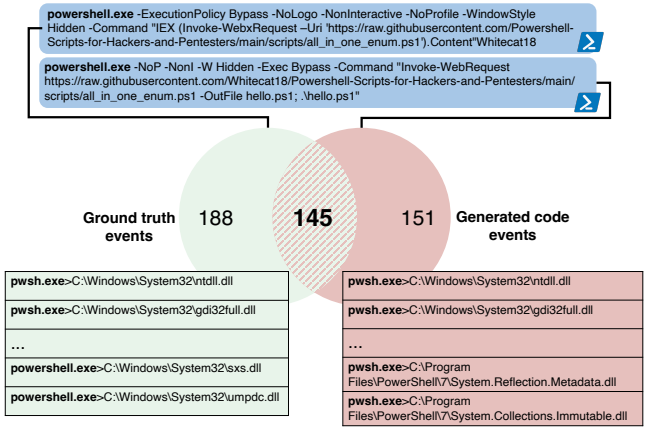

Figure 6: Comparison between events.

that occurred during their execution. The entire workflow for the execution analysis is shown in Figure 5.

We performed the experiments in a controlled and dedicated testing environment. The controlled environment consists of a virtualized Windows 10 system running in VirtualBox 7. The system is equipped with a set of security-related tools, such as PowerSploit [68] and Mimikatz [69], that are invoked by many samples of offensive code in our dataset. We assume that these tools have been previously infiltrated by the attacker in a previous stage, as typical of advanced malicious campaigns. To monitor the execution of Powershell code, we integrated Sysmon [70], a popular Windows service for gathering system events, including the filesystem, the network, and the Windows Registry. To be able to run the generated code on the system, we assume the scenario in which an attacker already bypassed part of the security mechanisms by deactivating the Microsoft Defender Firewall, Windows Defender, and Microsoft Defender SmartScreen.

The evaluation involved executing each command from both the generated ones and those from ground truth multiple times as a single-line PowerShell script. This generates a process through the standard Windows *System.Diagnostics.Process*. We filter the events recorded by Sysmon by filtering out records related to previous irrelevant

Model	Precision (%)	Recall (%)	F1-Score (%)
CodeT5+	97.26	80.94	88.35
CodeGPT	91.86	85.23	88.42
CodeGen	96.94	80.97	88.24

Table 8: Execution analysis results.

events and selecting records based on the Process ID (PID), focusing on both the parent process responsible for executing the PowerShell command and its child processes. The comparison has been performed comparing the events triggered by the generated command (called *retrieved records*) to those from the execution profile of the ground truth (called *relevant records*). The events that appear both when executing the generated code and the ground truth are *relevant records retrieved*. From these sets of events, we evaluate the *precision*, *recall* and *F1-score* of the generated code, defined as follows:

$$\text{precision} = \frac{1}{N}\sum_{i}^{N}\frac{\#(\text{relevant records retrieved})_i}{\#(\text{retrieved records})_i}$$

$$\text{recall} = \frac{1}{N}\sum_{i}^{N}\frac{\#(\text{relevant records retrieved})_i}{\#(\text{relevant records})_i}$$

$$\text{F1-Score} = 2\,\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Figure 6 illustrates an example of event analysis: given the ground truth and the generated PowerShell command, we execute them and compare the set of events triggered by each command to measure their overlap. To avoid noise in the analysis due to events that only occur sporadically (e.g., because of non-determinism sources in the system), we identify such events by performing multiple repeated runs of the code and discard non-reproducible events from the analysis. After every command execution, the Windows environment is restored to a clean state, by reloading the virtual machine from a snapshot, to avoid interferences caused by the effect of previous commands.

The results shown in Table 8 outline how all models share an overall precision higher than 90% and an overall recall higher than 80%, likewise, the Execution F1-Score is very similar between the different models and higher than 88%. Thus, although there were differences found in the textual similarity analysis, the generated code closely matches the ground truth in terms of dynamic events.

RQ3: How good is the generated code in terms of code quality and dynamic behavior?

The syntactic analysis of the generated code showed that the models are indeed capable of generating high-quality

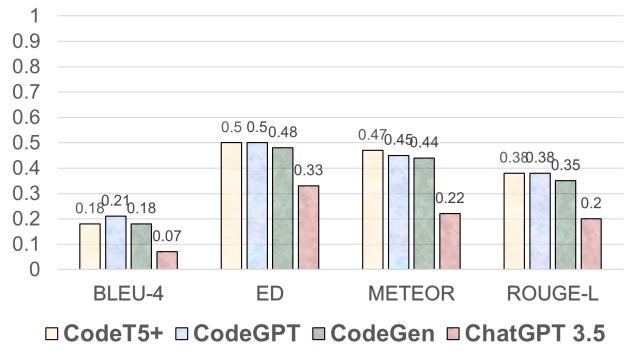

Figure 7: Comparison with ChatGPT on output similarity metrics.

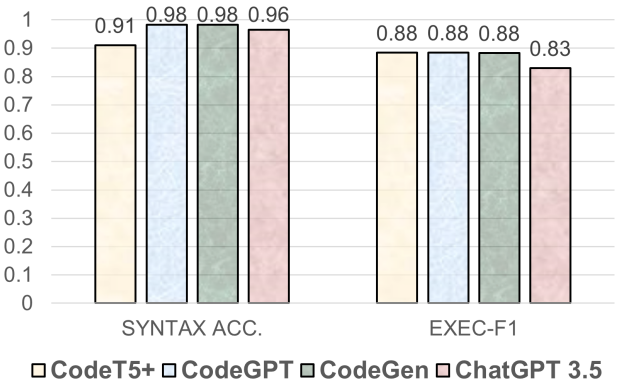

Figure 8: Comparison with ChatGPT on static and execution analysis evaluation metrics.

PowerShell code. CodeGPT and CodeGen achieve the best results in terms of Single and Comparative Accuracy, along with an amount of Warnings and ParseErrors comparable to the ground truth. The execution analysis revealed that the generated PowerShell code closely replicates the behavior of the ground truth code, generating the same events in the target system. This is indicative of the generated code's capability of performing the malicious actions described in the NL intents.

4.5 Comparison with Public AI Model

In this study, we conducted a comprehensive evaluation by comparing the performance of our fine-tuned models, CodeT5+, CodeGPT, and CodeGen, with ChatGPT, the OpenAI LLM service widely used for a variety of tasks, including code generation [71]. The purpose was to assess the specialized capabilities of our models in generating PowerShell code for offensive security tasks and to benchmark their performance against a publicly available, closed-source model. We leveraged ChatGPT 3.5, which represents the most recent free version at the time of this work.

To assess the capabilities of the OpenAI model, we first provided a detailed description of the required task, i.e., the generation of PowerShell commands starting from NL descriptions, including an example of input and the desired output. Then, we provided a list of natural language code descriptions and asked ChatGPT to automatically generate the corresponding PowerShell code. Specifically, following works and guidelines on prompt engineering [71, 72], we leveraged the following prompt: `I want you to act as a code generator. Given a natural language description of a PowerShell command, generate the corresponding PowerShell code.`

Figure 7 shows the results of this analysis. The figure shows that our fine-tuned models consistently outperform ChatGPT across multiple evaluation metrics. Specifically, ChatGPT exhibits a BLEU-4 score of 7.45%, an ED of 33.84%, a METEOR of 22.14%, and a ROUGE-L of 20.61%. In contrast, our fine-tuned models showcase superior overall performance across all output similarity metrics. The tailored training on the specialized fine-tuning dataset, designed specifically for offensive security code generation, results in more accurate code generation, enabling our models to surpass the capabilities of ChatGPT in this particular task. We also analyzed the syntactical quality of the PowerShell code generated by ChatGPT, obtaining a Syntax Single Accuracy of 95.58% and a Syntax Comparative Accuracy of 96.46%. These results underscore the commendable ability of ChatGPT to generate accurate and syntactically correct PowerShell code.

Finally, we extended the execution analysis to ChatGPT, following the same strategies described in Section 4.4, obtaining an overall Execution F1-Score of 82.92%. Despite the strong syntactic performance, ChatGPT remains one step below the fine-tuned models in the qualitative analysis of the generated PowerShell code. The results of this analysis are shown in Figure 8.

RQ4: How do fine-tuned NMT models, leveraging security-oriented training data, compare to a publicly available, closed-source model?

The comparative analysis with ChatGPT, a publicly available general-purpose language model, highlights the specialized strengths of privately fine-tuned models, CodeT5+, CodeGPT, and CodeGen, in offensive PowerShell code generation. The fine-tuned models consistently outperform ChatGPT across BLEU-4, Edit Distance, and METEOR scores. While showing notable performance on syntactic accuracy, ChatGPT achieves poorer results than the fine-tuned models for the execution analysis. This underscores the significance of

domain-specific fine-tuning and the benefits of training on security-oriented datasets, providing an advantage in generating offensive PowerShell code compared to a general-purpose language model. The results affirm the effectiveness of tailored training data for achieving superior performance in domain-specific tasks.

5 Threats To Validity

Model selection. The external validity of the study might be impacted by the choice of NMT models (CodeT5+, CodeGPT, CodeGen). To mitigate this, we carefully selected models with distinct architectures and capabilities, ensuring a representation of current advancements in the field [16,73,74]. This careful selection aims to ensure that our findings reflect broader trends in NMT model performance for code generation tasks.

Evaluation metrics. The reliance on output similarity metrics, although representing the most common solution in the field, poses a potential threat to construct validity, as these metrics may not fully encapsulate the correctness and functional adequacy of the generated PowerShell commands. To address this issue, our evaluation strategy encompasses a comprehensive suite of metrics, including similarity, syntactic, and execution metrics, each offering unique insights into the models' performance. By considering multiple variants of these metrics and aligning with common practices in code generation evaluation, we aim to provide a well-rounded assessment. No single metric is perfect, but analyzing them collectively allows for a more comprehensive evaluation of the code.

Fine-tuning data. The construction of our dataset, meticulously curated from several sources such as online repositories, Atomic Red Team, Stockpile, and Empire, introduces potential limitations regarding the generalizability of our models' performance across different offensive security contexts. To minimize the impact of these limitations, we sourced data from diverse origins and conducted manual verification of each sample in the labeled dataset, ensuring the completeness and coherence of descriptions with the intended programs. The diversity in data sources and the thorough verification process aim to diminish the influence of any singular source's peculiarities and errors in programs or descriptions, thereby enhancing the dataset's applicability and reliability for training and evaluating AI models in generating offensive PowerShell code. Furthermore, our approach to crafting NL descriptions, inspired by established styles found in PowerShell literature, mirrors real-world scenarios where such descriptions play a critical role in describing PowerShell commands. Finally, regarding the size of our dataset, we notice that it is in line with other state-of-the-art corpora used to fine-tune models, which are in the order of one thousand samples [52].

6 Ethical Considerations

Recognizing that attackers use attacks as a weapon, it is important to specify that the goal of the proof-of-concept (POC) is not to cause harm but to surface security weaknesses within the software. Identifying security issues allows companies to patch vulnerabilities and protect themselves against attacks.

Offensive security is a sub-field of security research that tests security measures from an adversary or competitor's perspective, employing ethical hackers to probe a system for vulnerabilities [75, 76]. Our work aims to automate attack generation to explore critical vulnerabilities before they are exploited by attackers [77]. Indeed, our work simplifies the process of coding the attacks to surface security weaknesses within the software and can provide valuable information about the technical skills, degree of experience, and intent of the attackers. With this information, it is possible to implement measures to detect and prevent attacks [78].

7 Conclusion

In this paper, we assessed the feasibility of using NMT models to generate PowerShell code for security contexts. We aimed to demonstrate that AI-based code generators are indeed fit to generate PowerShell code, specifically, offensive PowerShell, which spans several applications in the cybersecurity domain. The evaluation of CodeT5+, CodeGPT, and CodeGen demonstrated that these models achieve significant performance on the code generation task, both with and without pre-training. Moreover, the study showed that domain-specific fine-tuning allows our models to outperform state-of-the-art privately fine-tuned models, i.e., ChatGPT. We also introduced two novel datasets for PowerShell code generation to use for pre-training and fine-tuning AI-code generators.

Future work includes further analysis of the generated code, such as sandbox execution of the offensive scripts, to understand whether the code can evade detection measures, along with more NMT models spanning several architectures and capabilities.

Acknowledgments

This work has been partially supported by MUR PRIN 2022, project FLEGREA, CUP E53D23007950001 (https://flegrea.github.io) and by an Industrial Ph.D. grant (PNRR - DM 117/2023) from MUR and DigitalPlatforms S.p.A, CUP E66E23000580003.

References

[1] A. Applebaum, D. Miller, B. E. Strom, C. Korban, and R. Wolf, "Intelligent, automated red team emulation,"

Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016.

[2] A. B. Ajmal, M. A. Shah, C. Maple, M. N. Asghar, and S. U. Islam, "Offensive security: Towards proactive threat hunting via adversary emulation," *IEEE Access*, vol. 9, pp. 126 023–126 033, 2021.

[3] E. M. Hutchins, M. J. Cloppert, R. M. Amin *et al.*, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, no. 1, p. 80, 2011.

[4] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, "Mitre att&ck: Design and philosophy," in *Technical report*. The MITRE Corporation, 2018.

[5] Sudhakar and S. Kumar, "An emerging threat fileless malware: a survey and research challenges," *Cybersecurity*, vol. 3, no. 1, p. 1, 2020.

[6] I. Kara, "Fileless malware threats: Recent advances, analysis approach through memory forensics and research challenges," *Expert Systems with Applications*, vol. 214, p. 119133, 2023.

[7] Varonis, "What is Fileless Malware? PowerShell Exploited," https://www.varonis.com/blog/fileless-malware.

[8] Cybersecurity & Infrastructure Security Agency, "Identifying and Mitigating Living Off the Land Techniques," https://www.cisa.gov/sites/default/files/2024-02/Joint-Guidance-Identifying-and-Mitigating-LOTL_V3508c.pdf.

[9] R. Natella, P. Liguori, C. Improta, B. Cukic, and D. Cotroneo, "Ai code generators for security: Friend or foe?" *IEEE Security & Privacy*, 2024.

[10] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv: 2107.03374*, 2021.

[12] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[13] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation," *arXiv preprint arXiv: 2308.01861*, 2023.

[14] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A dataset for automatic shellcode generation," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, R. Lachmy, Z. Yao, G. Durrett, M. Gligoric, J. J. Li, R. Mooney, G. Neubig, Y. Su, H. Sun, and R. Tsarfaty, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 58–64. [Online]. Available: https://aclanthology.org/2021.nlp4prog-1.7

[15] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[16] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[17] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023.

[18] OpenAI, "ChatGPT," https://openai.com/chatgpt.

[19] Z. Li, Q. A. Chen, C. Xiong, Y. Chen, T. Zhu, and H. Yang, "Effective and light-weight deobfuscation and

semantic-aware attack detection for powershell scripts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1831–1847.

[20] M.-H. Tsai, C.-C. Lin, Z.-G. He, W.-C. Yang, and C.-L. Lei, "Powerdp: De-obfuscating and profiling malicious powershell commands with multi-label classifiers," *IEEE Access*, vol. 11, pp. 256–270, 2023.

[21] D. Hendler, S. Kels, and A. Rubin, "Detecting malicious powershell commands using deep neural networks," in *Proceedings of the 2018 on Asia conference on computer and communications security*, 2018, pp. 187–197.

[22] A. Rubin, S. Kels, and D. Hendler, "Amsi-based detection of malicious powershell code using contextual embeddings," *arXiv preprint arXiv:1905.09538*, 2019.

[23] M. Mimura and Y. Tajiri, "Static detection of malicious powershell based on word embeddings," *Internet of Things*, vol. 15, p. 100404, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660521000482

[24] Y. Mezawa and M. Mimura, "Evaluating the possibility of evasion attacks to machine learning-based models for malicious powershell detection," in *International Conference on Information Security Practice and Experience*. Springer, 2022, pp. 252–267.

[25] G. Rusak, A. Al-Dujaili, and U.-M. O'Reilly, "Ast-based deep learning for detecting malicious powershell," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2276–2278.

[26] M. Gupta, C. Akiri, K. Aryal, E. Parker, and L. Praharaj, "From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy," *IEEE Access*, vol. 11, pp. 80 218–80 245, 2023.

[27] P. Charan, H. Chunduri, P. M. Anand, and S. K. Shukla, "From text to mitre techniques: Exploring the malicious use of large language models for generating cyber attack payloads," *arXiv preprint arXiv:2305.15336*, 2023.

[28] 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.

[29] P. Liguori, E. Al-Hossami, V. Orbinato, R. Natella, S. Shaikh, D. Cotroneo, and B. Cukic, "Evil: exploiting software via natural language," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 321–332.

[30] G. Yang, X. Chen, Y. Zhou, and C. Yu, "Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning," in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 361–372.

[31] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Han, and T. Chen, "Exploitgen: Template-augmented exploit code generation based on codebert," *Journal of Systems and Software*, vol. 197, p. 111577, 2023.

[32] A. M. Dai and Q. V. Le, "Semi-supervised sequence learning," *Advances in neural information processing systems*, vol. 28, 2015.

[33] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," *arXiv preprint arXiv:2004.10964*, 2020.

[34] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *AI Open*, 2022.

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

[36] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[38] H. Wang, J. Li, H. Wu, E. Hovy, and Y. Sun, "Pre-trained language models and their applications," *Engineering*, 2022.

[39] R. Tufano, L. Pascarella, and G. Bavota, "Automating code-related tasks through transformers: The impact of pre-training," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2425–2437.

[40] 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.

[41] Red Canary, "Atomic Red Team," https://atomicredteam.io/.

[42] M. Corporation, "MITRE ATT&CK," https://attack.mitre.org/.

[43] MITRE, "CALDERA plugin: Stockpile," https://github.com/mitre/stockpile.

[44] MITRE, "CALDERA," https://github.com/mitre/caldera.

[45] Empire Project, "Empire," https://github.com/EmpireProject/Empire.

[46] Hacktricks, "Hacktricks," https://book.hacktricks.xyz/.

[47] R. T. Recipe, "PowerShell tips & tricks," https://redteamrecipe.com/powershell-tips-tricks/.

[48] I. Matter, "PowerShell commands for pentesters," https://www.infosecmatter.com/powershell-commands-for-pentesters/.

[49] Tutorial's Point, "Learn PowerShell," https://www.tutorialspoint.com/powershell/index.htm.

[50] T. Lee, K. Mitschke, M. E. Schill, and T. Tanasovski, *Windows PowerShell 2.0 Bible*. John Wiley & Sons, 2011.

[51] L. Holmes, *Windows PowerShell Cookbook: The Complete Guide to Scripting Microsoft's Command Shell*. O'Reilly Media, 2012.

[52] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu, S. Zhang, G. Ghosh, M. Lewis, L. Zettlemoyer, and O. Levy, "LIMA: less is more for alignment," *CoRR*, vol. abs/2305.11206, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.11206

[53] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, pp. 140:1–140:67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[54] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html

[55] P. Liguori, C. Improta, R. Natella, B. Cukic, and D. Cotroneo, "Who evaluates the evaluators? on automatic metrics for assessing ai-based offensive code generators," *Expert Systems with Applications*, vol. 225, p. 120073, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423005754

[56] X. Ruan, Y. Yu, W. Ma, and B. Cai, "Prompt learning for developing software exploits," in *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, 2023, pp. 154–164.

[57] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040/

[58] L. Han, "Machine translation evaluation resources and methods: A survey," *arXiv preprint arXiv:1605.04515*, 2016.

[59] D. Munkova, P. Hajek, M. Munk, and J. Skalka, "Evaluation of machine translation quality through the metrics of error rate and accuracy," *Procedia Computer Science*, vol. 171, pp. 1327–1336, 2020.

[60] NLTK, "Natural Language Toolkit (NLTK), bleu_score module," 2023. [Online]. Available: https://www.nltk.org/api/nltk.translate.bleu_score.html

[61] pylcs, "Python library pylcs," 2023. [Online]. Available: https://pypi.org/project/pylcs/

[62] A. Lavie and A. Agarwal, "Meteor: An automatic metric for mt evaluation with high levels of correlation with human judgments," in *Proceedings of the Second Workshop on Statistical Machine Translation*, ser. StatMT '07. USA: Association for Computational Linguistics, 2007, p. 228–231.

[63] evaluate, "Python library evaluate," 2022. [Online]. Available: https://pypi.org/project/evaluate/

[64] rouge, "Python ROUGE Score Implementation," 2021. [Online]. Available: https://pypi.org/project/rouge/

[65] J. Shin, M. Wei, J. Wang, L. Shi, and S. Wang, "The good, the bad, and the missing: Neural code generation for machine learning tasks," *arXiv preprint arXiv:2305.09082*, 2023.

[66] J. Shi, S. Jiang, B. Xu, J. Liang, Y. Xiao, and W. Wang, "Shellgpt: Generative pre-trained transformer model for shell language understanding," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 671–682.

[67] Microsoft, "PSScriptAnalyzer," https://github.com/PowerShell/PSScriptAnalyzer.

[68] Will Schroeder, "PowerSploit," https://github.com/PowerShellMafia/PowerSploit.

[69] Benjamin Delpy, "Mimikatz," https://github.com/gentilkiwi/mimikatz.

[70] Mark Russinovich,Thomas Garnier, "System Monitor," https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon.

[71] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023.

[72] Microsoft, "Prompt Engineering - Learn how to use AI models with prompt engineering," https://microsoft.github.io/prompt-engineering/.

[73] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[74] S. Tipirneni, M. Zhu, and C. K. Reddy, "Structcoder: Structure-aware transformer for code generation," *arXiv preprint arXiv:2206.05239*, 2022.

[75] S. Bratus, I. Arce, M. E. Locasto, and S. Zanero, "Why offensive security needs engineering textbooks," *Yale Law & Policy Review*, p. 2, 2013.

[76] J. G. Oakley, "The state of modern offensive security," in *Professional Red Teaming*. Springer, 2019, pp. 29–41.

[77] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *NDSS*, 2011.

[78] I. Arce, "The shellcode generation," *IEEE security & privacy*, vol. 2, no. 5, pp. 72–76, 2004.