AI-Generated PowerShell Malware: An Experimental Framework and Dataset

Luciano Pianese, Vittorio Orbinato, Pietro Liguori, Roberto Natella

***Abstract*—Generative AI has emerged as a significant cybersecurity threat, with several recent attack campaigns leveraging LLMs to generate code for malicious purposes via scripting languages such as PowerShell. Consequently, for cybersecurity analysts, it is imperative to investigate the offensive capabilities of AI code generators. In this paper, we propose an experimental framework to assess LLM-generated PowerShell malware, which comprises a novel sandbox approach for dynamic analysis of AI-generated malware. Furthermore, we present a novel, manually curated dataset of real-world PowerShell malware, annotated in natural language to assist the training and evaluation of LLMs. Finally, this study evaluates permissive, open-weight LLMs adapted to PowerShell malware generation. Our results reveal a high degree of similarity between real malware and LLM-generated ones in terms of triggered OS malicious events, with a median Jaccard index of 84.5% and 48.4% of instances achieving complete overlap.**

***Index Terms*—AI Code Generation; Cyber Threat Intelligence; Adversary Emulation; Large Language Models.**

I. Introduction

Generative Artificial Intelligence (AI) has become a key asset in the hands of malicious attackers. Industry leaders including Microsoft [1], OpenAI [2], and Anthropic [3] have been reporting attacks from Advanced Persistent Threats (APTs) that leverage Large Language Models (LLMs), including nation-state sponsored actors and cybercrime, to develop malicious code at a higher efficiency.

The practical utilization of LLMs is now manifest in extant malware samples found in the wild. Forensic analysis identified several such strains, such as PromptLock [4], an AI-powered malware that leverages an open-source LLM model locally to dynamically generate malicious scripts and execute them in real time, such as for encrypting data, exfiltrating, and destroying files. Further examples are LAMEHUG [5], attributed to APT28 [6], a malicious script that embeds prompts to dynamically craft Windows reconnaissance commands and execute them on-the-fly using one of the open-source models from the Qwen2.5-Coder collection [7]; PROMPTFLUX [8], which prompts an LLM to generate scripts in order to evade antivirus detection; and Mythic's Medusa [9], a command-and-control (C2) agent that generates OS-level commands for post-exploitation from prompts in natural language. Given their ability to perform a variety of actions without delivering executable files, *scripting languages* are the preminent instrument for pursuing malicious intents [10], including AI-driven APT campaigns. In particular, PowerShell is the most used language [11], [12], given its ability to perform a wide variety of malicious actions, and the ubiquity of the Windows operating system within strategic enterprise infrastructures.

This work presents an empirical study on the capabilities of LLMs at generating PowerShell malware. The objective of this study is to provide security analysts with an approach to understand the potential capabilities of attackers. Moreover, the study supports LLM-based malware generation for *offensive security* purposes, which deliberately emulates adversaries for assessing intrusion detection and threat hunting procedures [13], [14].

Prior work has primarily focused on prompting and jailbreaking strategies to generate malicious code snippets using proprietary LLMs, working around their guardrails [15], [16]. However, the potential of attackers to create their own home-grown LLMs for generating full-fledged malware, rather than individual code snippets, still remains mostly underexplored. This study addresses this limitation by providing a new experimental framework and dataset, to validate the effectiveness of LLMs at emulating real-world malware.

First, we propose *PSStrikes*, a novel, unprecedented curated dataset of real-world PowerShell malware samples to support the empirical study. A key aspect of the dataset is that the malware samples are paired with manually-curated descriptions in natural language (NL). These descriptions reflect how an attacker could instruct an LLM to generate malicious scripts, which are essential for the training and evaluation of LLMs.

Our research also provides a novel experimental framework to evaluating and analyzing malware generated by LLMs. This framework consists of a three-stage pipeline that evaluates how LLM-generated malware compares to real malware samples in the dataset. The evaluation leverages both static and dynamic analysis of the malicious code. In particular, the framework features the use of *sandboxing* techniques to characterize generated malicious behavior, which is a novel approach in the context of research in LLM code generators that is limited to textual analysis and to functional test suites. This analysis is supported by *PSSandman*, a further contribution of this study, which is an automation system specifically tailored for analyzing generated PowerShell malware.

Finally, we present an experimental analysis on permis-

The authors are with the Department of Electrical Engineering and Information Technology (DIETI), Università degli Studi di Napoli Federico II, Naples, Italy. E-mail: {pietro.liguori, vittorio.orbinato, luciano.pianese}@unina.it and Gran Sasso Science Institute, L'Aquila, Italy. E-mail: roberto.natella@gssi.it.
This work has been partially supported by MUR PRIN 2022, project FLEGREA, CUP E53D23007950001 (https://flegrea.github.io); by the Industrial Ph.D. grant (PNRR - DM 117/2023) from MUR and DigitalPlatforms S.p.A, CUP E66E23000580003; and by the ISCRA program that awarded access to the LEONARDO supercomputer, owned by the EuroHPC Joint Undertaking, hosted by CINECA in Italy. We are grateful to Christian Marescalco for the help in the early stage of this work.

sively licensed, open-source LLMs. Our evaluation reveals a concerning finding: high-fidelity PowerShell malware can be generated by LLMs with fewer than 10 billion parameters, even using consumer hardware, leveraging Quantized Low-Rank Adaptation (QLoRa) training procedures and quantization algorithms for deployment. Thus, attackers can potentially use smaller models to avoid security guardrails introduced by proprietary LLMs. Another key finding is that LLM-generated malware frequently diverges from reference implementations (e.g., in terms of textual similarity between generated code and the corresponding real malware sample), while still exhibiting the intended malicious behavior, due to the generalization capabilities of LLMs. This behavioral convergence is empirically validated by a median Jaccard index of 84.5%, with 48.4% of the instances exhibiting complete event overlap compared to real-world malware reference, thereby confirming the importance of using dynamic analysis in evaluating LLM-generated malware.

Hence, this study shows the effectiveness of LLMs in facilitating cyber-adversarial tasks. By delivering a novel methodology, dataset and sandboxing tool, we provide a foundation for future research in offensive security. Overall, our research makes the following key contributions:

- *PSStrikes*[1], a curated human-labeled dataset of real-world malware samples written in PowerShell, accompanied by natural language descriptions for LLMs;
- An *experimental framework* to evaluate LLM-generated PowerShell malware, combining static and dynamic analysis;
- *PSSandman*[2], an open-source sandbox-based system to support the experimental framework;
- An *experimental study* on the capabilities of LLMs in generating PowerShell malware.

A foundational baseline in adversarial PowerShell generation was established in our prior work [17], where we studied LLMs using a specialized dataset of single-line commands. In this work, we transition from individual commands to real-world malware, which introduces several new methodological challenges. In particular, our previous work analyzed relatively-small logs of raw events produced by the single-line commands. In this work, the proposed experimental framework and sandbox architecture have been re-designed to identify high-level malicious traits from raw event logs, enabling a more accurate analysis of behavioral dynamics. Moreover, this work addresses obfuscation, which is prevalent in real-world malware, and it provides more structured descriptions in natural language, enabling the use of LLMs for complex malware. Finally, we investigate ML optimizations that can be used by attackers to deliver maliciously-trained LLMs in resource-constrained environments, by adopting parameter-efficient training and model quantization.

The rest of the paper is organized as follows. We first present our dataset in Section II, and the evaluation framework in Section III. Section IV presents the results of the experimental study. We address ethical aspects in Section V, and discuss threats to validity in Section VI. Section VII reviews related work, and Section VIII draws conclusions.

[1]https://huggingface.co/datasets/dessertlab/PowerShell-attacks
[2]https://github.com/dessertlab/PSSandman

II. Data Collection and Formal Curation

A key element of our research is the release of a curated dataset specifically designed for malicious PowerShell generation. In contrast to extant, fragmented, and unvalidated repositories, our data underwent a rigorous procedure, from acquisition to manual validation and labeling, with the purpose of presenting a novel dataset.

The dataset is composed of malware samples paired with natural language descriptions. The description serves in two ways: first, to reproduce a threat actor interacting with an LLM using malevolent intent to generate a malicious script; and, it functions as inputs, which are essential for fine-tuning a pre-trained model to a downstream task, in our case, PowerShell malware generation.

To the best of our knowledge, this work presents the first structured repository of deobfuscated, real-world PowerShell malware specifically designed to assess LLMs in malware generation. While recent efforts have introduced natural language-annotated datasets in the PowerShell domain, they diverge significantly in both scope and dataset composition. For instance, AutoMalDesc [18] is primarily oriented toward automated malware detection and behavioral summarization. Conversely, works such as RedShell [19], which builds upon our previous work [17], focus on generation but rely only on standard offensive frameworks (e.g., Mimikatz, Nishang) and controlled training environments (e.g., TryHackMe). These samples lack the complexity and adversarial patterns found in genuine malware. Our repository addresses this discrepancy by providing complete, real-world malware samples. We here discuss our procedure for constructing the dataset, as shown in Figure 1, which includes data acquisition and pre-filtering, a deobfuscation pipeline, and a labeling procedure of the malware samples.

A. Data Acquisition and Pre-filtering

The candidate PowerShell scripts were obtained from various sources, including academic research and security practitioners. We initially look for malware sample repositories on GitHub, searching in metadata for collections with a consistent number of PowerShell samples. We selected repositories containing malicious scripts for security analysts [20], [21] and collections such as Malicious-PowerShell-Dataset [22], which comprises itself a collection of diverse sources, including Malware Bazaar [23], Hybrid Analysis [24], and Triage. These sources contain malware samples analyzed and shared by forensic specialists. Then, we incorporated PowerShell scripts from offensive security frameworks, such as Nishang [25] and PowerSploit [26]. In addition, we included data from Chai et al. [27], which proposed, along with a deobfuscation framework, a collection of obfuscated PowerShell scripts from the wild gathered by a cybersecurity firm from enterprise infrastructures.

We ensured data consistency and suitability for LLMs by applying a strict filtering protocol with the following criteria:

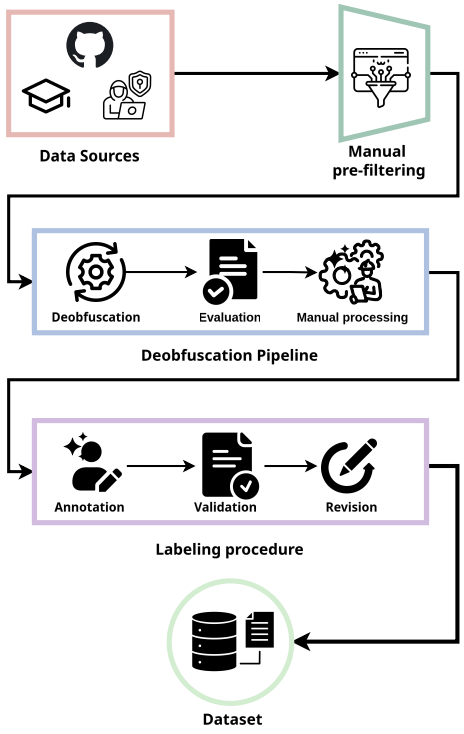


Fig. 1: Dataset construction procedure

- **Format Integrity:** Among the data, we filtered out all non-PowerShell files, such as binary executables, which are outside the scope of this work. Furthermore, we have exclusively selected scripts that are fully executable via PowerShell.
- **Structural Constraints:** Scripts were limited to approximately 250 lines of code by aligning with a 4,096-token context window supported by both smaller and larger LLMs. In our experimental study, we aim to evaluate attackers with limited hardware resources, who cannot afford the training of LLMs with large context windows. Moreover, training with larger windows is potentially affected by the *Lost in the Middle* effect [28], where model precision on core logic degrades. The selected window size suffices to obtain a still large number of samples of real, complex PowerShell malware. Although large modular toolkits are excluded, the dataset effectively covers the most common and dangerous PowerShell malware archetypes. By focusing on scripts under 250 lines, we analyze the standalone obfuscated payloads that represent a representative share of detected threats.
- **Remove redundancy:** We conducted manual deduplication to remove scripts with identical functional logic but varying payloads (e.g., different C2C URLs).

In total, the malicious scripts from this collection consist of 830 samples. These samples will be further filtered by the next stages of our procedure.

Finally, our dataset builds on top of the dataset from our previous work [17], which was limited to malicious one-line PowerShell commands. It includes $1,127$ samples obtained from offensive security frameworks, such as Atomic Red Team [29], StockPile [30], and Empire [31], and from public knowledge bases from the web. We include these data in the PSStrikes dataset since they can still contribute to the training and evaluation of LLMs for code generation, and they reflect individual actions that are part of wider attack campaigns.

In summary, the dataset encompasses various types of malicious code, including payloads from penetration testing tools, one-line commands commonly used by attackers, and complete malicious scripts from the field, which form a representative sample of real-world PowerShell malware.

B. Deobfuscation Pipeline

Obfuscation is a pervasive technique in modern malware development, particularly within interpreted scripting languages like PowerShell. This technique is primarily employed to circumvent static signature-based security in Endpoint Detection and Response (EDR) systems, as well as to bypass detection systems such as Anti-Malware Scan Interface (AMSI) and to hinder forensic analysis.

For experimentation purposes, training an LLM on obfuscated scripts would conflate two different learning objectives: malware generation and its obfuscation. This approach would not reflect actual malware development, where obfuscation procedures are applied subsequently to writing malware, e.g., through dedicated obfuscation frameworks [32]. Moreover, conflating these objectives during model training would contrast with best practices for the use of LLMs, which benefit from decomposition to simpler tasks [33].

To guarantee cleaned data, our process includes a deobfuscation stage. The stage follows a tiered approach, making use of two widely used tools to deobfuscate and validate data. We started by quantifying the obfuscation level of the samples using Revoke-Obfuscation [34], a tool that provides both the count of obfuscated lines and the specific detection rationale (e.g., character frequency anomalies or encoding patterns) as an indicator of obfuscation. In this study, we adopted a conservative threshold for classification: a script is labeled as obfuscated if it contains at least one flagged line of code. The objective of this constraint is to optimize the recall on code that has been flagged as obfuscated. In the event that the code remains in its false positive state, it will be addressed in a subsequent manual stage of the process. Then, we employed PowerDecode [35], [36] to handle common obfuscation techniques, such as string concatenation, Base64 encoding, and ASCII compression. PowerDecode is a reliable deobfuscation tool that supports the deobfuscation of several obfuscation techniques through dynamic execution in a sandboxed environment.

For the scripts that remained obfuscated after this phase, we conducted a manual refinement. The residual subset was sufficiently small to permit a rigorous, line-by-line inspection. This inspection was initially performed by two junior security analysts with the support of an LLM[3]. The lead

[3]As discussed in Section IV-A, we are unable to disclose the name of the LLM due to licensing restrictions. The use of the LLM was limited to small, individual snippets extrapolated from the malware, by framing queries in the context of forensic analysis for academic purposes.

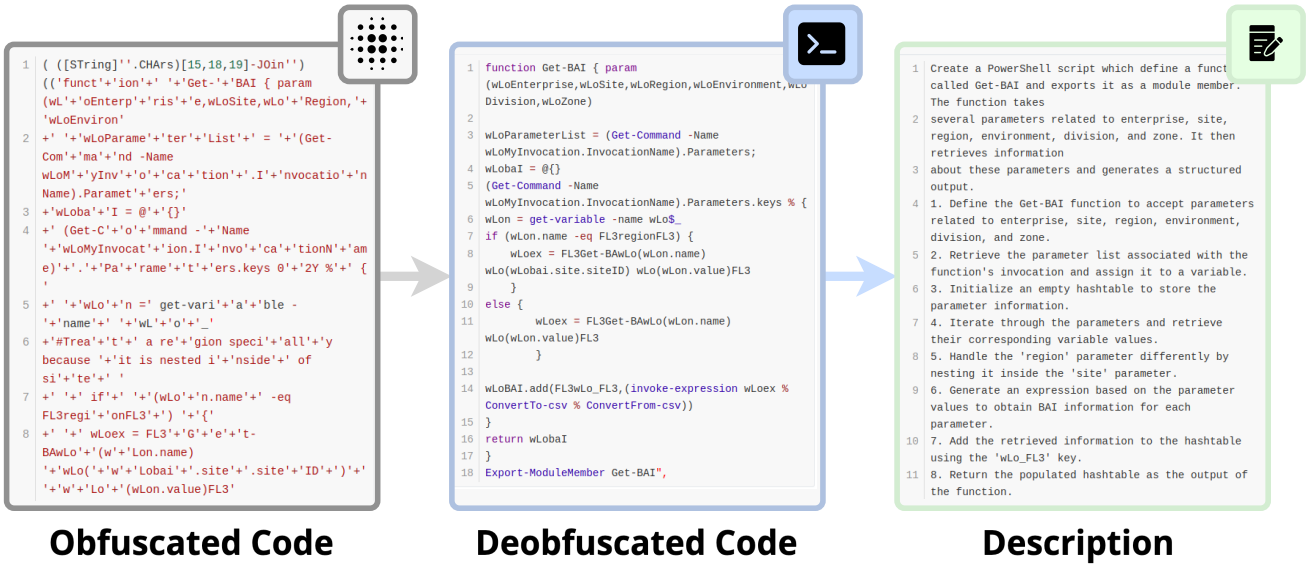


Fig. 2: Example of deobfuscation and labeling.

author validated the de-obfuscated code, by ensuring that it integrates with the other parts of the script that were already deobfuscated, and that it aligns with the malicious intent of the malware. In cases of disagreement, the code was reviewed and discussed by all co-authors.

The deobfuscation stage enabled the resolution of sophisticated patterns that were identified by Revoke-Obfuscation but not resolved by PowerDecode. An example of a deobfuscated sample is shown in Figure 2. The image showcases a multi-layered PowerShell obfuscation pipeline that leverages metadata access, late-binding syntax restoration, and obfuscation of the data binding, whereas the de-obfuscated version explicitly incorporates PowerShell keywords, eliminating any requirement for intermediate interpretation and JIT (Just-In-Time) string manipulation.

This tiered approach was effective at significantly reducing obfuscation. The raw collection initially exhibited 25% of the total samples flagged as "obfuscated". After automated processing via PowerDecode, obfuscated-flagged samples drop to 11% of the entire collection. Finally, the manual analysis further lowered the presence of obfuscated samples to a negligible 1.53% (13 samples out of 830), resulting in a high-fidelity dataset optimized for offensive code generation tasks. The remaining 13 samples still flagged as "obfuscated" scripts were dropped from the final dataset, resulting in the final 817 subset of real-world deobfuscated samples.

C. Semantic Enrichment and Natural Language Annotation

To establish a ground truth for our experimental framework, we manually annotated deobfuscated scripts with descriptions in natural language (NL). For each script, we provide both a **High-Level Summary** and an **Operational Specification**. The former is a paragraph that defines the overall objectives and scope of the malware, using a few short sentences. The latter consists of a granular step-by-step description, where each step is an individual operation to be performed by the script, and which will result in one or a few lines of generated code. These step-by-step descriptions use imperative verbs and reference OS resources and Application Programming Interfaces (APIs) (e.g., file system, registry) to be accessed by the generated code. This style reflects the prompts used by LLM-based malware found in the wild [4], [8], [5].

To ensure the accuracy of the dataset, we implemented a rigorous validation protocol consisting of three rounds of human review. This process begins with two junior security analysts responsible for creating a first version of the descriptions via an LLM-assisted human-in-the-loop process. Then, the descriptions are reviewed by the lead author and finally approved by all co-authors. The review checks the alignment between the NL narratives and the actual execution logic of the PowerShell scripts, by adopting the following criteria: (i) **Procedural Sequencing**: Evaluates the chronological integrity of the description, confirming that it accurately reflects the execution flow of the script. (ii) **Completeness**: Ensures that the description encompasses the entire adversarial lifecycle within the script, from initial environment preparation to the final payload execution, without omitting critical malicious stages or intermediate stagers. (iii) **Technical Fidelity**: Verifies that the description correctly references OS resources and APIs used by the malware, such as .NET assemblies and WMI classes, ensuring that the underlying functional primitives are correctly identified. (iv) **Contextual Grounding**: Assesses the precision in identifying affected Windows system artifacts, including specific registry keys, environment variables, and defensive telemetry interfaces (e.g., AMSI, ETW) manipulated by the script. (v) **Syntactic Consistency**: Enforces a uniform, formal, and objective imperative tone across the corpus, ensuring that the descriptive language adheres to real-world malware descriptions. To ensure quality, we implemented a cross-validation protocol where annotators reviewed each other's work. This yielded an 87% initial acceptance rate; the remaining 13% of cases were adjudicated by the lead author,

TABLE I: Descriptive statistics of the dataset.

Metric	Value
Dataset Volume & Uniqueness	
Total Samples (N)	$1,944$
Unique Code Snippets[a] (%)	97.95%
Code Description Description Statistics (Tokens)[b]	
Average Length (μ)	104.96
Code Statistics (Tokens) [b]	
Average Length (μ)	396.20
Code Description Linguistic Diversity	
Corpus Vocabulary Size	$5,470$
Corpus Lexical Diversity (TTR)	0.030

[a]Based on SHA-256 hash of whitespace-normalized code.
[b]Computed using the GPT-2 BPE tokenizer.

with all authors formally approving the final process.

This pipeline yielded a final corpus of unique, high-fidelity samples. In total, the PSStrike dataset is the union between the $1,127$ one-liner scripts from our previous work, and the 817 new real-world malware scripts from the multi-stage deobfuscation and annotation procedure. The dataset is made available to the scientific community. Table I provides a comprehensive overview of the dataset distribution. The dataset exhibits high structural integrity with 97.95% unique code snippets across $1,944$ samples, and utilizes dense, domain-specific natural language descriptions ($TTR \approx 0.03$, comparable with other instruction-based datasets) effectively minimizing redundancy and data leakage risks. The code's pronounced long-tail distribution, with outliers exceeding 2500 tokens, confirms the genuineness of our data as it accurately reflects the complexity of real-world deobfuscated attacks. The mean length of 104.96 tokens ensures the dataset is representative of real-world prompts, which typically take several dozen words to articulate malware operations.

III. Evaluation Framework

We designed an evaluation framework to assess malware generated by LLMs (Figure 3). It consists of three stages, which look at complementary perspectives: code similarity, code quality, and dynamic behavior of generated malware, compared to ground truth from our dataset.

A. Code Similarity

In code generation tasks, a common approach is to assess the similarity of the LLM's output with respect to a reference ground truth. This initial validation relies on string-based metrics to quantify the adherence of the generative synthesis against reference samples (in our case, real-world malware). Historically, previous work evaluated generated code using metrics for Neural Machine Translation (NMT), such as BLEU and Edit Distance (ED), which measure lexical overlap between string. However, a study from Evtikhiev et al. [37] demonstrated that these classical metrics derived from NMT only exhibit a marginal correlation with functional correctness, largely because they fail to account for the discrete and formal nature of programming languages. Hence, for our study, we selected two of the metrics recommended by Evtikhiev et al. [37], and added a metric specifically designed for code output similarity, based on the traditional BLEU metric. The metrics chosen are:

- **ChrF**, proposed by Popović [38]. It operates by averaging precision and recall over character n-grams (1 to 6). This character-level granularity is particularly effective in capturing the structural nuances of malware code.
- **METEOR** [39]. A metric based on the harmonic mean of unigram precision and recall, calculated through an explicit mapping of token-level unigrams between the generated and reference code. Unlike simpler overlap measures, it incorporates a penalty for fragmentation to account for the structural alignment of code sequences.
- **CrystalBLEU**, proposed by Eghbali and Pradel [40]. This metric adapts standard BLEU for code generation tasks by filtering trivially frequent n-grams. By reducing the weight of repetitive syntactic boilerplate, it yields a score that correlates more closely with functional similarity, making it highly suitable for programming languages.

These metrics evaluate code similarity from different perspectives: ChrF captures character-level insights, METEOR evaluates sentence-level lexical overlap, and CrystalBLEU isolates functional logic from syntactic noise. This approach mitigates the inherent biases of individual scoring mechanisms. Consequently, the resulting scores offer a holistic proxy for the structural and functional fidelity of the synthesized PowerShell scripts.

B. Code Quality

The second perspective of the evaluation is code quality. While similarity metrics provide a proxy for model accuracy compared to a reference, they often fail to capture the functional integrity and the architectural quality of generated code. For instance, a high CrystalBLEU score may mask deep-seated structural deficiencies within the synthesized PowerShell code. Specifically, a script may mirror the reference's token distribution while disregarding the strict verb-noun naming conventions or the object-oriented nature of PowerShell pipelines. Such scripts often harbor "silent" failures (such as improperly scoped variables or broken downstream dependencies) that, while appearing textually similar to the ground truth, do not reflect real PowerShell programs. To evaluate code quality, we analyze both the syntax correctness of the generated script and its adherence to PowerShell language best practices.

We first evaluate the syntactical correctness of the scripts, by checking whether they are interpretable by the PowerShell parser [41]. Errors such as `UnexpectedToken` typically indicate a failure of the generative model to adhere to the formal grammar constraints of the language. We measure the percentage of generated scripts that are fully interpretable by PowerShell ("Syntax Correctness" metric).

Additionally, we perform static code analysis for a deeper evaluation of the code. We used PSScriptAnalyzer [42], a static analysis tool for PowerShell maintained by Microsoft. This tool is an industry-grade analyzer that represents the authoritative standard for PowerShell code quality. The tool

inspects PowerShell scripts by leveraging the Abstract Syntax Tree (AST) representation of the code to identify structural and semantic irregularities. It evaluates code against a collection of built-in rules derived from best practices in PowerShell programming [43], [44], [45]. Using static analysis, we measure the share of generated malware that does not trigger violations ("Clearness" metric). Moreover, our evaluation extends into the full diagnostic breadth of PSScriptAnalyzer, analyzing the complete spectrum of markers provided by this tool, including Errors, Warnings, and Information severities.

To provide context for these metrics, we measure them both on generated malware and on real malware from the dataset. The analysis enables us to evaluate the generated scripts across multiple dimensions, providing a detailed assessment of code quality.

C. Code Execution Behaviour

Finally, we introduce a dynamic runtime evaluation of the generated malware samples. Code similarity and static analysis are insufficient for evaluating LLM-generated malware, as these aspects are subordinate to the malware's actual behavior. For instance, in the context of PowerShell malware, an LLM might generate a script that exhibits high structural similarity to known malicious patterns, but still fails to perform the intended operations on OS resources and APIs due to minor, yet severe errors (e.g., unhandled environmental constraints, such as PowerShell execution policies). Conversely, syntactically anomalous code, such as unusual patterns for dynamic API resolution, might score poorly on static similarity metrics yet successfully achieve unauthorized code execution. Therefore, assessing the true capability of an LLM to generate viable malware strictly requires executing the samples within a controlled, representative infrastructure to verify their runtime behavior and evasion capabilities.

We introduce PSSandman, a novel, purpose-built execution framework for PowerShell malware. PSSandman performs dynamic analysis in a controlled Windows environment. The sandbox executes malware within an instrumented virtual machine (VM), and retrieves event data for subsequent analysis. Both the generated malware and the corresponding real-world ground-truth malware are executed, in separate VM instances. The VM is controlled by an automation script responsible for transmitting commands and acquiring event data generated by the execution, using host-VM communication APIs provided by the hypervisor [46]. To ensure correct execution and analysis, the framework executes all samples from the same initial "safe" state using a snapshot, returning to the safe state after each execution. Snapshotting guarantees that each malware executes under the same conditions, and isolates executions from disruptive behavior of previous runs. The sandbox adopts VirtualBox version 7.2.2 as hypervisor, and Windows 10H2 as guest OS for the virtual machine. We simulate a post-compromised scenario in which an attacker gets the opportunity to run the malware on the target machine (e.g., by persuading the user to run an executable, or by gaining initial access by exploiting a software vulnerability). Thus, security mechanisms are disabled to allow the malware to run. Furthermore, the virtual machine has been configured to ensure a "stimulating" environment for malware execution. For example, to facilitate the proper execution of keylogger samples, we have injected pre-recorded keyboard events simulating the human interaction with the machine. Furthermore, we have installed Living Off The Land (LOL) Binaries and penetration testing frameworks such as Mimikatz [47] and PowerSploit [48] to enable the execution of all the samples that require them. Finally, we implemented a responsive network emulation layer that intercepts all outbound traffic, serving synthetic executable payloads to satisfy arbitrary malware dependencies and maximize behavioural exposure. The core of the framework is a Python script that configures the virtual machine, by installing external software packages, and that orchestrates the virtual machine, and the upload and execution of malware scripts.

Our sandbox applies system monitoring techniques to collect event logs, and uses events to characterize malicious behavior. In general, an event is an operation performed on system resources and APIs, e.g., process creation, registry modifications, and file system interactions. PSSandman processes event logs to identify *malicious traits* of an execution, that is, high-level actions performed by an attacker to achieve a tactical goal (e.g., escalating privileges, getting persistence, exfiltrating data) [49]. This abstraction allows PSSandman to distinguish between routine system activity and purposeful adversarial maneuvers, making the comparison between generated and real malware more accurate that comparing raw events. For monitoring events, we leverage *Sysmon* [50], a component of the Windows Sysinternals suite. Sysmon runs a background service and device driver that monitors objects in the kernel and logs activities on them, including process creations, network connections, and file changes, directly into the Windows Event Log. Furthermore, to achieve more visibility into script-based behaviors, we augment our audit streams with the *PowerShell Operational log*. This integration allows us to capture raw script blocks (e.g., Event ID 4104) and dynamic module executions, exposing memory-resident commands like Invoke-Expression that typically remain opaque to standard kernel-level monitors. To complete our monitoring architecture, we added standard Windows System and Security logs.

Then, detection of malicious traits is achieved via *heuristic rules* that analyze event attributes and multi-event sequences, including both kernel-level events, such as filesystem accesses and Windows registry operations, and user-level events, including PowerShell events and the process lifecycle. We adopt detection rules written using Sigma, a vendor-agnostic domain-specific language (DSL) designed for standardized specification of detection rules [51]. By adopting this open standard, our system leverages signatures crafted by the security research community to identify relevant malicious traits. In particular, we derive the detection rules from SigmaHQ [52], the official repository for the Sigma project. We only remove rules that are triggered by generic events, such as "Non-Interactive PowerShell Process Spawned" and "New PowerShell Instance Created", that are triggered by the mere execution of the scripts and do not offer additional insights.

To detect the occurrence of malicious traits in the malware,

we analyze the events using the Sigma rules through the open-source engine *Zircolite* [53]. The Sigma rules also provide additional insight on malicious behaviors, by mapping rules to Tactics, Techniques, and Procedures (TTPs) in the MITRE ATT&CK taxonomy. In this work, we adopt MITRE ATT&CK version 18.1 [54]. For instance, a generated PowerShell script that uses dynamic .NET reflection to bypass the Anti-Malware Scan Interface (AMSI) would trigger a Sigma rule targeting the `amsiInitFailed` pattern within the System Management Automation namespace, mapped to the T1562.001 technique of the TA0005 tactic in MITRE ATT&CK version 18.1 [55].

This process allows us to accurately identify the most severe and recurring malicious traits in generated malware, based on mature knowledge bases from the research community. Table II illustrates the distribution of Sigma rules across various log sources and their respective threat severities. Another perspective is given by Table III which maps the ruleset to the Tactics of MITRE ATT&CK framework.

Based on detection rules, we assess behavioral equivalence between generated malware samples and ground-truth ones. We frame the assessment as a strict set-similarity measurement problem. We handle the Sigma rules triggered by both the generated payloads and the ground truth as discrete sets, and evaluate their overlap. Let G_i denote the set of rule IDs dynamically triggered by the i-th generated sample, and T_i the set of rule IDs triggered by the corresponding ground-truth sample, evaluated over n independent trials. Since the triggered rules are discrete, non-semantic identifiers within a highly sparse domain, traditional multi-label classification metrics such as Hamming Loss are structurally inadequate, as their reliance on true negatives invariably leads to an accuracy paradox in sparse environments [56]. Consequently, we adopt the following set-theoretic evaluation metrics:

Exact Match Ratio (Subset Accuracy): Originating from the evaluation of multi-label classification systems [57], this is the strictest criterion. It requires the behavioral footprint of the generated code (G_i) to perfectly match the ground truth (T_i) for a given file i, yielding neither false positives nor false negatives. It is formally defined via the indicator function $\mathbb{I}$:

$$EMR_i = \mathbb{I}(G_i = T_i) \tag{1}$$

By evaluating EMR_i across all trials, we obtain a binary distribution whose sample mean ($\overline{EMR}$) represents the overall expected subset accuracy.

Jaccard Similarity: Initially introduced as the *coefficient of community* [58], this serves as our primary set-theoretic measure for proportional overlap. For a single file i, it is defined as:

$$J_i = \frac{|G_i \cap T_i|}{|G_i \cup T_i|} \tag{2}$$

Crucially, J_i does not assume a closed vocabulary and is intrinsically invariant to label sparsity [59]. We analyze the distribution of J_i across files alongside its median value $\tilde{J}$.

Sørensen-Dice Coefficient: Formulated independently to measure ecological association [60], [61], this quotient is mathematically equivalent to the instance-level F1-score. By assigning double weight to the intersection, it effectively emphasizes mutually agreed-upon system behaviors over divergent ones for each file i:

$$D_i = \frac{2|G_i \cap T_i|}{|G_i| + |T_i|} \tag{3}$$

Extracting the distribution of D_i allows us to characterize file-level variance, which is then summarized by its median $\tilde{D}$.

Symmetric Difference: To quantify the absolute error without the bias introduced by true negatives, we measure the cardinality of the symmetric difference for each file i. This metric effectively replaces the traditional Hamming Loss by returning the absolute integer count of unshared, diverging rules (i.e., the sum of hallucinated and missed events):

$$\Delta_i = |G_i \triangle T_i| = |G_i \cup T_i| - |G_i \cap T_i| \tag{4}$$

The statistical dispersion of Δ_i characterizes the magnitude of absolute errors, with its median value $\tilde{\Delta}$ representing the expected per-file divergence.

To provide a comprehensive statistical characterization of the behavioral discrepancies, we analyze not only the median of each metric but also their empirical distributions and quantile limits across the test dataset

TABLE II: Sigma rules by data source and severity.

Log Source	Crit.	High	Total	Key Behaviors
Sysmon (Proc.)	22	65	87	Parent-Child, LOLBins
Sysmon (File/Reg)	15	28	43	Persistence, Wipers
Sysmon (Net/DNS)	5	12	17	C2 Beacons, Exfil
PS Operational	18	10	28	Obfuscation, IEX
System/Security	8	15	23	UAC Bypass, PrivEsc
Total Rules	**68**	**130**	**198**	—

TABLE III: Sigma rules by MITRE ATT&CK tactics.

MITRE Tactic	Targeted Artifacts	Sev.	Rules
Discovery	Sys/Net recon, enumeration	High	12
Execution	Cradles, WMI/DCOM, IEX	Crit.	45
Persistence	Run keys, Tasks, Backdoors	Crit.	38
Priv. Escal.	UAC bypass, Token manip.	Crit.	25
Def. Evasion	Sideloading, AMSI/ETW	Crit.	42
Cred. Access	LSASS dump, SAM export	Crit.	30
C2	DNS beacons, Rev. Shells	Crit.	18
Exfiltration	BITSadmin, Web-transfer	High	10
Impact	Shadow Copy, Wipers	Crit.	7

IV. Experimental Results

In the following, we present an empirical investigation of LLM-based PowerShell malware generation, using our proposed evaluation framework and dataset.

A. Experimental setup

We focus our analysis on *open-weight* LLMs, which ensure the full reproducibility of experiments, and comprehensive oversight of the training and inference operations. Moreover, we select LLMs whose license does not forbid offensive security applications. Because of these reasons, we do not address in this work proprietary LLMs that are only accessible through remote APIs, which lack deterministic versioning and

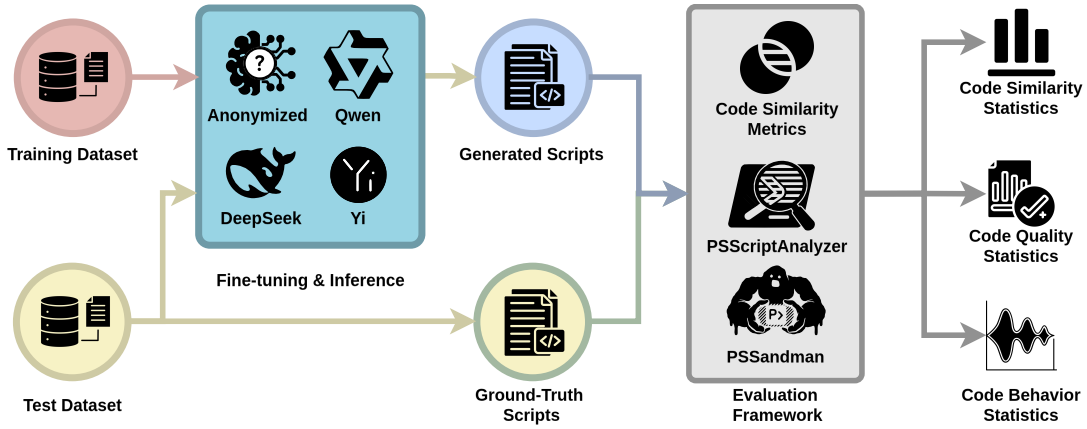


Fig. 3: Evaluation framework.

often employ opaque safety-alignment layers that censor the generation of security-relevant code. Moreover, these models come with "Acceptable Use" policies that strictly prohibit malevolent usage, even for research purposes.

Adhering to the aforementioned constraints, our setup encompasses three compliant LLM candidates. To isolate the effects of our fine-tuning process, we selected LLMs with dense- decoder-only architecture in the 7-9 billion parameters. Furthermore, we opted for instruction-optimized versions to facilitate the processing of the imperative-style prompts that characterize our malware dataset. The models chosen are:

- *Yi-Coder-9B-Chat* (hereafter Yi) from 01-AI [62], chosen for its proficiency in long-sequence dependencies;
- *Qwen2.5-Coder-7B-Instruct* (hereafter Qwen) is representative of high-density instruction tuning from the second generation of the Qwen coder family [63], the same family used in the LAMEHUG campaign [5];
- *Deepseek-coder-7b-instruct-v1.5* [64] (hereafter Deepseek) due to its widespread adoption as a state-of-the-art baseline for code generation tasks.

Moreover, we consider fourth popular open-weight model often used as baseline in the research literature. The license of this model prohibits its use for generating malicious code, without exceptions for research purposes. However, since the model can still be deployed internally on a local machine, we opted to experiment on it, without disclosing any of the malware generated by this model, and without sharing our source code that we used to train and run the model for generating malware. In the following, the name of the model has been anonymized. The detailed technical specifications and licensing of the selected model are presented in Table IV.

The experiments have been designed to align with realistic operational scenarios in automated malware generation. Our preliminary evaluations showed us that pre-trained, "off-the-shelf" models lack the domain-specific accuracy required to generate functional PowerShell malware, as PowerShell is under-represented in the training data of pre-trained models [65]. Consequently, we investigate *fine-tuning* techniques to adapt pre-trained models to PowerShell code generation. It is also important to note that adversaries can adopt different fine-tuning methodologies and deployment architectures, depending on their specific operational constraints (such as, hosting an LLM on a high-resource remote server, or locally on a resource-constrained machine). Hence, to provide a comprehensive evaluation, we define two usage strategies representing the functional extremes of current AI training and deployment paradigms. These strategies correspond to two distinct adversarial profiles: 1) **High-Resource Attack**: This scenario assumes that code generation runs on an independent infrastructure controlled by the attacker, with sufficient resource availability. Therefore, the model is fine-tuned and deployed at Baseline BF16 precision (hereafter "native precision") on an attacker-controlled server. 2) **Resource-Constrained Attack**: This scenario models an attacker with limited computing resources, or that embeds the LLM in the malware and runs it within the victim's resource-constrained environment [4]. In this deployment, we use a combination of QLoRA for training and 4-bit AWQ quantization for inference.

We executed fine-tuning and inference on a system with 32-core Intel Ice Lake CPU with four NVIDIA A100 SXM4 64GB GPUs from the HPC Leonardo infrastructure. For inference, we used the vLLM serving engine version 0.17.1. All fine-tuning and inference operations were performed using the original implementation frameworks provided by the model developers.For evaluation purposes, we split the dataset into a training and a test set. The test set includes both real-world malware samples and one-line malicious commands. It is obtained through stratified sampling, by taking 15% of the malware scripts, then adding 10% of the one-line commands. This procedure was designed to assure the contribution of real-world malware in both training and evaluation. All LLMs were trained and evaluated with the same data split.

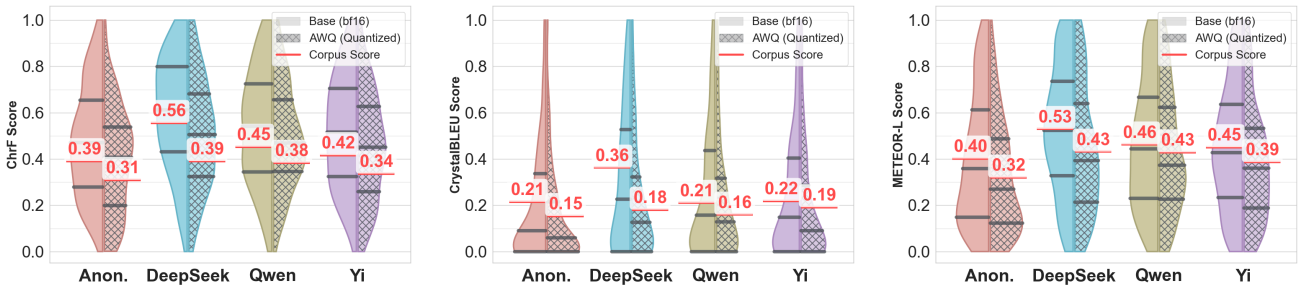
B. Code similarity results

The overall results are shown in the violin plot in Figure 4, which shows the distribution of code similarity metrics computed across the corpus, with median values annotated (Corpus Score). Regarding the native precision models, in each metric, the scores exhibit a uniform trend: the relative ranking of the LLMs is consistent across metrics, where DeepSeek achieves the best score. The trend also applies to the quantized models, with DeepSeek achieving the highest

TABLE IV: Technical profile of LLMs for PowerShell malware generation.

Model	Developer	Params	Context	Training	License	Key Rationale & Strength
Yi-Coder-9B-Chat	01-AI	9B	128K	N/A*	Apache 2.0	High parameter efficiency; SOTA performance on LiveCodeBench [62].
Qwen2.5-Coder-7B-I	Alibaba	7.6B	128K	5.5T	Apache 2.0	Advanced logic reasoning and instruction-following capabilities [63].
DeepSeek-Coder-1.5-7B-I	DeepSeek	7B	128K	2T	Custom, Liberal	Specialized repo-level pre-training for structural code integrity [64].
Anonymized Baseline	Anonymized	7B	4K	500B	Custom, Restricted	Standard safety-aligned baseline.

**The official documentation for Yi-Coder emphasizes architectural efficiency over raw token count.*



(a) ChrF (b) CrystalBLEU (c) METEOR-L

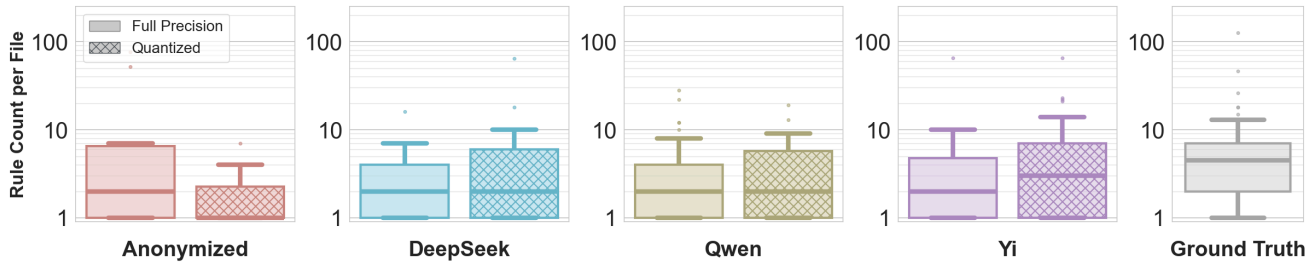
Fig. 4: Code Similarity metrics across model families.

ChrF and METEOR scores, and second CrystalBLEU score, by a one-point margin compared to Yi. By comparing native-precision with quantized models' scores, we notice a reduction of the corpus score for all metrics, with distributions still comparable to the ones of native-precision models. DeepSeek exhibits the most significant performance drop, with corpus scores declining by double-digit percentages and a visible shift in its metric distribution. Among the models, Qwen it seems to be the most resilient to quantization.

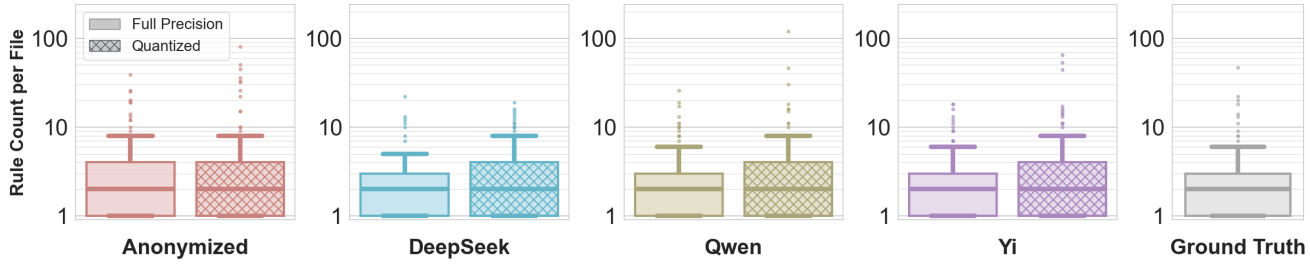
Analyzing CrystalBLEU indicates that Anonymized, Qwen, and Yi exhibit statistically similar distributions; however, these models show significantly lower logical fidelity to the ground truth compared to DeepSeek. However, this performance shifts under quantization, where the DeepSeek advantage becomes attenuated. The models demonstrate coherent performance in character-level overlap, with DeepSeek consistently outperforming other implementations. This observation extends to sentence-level overlap, thereby validating the trends observed via ChrF and indicating that the native-precision DeepSeek model exhibits the highest degree of syntactical alignment with the ground truth. The experimental findings demonstrate high consistency across code similarity metrics; notably, the DeepSeek model maintains significantly higher syntactic fidelity to the ground-truth references than other configurations.

C. Code quality results

The first dimension of code quality evaluated is the proportion of non-parsable scripts generated by the LLMs. As detailed in Table V, the Syntax Correctness Rate remains relatively consistent across all models; DeepSeek emerges as the most robust, with a non-parsable rate of only 6.6%. A key finding is that the parsability of scripts remains unaffected by QLoRa training and quantization, except for Yi, which shows a 16.1 percentage-point decline in performance upon quantization. To investigate this discrepancy, we analyzed the most frequent Parsing Errors. For the Anonymized, DeepSeek, and Qwen models (both native-precision and quantized formats), the primary errors are `MissingEndCurlyBrace` and `UnexpectedToken`. These flags indicate truncated output where the model fails to generate a complete logical block, a limitation likely tied to token-generation constraints. In contrast, the quantized version of Yi exhibits a significantly higher frequency of `MissingArgument` errors than any other model. This suggests that for Yi, quantization does not merely lead to premature termination but fundamentally compromises the model's ability to adhere to PowerShell's command-argument syntax, indicating higher sensitivity to precision loss in its syntactic representations. Another view is the percentage of scripts that have triggered no rules. The Clearness rates follow the trend underlined by the Syntax Correctness Rate. DeepSeek has a higher score in native precision, and quantized Yi has the lower one. Furthermore, the comparison with the ground-truth evidence shows that mostly generated malware tends to be "clean". It is reasonable to expect that an LLM is more likely to generate code that is neater than that of a threat actor, due to the extensive pre-training of LLMs. This phenomenon can be noticed by examining the distribution of the number of rules triggered per file, divided by severity, as illustrated in Figure 5a, where the distribution of violations at "Warning" severity level is comparable to the ground truth, and the violations at "Information" level are even less frequent. It is also worth noting that static analysis did not raise any violation at "Error" severity level among generated samples, as well as in the ground-truth. We



(a) Severity: Information.



(b) Severity: Warning.

Fig. 5: Distribution of static analysis violations, categorized by severity level.

TABLE V: Syntactic evaluation for Base and AWQ models.

Model	Syntax Corr. (%)	Clearness (%)
Anonymized	78.8	43.2%
Anonymized-AWQ	79.2	33.4%
DeepSeek	93.4	44.0%
DeepSeek-AWQ	93.4	42.7%
Qwen	86.8	42.3%
Qwen-AWQ	82.6	42.3%
Yi	84.7	41.9%
Yi-AWQ	68.6	36.4%
Ground Truth	-	38.9%

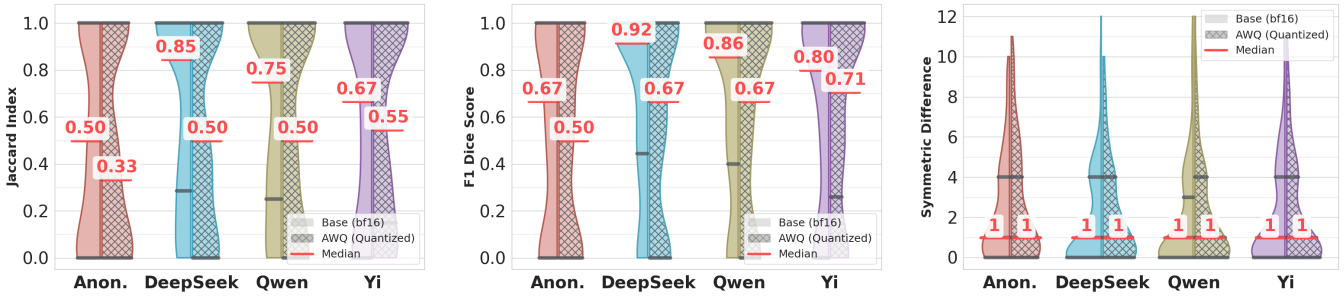
conclude that generated code achieves a reasonable quality for adversary emulation purposes.

D. Code execution results

Starting from our test set of 236 samples, we focus on analyzing malware for which the ground truth triggers at least one Sigma detection rule, which amounts to 212 samples Table VI shows the mean values of the "Exact Match Ratio"; moreover, Figure 6 shows the violin plot of the distributions of Jaccard Index, Dice Index, and Symmetric Difference, highlighting the median values. In the native-precision configuration, the models demonstrate a varying but generally robust capacity for structured event generation. Once again, DeepSeek achieves the best performance, achieving a median Jaccard Index of 84.5% and an Exact Match Ratio (EMR) of 48.4%. Beyond individual performance of the models, these data reveal a general trend regarding structural fidelity: across all models, the consistently high median Dice Index (three out of four models with $\geq 80\%$) and the squashed Symmetric Difference (Δ) distributions indicate that wrong execution behaviors have limited impact. They represent "near-misses", where the core event logic is captured, but marginal attributes are either omitted or hallucinated. Furthermore, the median Symmetric Difference $\tilde{\Delta}$ is 1 across models, suggesting that generated malware is limited to only few unshared events. Probability densities concentrate around higher values for Dice and Jaccard, and lower values for $\tilde{\Delta}$.

Quantization has a significant impact on the execution behavior of malware by high-performing models, specifically DeepSeek and Qwen. The collapse of DeepSeek's EMR from 48.4% to 35.9% suggests that the high-dimensional features required for exact set-matching are highly sensitive to bit-depth reduction. Conversely, the other models, Yi and the [Anonymized] candidate, manifest minimal performance degradation. Crucially, the underlying drivers differ: the plateau of the [Anonymized] candidate stems from its low baseline performance in BF16, while Yi demonstrates genuine resilience, maintaining a narrow performance gap across all evaluation metrics and the best score in quantized configuration. To rigorously evaluate the impact of quantization on execution behaviors, we performed Wilcoxon Signed-Rank tests on the paired Jaccard Index scores (BF16 against AWQ models), applying the Benjamini-Hochberg False Discovery Rate (FDR) correction to account for multiple comparisons ($\alpha = 0.05$). This statistical test highlights that quantization introduced a performance penalty for specific architectures: DeepSeek and Qwen exhibited statistically significant drops in accuracy ($p < 0.001$ and $p = 0.041$, respectively). Notably, the Yi and [Anonymized] models demonstrated no significant differences across either event sources ($p = 0.855$ and $p = 0.305$, respectively).

In conclusion, the overall results demonstrate that the generated malware behavior closely aligns with real-world references. The high Jaccard Index, corroborated by even



(a) Jaccard Index (b) Sørensen-Dice Index (c) Symmetric Difference

Fig. 6: Distribution density for set-based metrics based on Sigma rules.

TABLE VI: Exact Match Ratio for execution events.

Model	Exact Match Ratio	
	Native Precision	Quantized
Anonymized	30.8%	32.0%
DeepSeek	48.4%	35.9%
Qwen	46.6%	39.2%
Yi	41.3%	40.2%

higher Dice Index values, confirms that behavioral deviations are limited. Furthermore, almost 50% of the generated scripts achieve perfect alignment in the best case, while the median Symmetric Difference shows that the remaining mismatch is narrow. Finally, quantization yields significantly degraded results. Nevertheless, these models retain the capacity to pursue malicious behaviors, even in unfavorable configurations.

V. Ethical Considerations

Our research is positioned within the domain of *offensive security*, a discipline that analyzes adversarial artifacts and methodologies to deepen the systemic understanding of contemporary threats [13], [14] The objective of this work is to facilitate the development of robust, evidence-based detection mechanisms against AI-generated malware.

However, the disclosure of techniques to automate malware generation brings concerns about the dual-use of that knowledge by malicious actors, and about self-harm to benign users. We adhere to the ethical framework outlined in the Menlo Report [66], [67]: we have performed a risk-utility analysis, concluding that the scientific benefit, i.e., equipping defenders with means to defend against AI-generated malware, outweighs the marginal risk of providing adversarial actors with LLM-driven automation. From a utilitarian standpoint, the "net utility" of this research is positive. As LLMs become ubiquitous, providing the community with a controlled study and a dedicated sandboxing architecture prevents a future state where defenders are unprepared for high-volume, LLM-generated attacks. As a matter of fact, we are already witnessing AI-generated malware in the field [4], [5], [8], [9].

We recognize a moral duty to avoid harms alongside a professional duty towards scientific transparency. To reconcile these, we share our findings while carefully controlling the distribution of dangerous artifacts.

The dataset and LLMs for this study exclusively consists of public-domain artifacts sourced from established academic repositories and open security forums. Our research does not introduce novel malicious payloads; rather, it analyzes existing threat patterns to improve automated detection. Our procedures and datasets are shared solely for scientific reproducibility, and to enable benign users to safely analyze LLM-generated PowerShell malware. We emphasize that the generation of malicious artifacts was restricted entirely to academic research, and no production systems or uninformed third-parties were targeted. All experiments were conducted in a virtual isolated environment specifically designed for safe malware execution.

VI. Threats to Validity

In this section, we analyze threats to the validity of our study. For each threat, we detail the methodological countermeasures adopted to mitigate adverse effects on our findings.

Conclusion validity pertains to the robustness and appropriateness of the quantitative inferences drawn from our experiments. A primary concern in evaluating LLMs resides in the inherent stochasticity of both training and inference. To mitigate variations stemming from the training process, all models were fine-tuned using identical hardware infrastructure, strictly standardized environments, and identical hyperparameters, adhering to established best practices in neural network optimization. Furthermore, we mitigate variations in LLM outputs during inference by forcing a purely greedy, deterministic generation process. All inference queries were executed uniformly across the same hardware and software configuration to prevent systemic biases. Similarly, to assess the impact of model compression, the quantization procedures were systematically standardized, applying identical compression algorithms across all evaluated architectures.

As for **internal validity**, the most critical threat is data leakage, i.e., models that simply memorize payloads during pre-training rather than demonstrating genuine generalization. We rigorously addressed this by evaluating the generated malware exclusively against a strictly partitioned hold-out test

set that was never exposed to the models during the training procedures. Additionally, the risk of pre-training contamination is mathematically negligible: the training corpus consists of in-the-wild malware that we manually deobfuscated and annotated with natural language labels, thereby creating a novel data distribution unseen by public base models. A secondary threat involves the potential introduction of annotator bias and syntactic artifacts during the manual deobfuscation and labeling pipeline. To defend against this, our preprocessing methodology incorporated a strict inter-rater reliability protocol, wherein subsequent researchers systematically cross-verified the transformations and annotations by their predecessors. To further guarantee data quality, any sample that remained irrevocably obfuscated was excluded from the dataset, ensuring that models were trained on sound inputs.

As for **construct validity**, a well-documented limitation in assessing synthesized code is the mono-operation bias, where relying solely on standard Natural Language Translation metrics fails to capture semantic equivalence. Therefore, we designed a multi-dimensional evaluation framework that triangulates syntactic, structural, and dynamic behavioral characteristics. Syntactically, we decoupled our analysis from generic text metrics by adopting an ensemble of complementary code-specific evaluators. Additionally, we measured code quality, by adopting industry-grade standards analysis for PowerShell code, which provides an orthogonal perspective on the scripts beyond the similarity to reference ones. Finally, we designed a custom sandbox for dynamic analysis to capture execution behaviors, using vendor-agnostic threat detection rules from the research community, and using set-theoretic metrics to compare the triggered rules against the ground truth execution. This triangulation ensures that purely syntactic variations are not erroneously penalized if the underlying adversarial intent remains functionally intact.

Finally, with respect to **external validity**, the primary limitation of our study lies in the ecological constraints of the custom sandbox environment. We acknowledge that an isolated sandbox does not perfectly simulate the noise, latency, and comprehensive telemetry of a live enterprise network equipped with fully operational, ML-driven Endpoint Detection and Response solutions. However, this constraint is a deliberate architectural choice. The fundamental objective of this study is not to demonstrate the end-to-end evasion of production EDRs across a complete attack kill-chain, but rather to evaluate the capabilities of LLMs at synthesizing post-compromise adversarial behaviors derived from real-world samples. By standardizing the environment and focusing strictly on the behavioral footprint (via Sigma rules trigger rates) of the generated payloads compared to their real-world counterparts, our methodology establishes a rigorously controlled baseline. Consequently, while the environment is bounded, the insights regarding the models' offensive generative capabilities remain highly representative and directly applicable to the advancement of automated adversarial simulation.

VII. Related Work

In the offensive security domain, most research on LLMs focuses on the orchestration of penetration testing tools. These approaches leverage LLMs to manage external tools, by acting as planning engines and generating command-line invocations. Deng et al. [68] proposed PentestGPT, an early automated penetration testing framework based on LLMs. AutoPentester [69] is another LLM framework that implements strategic planning and self-adjustment strategies. These methods rely on existing tools to pursue offensive security; therefore, their efficacy is strictly bound by the capabilities of the underlying security tools. Our research is complementary to these efforts, as we focus on the generation of novel attack artifacts.

This work is related to previous studies on the generation of malicious code, which are focused on *prompting* LLMs, often coupled with *jailbreaking* techniques to bypass safety alignments. Çetin et al. [70] proposed a framework with a collection of prompts for generating keylogger malware, using commercial LLMs. Another example is MalGene [71], a framework based on LLMs to generate executable files starting from MITRE ATT&CK descriptions, using a divide-et-impera approach to deceive safeguards. Similarly, Sugio and Ito [72] proposed a function-oriented prompting method to generate executable malicious software. MalGEN [73] used agentic AI to decompose malicious intent into smaller inputs to several LLMs to circumvent security measures.

While these approaches demonstrate the efficacy of prompt-level manipulation, their findings remain closely coupled with the current, non-static safety filters of commercial APIs. Furthermore, their reliance on commercial, black-box interfaces obscures the underlying generative potential of models themselves, and, more importantly, this focus diverges from the strategic paradigms of a structured APT. Unlike opportunistic attackers, nation-state actors engaged in espionage likely leverage unfiltered, self-hosted environments to orchestrate custom malware required for persistent operations, a dimension currently not covered by previous work.

Our work leverages domain adaptation to unlock specialized generative capabilities. This approach aligns with the operational reality of sophisticated APTs, which likely eschew commercial APIs in favor of smaller, locally-trained models to achieve deterministic, high-fidelity generation of malicious intent without external dependencies. The study of El Zemity et al. [74] proposed a dataset of pseudo-malicious data, with the purpose of evaluating safeguards of open-weight LLMs. Another example is the study by Xu, Gardier, and Belguith [75], which investigated fine-tuning attacks on pre-trained CoT models, using data gathered by Sheshadri et al. [76]. Bao et al. [77] investigated the generation of synthetic malware using GANs and diffusion models.

In summary, while the current state-of-the-art has matured in terms of tool orchestration and prompt-based manipulation, there remains a significant research gap regarding the efficiency of APT-suitable, domain-adapted LLMs. Unlike prior work, we propose a specialized evaluation framework for LLM-generated malware that leverages real-world samples to uncover the models' true generative capabilities.

VIII. Conclusion

In this work, we presented PSStrikes, a dataset of wild PowerShell malware annotated with human-labeled descrip-

tions developed to assess LLMs' capabilities in generating PowerShell malware. We further described our novel evaluation framework to analyze AI-generated PowerShell malware. The evaluation process, encompassing three stages, included a novel dynamic analysis framework backed by our further contribution: PSSandman, a sandbox specifically designed to evaluate LLM-generated PowerShell malware. Finally, we showcased the application of our contributions by conducting an empirical analysis using open-weight LLMs trained with our data and then assessed with the proposed framework. Our approach provided a broad view of the chosen model's capabilities in generating PowerShell malware, highlighting the strengths and limitations of chosen LLMs in different training methodologies and inference settings. By introducing a new dataset of real-world samples and a customized evaluation framework, we offer a practical way to analyze how LLMs can be used for malicious purposes. This setup allows the community to better understand and counteract these emerging threats. Our experimental study findings unveil the threat of possible malicious campaigns leveraging AI-generated malware. In particular, the results reveal the LLM's capability, equipped with their generalization abilities, in adhering to the behavior of real-world malware.

REFERENCES

[1] "Staying ahead of threat actors in the age of AI," 2024. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2024/02/14/staying-ahead-of-threat-actors-in-the-age-of-ai/

[2] OpenAI, "Disrupting malicious uses of AI." [Online]. Available: https://openai.com/it-IT/global-affairs/disrupting-malicious-uses-of-ai-october-2025/

[3] A. Moix, L. Ken, and Jacob Klein, "Threat Intelligence Report: August 2025," 2025.

[4] ESET Research, "The PromptLock malware," https://infosec.exchange/@ESETresearch/115095803130379945.

[5] Computer Emergency Response Team of Ukraine, "UAC-0001 cyberattacks on the security and defense sector using LAMEHUG software using LLM," 2025.

[6] MITRE, "APT28." [Online]. Available: https://attack.mitre.org/groups/G0007/

[7] "Qwen2.5-Coder Collection Repository." [Online]. Available: https://huggingface.co/collections/Qwen/qwen25-coder

[8] R. O. Zhen Heng, "PROMPTFLUX malware," https://promptintel.novahunting.ai/prompt/a816c8b5-5188-4a9c-bd9a-1e6cbcd47686.

[9] GoSecure, "Talk To Your Malware - Integrating AI Capability in an Open-Source C2 Agent," 2025. [Online]. Available: https://www.gosecure.ai/blog/talk-to-your-malware-integrating-ai-capability-in-an-open-source-c2-agent

[10] CrowdStrike, "2026 Global Threat Report," CrowdStrike, Tech. Rep., 2026. [Online]. Available: https://www.crowdstrike.com/explore/2026-global-threat-report/2026-global-threat-report

[11] red canary, "Threat Detection Report." [Online]. Available: https://redcanary.com/threat-detection-report/techniques/powershell/

[12] J. Kutscher, "M-Trends 2025: Special Report," Google Cloud, Mandiant, Tech. Rep., 2025. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/m-trends-2025

[13] M. Antonakakis, T. April, M. . Bailey *et al.*, "Understanding the mirai botnet," in *USENIX Security Symposium*, 2017.

[14] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, 2014.

[15] S. Yan, S. Wang, Y. Duan, H. Hong, K. Lee, D. Kim, and Y. Hong, "An LLM-Assisted Easy-to-Trigger Backdoor Attack on Code Completion Models: Injecting Disguised Vulnerabilities against Strong Detection," in *USENIX Security Symposium*, 2024.

[16] Y. Huang, X. Jia, W. Guo, Y. Sun, Y. Huang, C. Wang, and Y. Liu, "Casting a SPELL: Sentence Pairing Exploration for LLM Limitation-breaking," *arXiv preprint arXiv:2512.21236*, 2025.

[17] P. Liguori, C. Marescalco, R. Natella, V. Orbinato, and L. Pianese, "The Power of Words: Generating PowerShell Attacks from Natural Language," in *USENIX WOOT Conf. on Offensive Technologies*, 2024.

[18] A.-M. Apostu, A. Preda, A. D. Damir, D. Bolocan, R. T. Ionescu, I. Croitoru, and M. Gaman, "AutoMalDesc: Large-Scale Script Analysis for Cyber Threat Research," in *AAAI Conf. on Artificial Intelligence*, vol. 40, no. 1, 2026.

[19] R. Bessa, R. Claro, J. Trindade, and J. Lourenço, "RedShell: A Generative AI-Based Approach to Ethical Hacking," *arXiv preprint arXiv:2604.11506*, 2026.

[20] "PowerShell-for-Hackers." [Online]. Available: https://github.com/I-Am-Jakoby/PowerShell-for-Hackers

[21] "Powershell-Scripts-for-Hackers-and-Pentesters." [Online]. Available: https://github.com/Whitecat18/Powershell-Scripts-for-Hackers-and-Pentesters

[22] "Malicious-PowerShell-Dataset." [Online]. Available: https://github.com/Fa2y/Malicious-PowerShell-Dataset

[23] ABUSE.ch, "MalwareBazaar," https://bazaar.abuse.ch/.

[24] MITRE, "HybridAnalysis," https://hybrid-analysis.com/.

[25] "Nishang." [Online]. Available: https://github.com/samratashok/nishang

[26] "PowerSploit." [Online]. Available: https://github.com/PowerShellMafia/PowerSploit

[27] H. Chai, L. Ying, H. Duan, and D. Zha, "Invoke-deobfuscation: AST-based and semantics-preserving deobfuscation for PowerShell scripts," in *IEEE/IFIP Intl. Conf. on Dep. Systems and Networks (DSN)*, 2022.

[28] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, 2024.

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

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

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

[32] "Invoke-obfuscation v1.8." [Online]. Available: https://github.com/danielbohannon/Invoke-obfuscation

[33] A. Creswell, M. Shanahan, and I. Higgins, "Selection-inference: Exploiting large language models for interpretable logical reasoning," *arXiv preprint arXiv:2205.09712*, 2022.

[34] "Revoke-Obfuscation v1.0." [Online]. Available: https://github.com/danielbohannon/Revoke-Obfuscation

[35] "PowerDecode." [Online]. Available: https://github.com/Malandrone/PowerDecode

[36] G. M. Malandrone, G. Virdis, G. Giacinto, D. Maiorca *et al.*, "Powerdecode: a powershell script decoder dedicated to malware analysis," in *Italian Conf. on Cybersecurity, ITASEC*, vol. 2940, 2021.

[37] M. Evtikhiev, E. Bogomolov, Y. Sokolov, and T. Bryksin, "Out of the bleu: how should we assess quality of the code generation models?" *Elsevier: Journal of Systems and Software*, vol. 203, 2023.

[38] M. Popović, "chrF: character n-gram F-score for automatic MT evaluation," in *Wksp. on Statistical Machine Translation*, 2015.

[39] M. Denkowski and A. Lavie, "Meteor universal: Language specific translation evaluation for any target language," in *Wksp. on Statistical Machine Translation*, 2014.

[40] A. Eghbali and M. Pradel, "CrystalBLEU: precisely and efficiently measuring the similarity of code," in *IEEE/ACM Intl. Conf. on Automated Soft. Eng.*, 2022.

[41] "Powershell SDK: Parser Class." [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.management.automation.language.parser?view=powershellsdk-7.4.0

[42] "PSScriptAnalyzer." [Online]. Available: https://github.com/powershell/psscriptanalyzer

[43] B. Payette and S. Richard, *Windows PowerShell in Action*. Manning Publications, 2018.

[44] M. Shepard, C. Venkatesan, S. Talaat, and B. J. Blawat, *PowerShell: Automating Administrative Tasks: The art of automating and managing Windows environments*. Packt Publications, 2017.

[45] C. Dent, *Mastering PowerShell Scripting: Automate repetitive tasks and simplify complex administrative tasks using PowerShell*. Packt Publications, 2024.

[46] "VirtualBox Technical Documentation." [Online]. Available: https://www.virtualbox.org/manual/ch08.html

[47] "Benjamin Delpy's implementation of Mimikatz." [Online]. Available: https://github.com/gentilkiwi/mimikatz

[48] "PowerSploit." [Online]. Available: https://attack.mitre.org/software/S0194/

[49] "MITRE ATT&CK Enterprise Techniques." [Online]. Available: https://attack.mitre.org/versions/v18//techniques/enterprise/

[50] "System Monitor (Sysmon)." [Online]. Available: https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon

[51] M. Roddie, J. Deyalsingh, and G. J. Katz, *Practical Threat Detection Engineering: A hands-on guide to planning, developing, and validating detection capabilitie*. Packt Publications, 2023.

[52] "Sigma - Generic Signature Format for SIEM Systems." [Online]. Available: https://github.com/sigmahq/sigma

[53] "Zircolite." [Online]. Available: https://github.com/wagga40/Zircolite

[54] "MITRE ATT&CK Matrix for Enterprise, version 18.1." [Online]. Available: https://attack.mitre.org/versions/v18/

[55] "Impair Defenses: Disable or Modify Tools." [Online]. Available: https://attack.mitre.org/versions/v18/techniques/T1562/001/

[56] K. Dembczyński, W. Waegeman, W. Cheng, and E. Hüllermeier, "On label dependence and loss minimization in multi-label classification," *Machine Learning*, vol. 88, no. 1, 2012.

[57] G. Tsoumakas and I. Katakis, "Multi-label classification: An overview," *Intl. J. of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, 2007.

[58] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bull Soc Vaudoise Sci Nat*, 1901.

[59] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[60] L. R. Dice, "Measures of the amount of ecologic association between species," *JSTOR Ecology*, 1945.

[61] T. Sφrensen, "A method of establishing groups of equal amplitude in plant sociologybased on similarity of species content and its application to analyses of the vegettation on Danish commons," *Biol. Skr., K. danske Vidensk*, vol. 5, 1948.

[62] A. Young, B. Chen, C. Li, C. Huang, G. Zhang, G. Zhang, G. Wang, H. Li, J. Zhu, J. Chen *et al.*, "Yi: Open foundation models by 01. ai," *arXiv preprint arXiv:2403.04652*, 2024.

[63] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang *et al.*, "Qwen2.5-Coder Technical Report," *arXiv preprint arXiv:2409.12186*, 2024.

[64] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[65] S. Joel, J. Wu, and F. Fard, "A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages," *ACM Transactions on Software Engineering and Methodology*, 2024.

[66] M. Bailey, D. A. Dittrich, E. Kenneally, and D. Maughan, "The Menlo Report," *IEEE Security & Privacy*, vol. 10, no. 2, 2012.

[67] E. Kenneally and D. Dittrich, "The menlo report: Ethical principles guiding information and communication technology research," *Available at SSRN 2445102*, 2012.

[68] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing," in *USENIX Security Symposium*, 2024.

[69] Y. Ginige, A. Niroshan, S. Jain, and S. Seneviratne, "Autopentester: An LLM agent-based framework for automated pentesting," in *IEEE Intl. Conf. on Trust, Security and Privacy in Comp. and Comm. (TrustCom)*, 2025.

[70] O. Çetin, B. Birinci, Ç. Uysal, and B. Arief, "Exploring the cybercrime potential of llms: A focus on phishing and malware generation," in *Europ. Interdisciplinary Cybersecurity Conf.*, 2025.

[71] Y.-J. Lin, P.-H. Chou, W.-Y. Shen, Y.-R. Guo, C.-L. Wu, and Y.-T. Huang, "Code as a Weapon: Generating Malware with Large Language Models," in *IEEE Conf. on Dep. and Sec. Comp. (DSC)*, 2025.

[72] N. Sugio and H. Ito, "Jailbreak-Free LLM-Assisted Malware Generation: An Empirical and Conceptual Analysis," *IEEE Access*, 2026.

[73] B. Saha and S. K. Shukla, "Malgen: A generative agent framework for modeling malicious software in cybersecurity," *arXiv preprint arXiv:2506.07586*, 2025.

[74] A. ElZemity, B. Arief, and S. Li, "CyberLLMInstruct: A Pseudo-Malicious Dataset Revealing Safety-Performance Trade-offs in Cyber Security LLM Fine-tuning," in *ACM Wksp. on Artificial Intelligence and Security*, 2025.

[75] Z. Xuan, J. Gardiner, and S. Belguith, "The dark deep side of DeepSeek: Fine-tuning attacks against the safety alignment of CoT-enabled models," *arXiv preprint arXiv:2502.01225*, 2025.

[76] A. Sheshadri, A. Ewart, P. . Guo *et al.*, "Latent adversarial training improves robustness to persistent harmful behaviors in llms," *arXiv preprint arXiv:2407.15549*, 2024.

[77] T. Bao, K. Trousil, Q. D. Tran, F. Di Troia, and Y. Park, "Generating Synthetic Malware Samples using Generative AI against Zero-day Attacks," *IEEE Access*, 2025.