QUT-DV25: A Dataset for Dynamic Analysis of Next-Gen Software Supply Chain Attacks

Sk Tanzir Mehedi
QUT, Brisbane, Australia, QLD 4000
tanzir.mehedi@hdr.qut.edu.au

Raja Jurdak
QUT, Brisbane, Australia, QLD 4000
r.jurdak@qut.edu.au

Chadni Islam
ECU, Joondalup, Australia, WA 6027
c.islam@ecu.edu.au

Gowri Ramachandran
QUT, Brisbane, Australia, QLD 4000
g.ramachandran@qut.edu.au

Abstract

Securing software supply chains is a growing challenge due to the inadequacy of existing datasets in capturing the complexity of next-gen attacks, such as multiphase malware execution, remote access activation, and dynamic payload generation. Existing datasets, which rely on metadata inspection and static code analysis, are inadequate for detecting such attacks. This creates a critical gap because these datasets do not capture what happens during and after a package is installed. To address this gap, we present QUT-DV25, a dynamic analysis dataset specifically designed to support and advance research on detecting and mitigating supply chain attacks within the Python Package Index (PyPI) ecosystem. This dataset captures install and post-install-time traces from 14,271 Python packages, of which 7,127 are malicious. The packages are executed in an isolated sandbox environment using an extended Berkeley Packet Filter (eBPF) kernel and user-level probes. It captures 36 real-time features, that includes system calls, network traffic, resource usages, directory access patterns, dependency logs, and installation behaviors, enabling the study of next-gen attack vectors. ML analysis using the `QUT-DV25` dataset identified four malicious PyPI packages previously labeled as benign, each with thousands of downloads. These packages deployed covert remote access and multi-phase payloads, were reported to PyPI maintainers, and subsequently removed. This highlights the practical value of `QUT-DV25`, as it outperforms reactive, metadata, and static datasets, offering a robust foundation for developing and benchmarking advanced threat detection within the evolving software supply chain ecosystem.

1 Introduction

The exponential growth of Open-Source Software (OSS) has introduced significant cybersecurity challenges, particularly in detecting sophisticated software supply chain attacks targeting ecosystems like PyPI [1, 2]. PyPI hosts over 620,000 packages and facilitates billions of daily downloads, underscoring its central role in modern software development [3, 4]. However, its open nature and rapid scalability have made it a prime target for adversaries [5]. As of July 2024, 1.2% of total PyPI packages have been identified as malicious, highlighting growing security concerns [6, 7, 8, 9]. These attacks are becoming more common as multi-stage threats that exploit vulnerabilities in OSS, including typosquatting, malware execution, remote access, and dynamic payload generation [10, 11, 12]. Such threats compromise the core security principles of confidentiality, integrity, and availability [2, 13]. Existing defense mechanisms, such as host-based firewalls and signature or rule-based malware scanners, struggle to counter these threats due to their inability to adapt to evolving, multi-stage adversarial tactics and their limited capacity for granular behavioral code analysis [14, 15, 16]. As a result, Malicious Detection Systems (MDS) have emerged as critical safeguards for the OSS ecosystem, using ML methods to identify and mitigate next-gen attacks [17, 18, 19].

Preprint.

The effectiveness of MDSs relies on their detection performance metrics, which require evaluation against comprehensive datasets containing both benign and malicious package behaviors [17, 5]. Widely adopted metadata and static dataset benchmarks, such as the PyPI Malware Registry [20], Backstabber's Knife Collection [21], DataDog [22], and PyPIGuard [23], have served as standards for MDS validation. However, recent studies highlight critical limitations in these datasets [5, 24]. For instance, metadata datasets capture only package details like descriptions and author profiles, while static datasets focus on code attributes such as function signatures and import statements, without executing packages during installation or post-installation [17, 5]. This omission severely limits their ability to detect dynamic threats such as typosquatting, remote access activation, and install-time-specific payload generation [14, 15, 16]. Hybrid datasets, which combine static and metadata features, partially address some threats but still fail to detect most of these threats, as they lack visibility into complex behaviors that occur during install-time and post-install-time [19, 24]. These limitations in existing datasets undermine the reliability of performance metrics, raising concerns about the generalizability of MDS evaluations to next-gen OSS supply chain attacks.

To address these challenges, this study introduces the `QUT-DV25` dataset, specifically designed to facilitate dynamic analysis of next-gen OSS software supply chain attacks within the PyPI ecosystem. `QUT-DV25` captures behavioral traces from 14,271 Python packages, 7,127 of which exhibit malicious behavior, through install and post-install-time in a controlled, isolated sandbox environment using an extended Berkeley Packet Filter (eBPF) kernel and user-level probes. This tool enables real-time tracing without kernel modification and supports flexible, programmable monitoring in C or Python [25]. The dataset records 36 real-time features for each package, including system calls, network traffic, resource consumption, directory access patterns, dependency resolution, and installation behaviors. These features enable comprehensive analysis of dynamic attack vectors such as multiphase malware execution, remote access activation, and post-install-time-specific payload generation. A detailed characterization of the dataset's structure, threat coverage, and example is also provided. Furthermore, the performance of MDSs is evaluated based on this dataset as a binary classification task using multiple supervised ML methods. By bridging the gap between static and dynamic analysis, `QUT-DV25` empowers cybersecurity researchers to develop robust defenses against evolving OSS supply chain threats. The dataset and code are publicly available on `QUT-DV25`[1], ensuring reproducibility and enabling further research. The key contributions of this study include:

- A controlled, isolated testbed framework to generate and collect datasets by installing and executing packages and capturing behaviors such as typosquatting, dynamic payload generation, and multiphase malware execution.
- `QUT-DV25`, a dataset of 14,271 packages, including 7,127 malicious ones, with 36 features across six categories that capture install-time and post-install-time behaviors previously unexplored for malicious package detection.
- A first-hand evaluation of four popular machine learning methods on the proposed dataset is also provided as a baseline for further research.

This study is organized as follows: Section 2 reviews existing datasets. Section 3 outlines the dataset construction and details of the dataset. Section 4 presents technical validation and benchmarks. Section 5 discusses limitations and usage examples, and Section 6 addresses safety and ethical considerations. Finally, Section 7 concludes the study and outlines future directions.

2 Existing Datasets

The effectiveness of MDS datasets depends on two factors: coverage of modern threats and diversity of benign behaviors to reduce false positives [5]. Datasets lacking realistic adversarial contexts risk misleading evaluations [24]. Existing benchmarks are categorized as metadata, static, hybrid, and dynamic, each with limitations in modeling next-gen multi-stage attacks.

Metadata datasets: Metadata datasets focus on static, non-execution-based features such as package names, descriptions, version histories, and author profiles [17, 26, 27]. These datasets are widely adopted in MDSs due to their computational efficiency and ease of analysis, enabling rapid screening of large package repositories [17]. For example, Guo et al. [20] proposed the PyPI Malware Registry dataset, and Marc et al. [21] proposed the Backstabber's Knife Collection dataset to flag suspicious packages based on anomalies such as mismatched author credentials or irregular update patterns. However, metadata datasets fail to capture dynamic behaviors-such as system interactions during

[1]Dataset: `https://doi.org/10.7910/DVN/LBMXJY` and package list: `https://qut-dv25.dysec.io`

install-time or post-install-time-that are critical for detecting sophisticated threats like typosquatting, multiphase malware execution, and dynamic payload generation [14, 15, 16]. Attackers can exploit this limitation by crafting packages with plausible metadata while embedding malicious logic that activates post-deployment [24]. Consequently, MDSs relying solely on metadata suffer from high false positive rates, as benign packages with irregular metadata are misclassified, and malicious ones evade detection through metadata obfuscation [18].

Static datasets: Static datasets analyze code attributes such as function signatures, import statements, and control flow structures to identify malicious patterns without executing packages [5, 18, 28]. Datasets such as DataDog [22], developed by Datadog Security Labs, detect known threats-including hardcoded backdoors and command-and-control (C2) functionalities, matching code artifacts against curated malicious signature patterns. These datasets excel at identifying obvious malicious code and are computationally lightweight, making them scalable for large-scale repository scans [18, 28]. Static analysis, however, cannot detect install-time and post-install-time-specific threats such as multi-stage malware or environment-triggered malicious behavior [5]. For instance, a package with innocuous static code may execute a hidden script during installation to exfiltrate sensitive data scenario invisible to static inspection [29, 30, 14]. Additionally, techniques like code obfuscation or encryption easily bypass static detection, as they mask malicious intent until installation [31].

Hybrid datasets: Hybrid datasets integrate metadata and static code features to balance efficiency and depth, aiming to detect threats that evade single-mode analysis [19, 24]. For example, Samaana et al. [5] developed a hybrid dataset by combining metadata attributes with static code features to improve malicious package detection. Similarly, the PyPIGuard dataset, proposed by Iqbal et al. [23], combines package metadata with static code attributes to flag packages that may appear benign in isolation but exhibit suspicious patterns when analyzed holistically. This dataset improves detection of contextual threats, such as dependency confusion attacks, where malicious packages mimic legitimate names but contain altered code [24]. Despite their advantages, hybrid datasets remain limited by their lack of install-time behavioral data. For instance, they cannot model indirect dependency hijacking or post-deployment behaviors [14]. Advanced threats like polymorphic malware, which alters its code or behavior based on environmental cues, further evade hybrid detection due to the absence of dynamic execution traces [32]. These gaps undermine hybrid datasets' ability to address multi-stage attacks, where malicious activity unfolds progressively across install-time and post-install-time phases. Table 1 presents the existing datasets for detecting malicious packages in PyPI.

eBPF-based dynamic datasets: eBPF offers a powerful framework for real-time system monitoring, providing fine-grained visibility into install-time and post-install-time behaviors with low overhead [33, 25]. This capability has been used in security applications, such as ransomware detection through system call trace analysis and network anomaly identification [34, 35]. However, existing eBPF-based implementations predominantly rely on rule-based threat detection methodologies. For instance, Higuchi and Kobayashi [36] developed a ransomware detection system using eBPF-traced system call patterns, while Zhuravchak and Dudykevych [34] employed predefined behavioral rules for real-time ransomware analysis. Such rule-based approaches, though effective for known threats, lack adaptability to zero-day attacks due to their dependence on static signatures.

To enhance flexibility, tools like `bpftrace`, `bpftool`, and `bcc-tools` extend eBPF's utility by enabling dynamic tracing of low-level kernel and user-space events without kernel modifications [33]. These tools support programmable tracing in C or Python, facilitating the extraction of behavioral signals such as system calls, network traffic, resource consumption, and directory access patterns [25]. While these traces provide a foundation for behavioral analysis, current ML-based MDS often lack datasets that capture such dynamic, real-time system-level behaviors.

Table 1: Existing datasets for PyPI malicious package detection.

Datasets	Detect manipulate metadata	Detect encoding technique	Dynamic payload generation	Detect typo-squatting	Remote access activation	Detect indirect dependencies
Metadata [20]	◐	○	○	◐	○	○
Static [22]	●	◐	○	◐	◐	○
Hybrid [23]	●	◐	○	◐	◐	○
QUT-DV25	●	●	●	●	●	●

Note: Malicious package detection - ● possible, ◐ partially possible, ○ not possible.

3 QUT-DV25 Dataset Construction

In this section, we first discuss the testbed configuration, followed by the dataset collection methodology and an overview of the dataset, including feature sets.

3.1 Testbed Configuration

The experimental testbed setup involves 16 Raspberry Pi devices $\mathcal{D}_{\text{RPi}} = \{d_k\}_{k=1}^{16}$, each running Ubuntu 24.4 LTS with Python 3.8-3.12 in isolated virtualized environments. A private network $\mathcal{N}_{\text{priv}} = \{\text{Router}, \text{Switch}, \text{Raspberry Pi}\}$ ensures secure traffic flow. Behavioral monitoring is implemented using eBPF integrated into the Linux kernel $\mathcal{K} = \text{v6.8.0-1012-raspi}$, with real-time tracing tools $\mathcal{T}_{\text{bcc}} = \{\text{bcc-tools}, \text{bpftool}, \text{bpftrace}\}$. Figure 1 shows a visualization of the isolated testbed configuration. To validate and scale the resulting dataset for ML, a high-performance computing cluster $\mathcal{C}_{\text{HPC}} = \{c_k\}_{k=1}^{m}$ is employed, where each node features 16-core CPUs, NVIDIA A100 GPUs, and 128 GB of RAM.

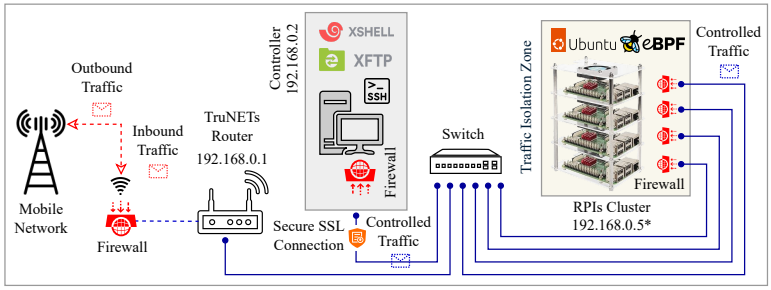

Figure 1: The isolated testbed configuration visualization for QUT-DV25.

3.2 Collection Methodology

We propose the QUT-DV25 Dataset Framework, a structured methodology for constructing a dataset that captures both install-time and post-install-time behaviors of software packages. This framework is designed to meet the growing need for dynamic datasets in detecting multi-stage, next-gen software supply chain attacks, particularly within ecosystems such as PyPI. The framework systematically integrates three phases: (i) dataset collection, (ii) labeling and validation, and (iii) trace extraction, as illustrated in Figure 2.

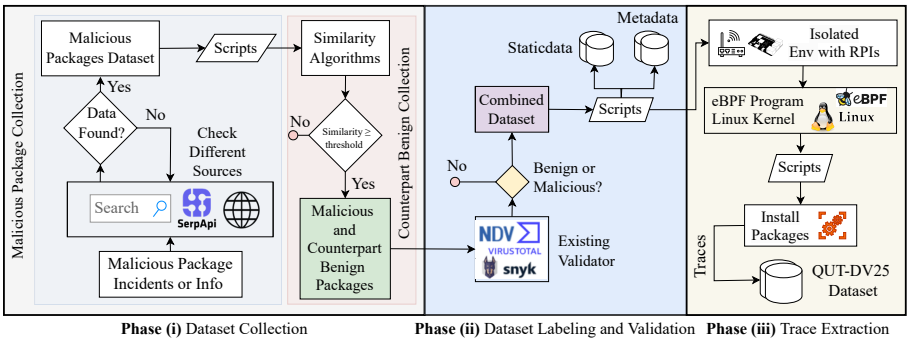

Figure 2: The overall framework for collecting the QUT-DV25 dataset.

Dataset collection: In the absence of a centralized repository of malicious PyPI packages, we collect data from multiple threat intelligence sources denoted by $\mathcal{S} = \{S_1, \ldots, S_K\}$, where each S_k corresponds to a source such as GitHub advisories or malware databases, and K is the total number of such sources. The combined set of malicious packages is defined as $\mathcal{M} = \bigcup_{k=1}^{K} S_k = \{(n_m^i, v_m^i)\}_{i=1}^{N}$, where n_m^i and v_m^i denote the name and version of the i-th malicious package and N is the total number of malicious samples collected. To enable comparative analysis and

support downstream classification tasks, we extract benign counterparts by defining the universe of benign packages as $\mathcal{U} = \{(n_b^j, v_b^j)\}_{j=1}^{M}$, where n_b^j and v_b^j denote the name and version of the j-th benign package, and M is the total number of benign packages. We apply similarity algorithms $\mathrm{sim}(n_m^i, n_b^j) \in [0,1]$ to compute name-based similarity between malicious and benign packages. For each malicious package, we form a candidate set $\mathcal{C}^i = \{(n_b^j, v_b^j) \in \mathcal{U} \mid \mathrm{sim}(n_m^i, n_b^j) \geq \tau\}$, where $\tau \in [0,1]$ is a similarity threshold. If $\mathcal{C}^i \neq \emptyset$, we select the most similar benign package $(n_b^{*i}, v_b^{*i}) = \arg\max_{(n_b, v_b) \in \mathcal{C}^i} \mathrm{sim}(n_m^i, n_b)$ and fetch the release date r_b^{*i}. The final similarity score is recorded as $s^{*i} = \mathrm{sim}(n_m^i, n_b^{*i})$. Then, construct the final malicious-benign dataset $\mathcal{D}_{\text{mb}} = \{(n_m^i, v_m^i, n_b^{*i}, v_b^{*i}, r_b^{*i}, s^{*i}) \mid s^{*i} \geq \tau\}$ which serves as input for validation and dynamic trace extraction phases. Algorithm 1 outlines the process for retrieving counterpart benign packages.

Algorithm 1: Lexical Similarity-Based Benign Package Retrieval

Input: Malicious set $\mathcal{M} = \{(n_m^i, v_m^i)\}_{i=1}^{N}$; benign universe $\mathcal{U} = \{(n_b^j, v_b^j)\}_{j=1}^{M}$; threshold $\tau \in [0,1]$
Output: Malicious-benign dataset $\mathcal{D}_{\text{mb}} = \{(n_m^i, v_m^i, n_b^{*i}, v_b^{*i}, r_b^{*i}, s^{*i})\}$
Precondition: Similarity metric; PyPI API accessible
foreach $(n_m^i, v_m^i) \in \mathcal{M}$ **do**
  $\mathcal{C}^i \leftarrow \{(n_b^j, v_b^j, s_{ij}) \mid (n_b^j, v_b^j) \in \mathcal{U},\ s_{ij} = \mathrm{sim}(n_m^i, n_b^j) \geq \tau\}$
  if $\mathcal{C}^i \neq \emptyset$ **then**
    $(n_b^{*i}, v_b^{*i}, s^{*i}) \leftarrow \arg\max \mathcal{C}^i$
    $r_b^{*i} \leftarrow$ `QueryPyPI`(n_b^{*i})
    $\mathcal{D}_{\text{mb}} \leftarrow \mathcal{D}_{\text{mb}} \cup \{(n_m^i, v_m^i, n_b^{*i}, v_b^{*i}, r_b^{*i}, s^{*i})\}$
  end
end
Export $\mathcal{D}_{\text{mb}}$ to file
Postcondition: Only pairs with $s^{*i} \geq \tau$ retained

Dataset labeling and validation: For each package $(n, v) \in \mathcal{D}_{\text{mb}}$, where n and v denote the package name and version respectively, a set of external threat intelligence validators $\mathcal{V} = \{\mathrm{VirusTotal}, \mathrm{NDV}, \mathrm{Snyk}\}$ is queried. Each validator returns a label $\mathrm{label}_k(n, v) \in \{0, 1, \perp\}$, indicating whether the package is malicious (1), benign (0), or inconclusive ($\perp$). The final label is determined as follows: $\mathrm{Label}(n, v) = 1$ (malicious) if at least two validators return 1, i.e., $\sum_{k \in \mathcal{V}} \mathbb{I}[\mathrm{label}_k(n, v) = 1] \geq 2$; $\mathrm{Label}(n, v) = 0$ (benign) only if all validators agree the package is benign, i.e., $\sum_{k \in \mathcal{V}} \mathbb{I}[\mathrm{label}_k(n, v) = 0] = |\mathcal{V}|$. If neither condition holds, due to inconclusive results, the label is assigned through manual inspection: $\mathrm{Label}(n, v) = \mathrm{ManualInspect}(n, v)$. The validated dataset is defined as $\mathcal{D}_{\text{valid}} = \{(n, v, \mathrm{Label}(n, v)) \mid (n, v) \in \mathcal{D}_{\text{mb}}\}$.

In parallel, metadata features $\mathcal{X}(n, v)$ (e.g., author, version history, description) and static features $\mathcal{Y}(n, v)$ (e.g., import statements, function definitions) are extracted for each package. These are combined to construct the final labeled dataset $\mathcal{D}_{\text{final}} = \{(n, v, \mathrm{Label}(n, v), \mathcal{X}(n, v), \mathcal{Y}(n, v)) \mid (n, v) \in \mathcal{D}_{\text{valid}}\}$. The dataset $\mathcal{D}_{\text{valid}}$ serves as the input to the next trace extraction step, while $\mathcal{D}_{\text{final}}$ is used as a benchmark for evaluating existing MDS methods.

Trace extraction: The validated package set $\mathcal{D}_{\text{valid}} = \{(n_j, v_j, \mathrm{Label}(n_j, v_j))\}_{j=1}^{m}$ serves as input for this phase. Each package archive is denoted by $\pi_j = (n_j, v_j) \in \mathcal{D}_{\text{valid}}$, and it is deployed to a uniformly random device $d_k = f(\pi_j)$ from the set of Raspberry Pi devices $\mathcal{D}_{\text{RPi}} = \{d_k\}_{k=1}^{n}$. Two binary indicators are defined: $\mathrm{Deploy}(\pi_j, d_k)$ and $\mathrm{Install}(\pi_j, d_k)$, which take the value 1 if the transfer and installation of π_j on d_k succeed, respectively. In cases where $\mathrm{Install}(\pi_j, d_k) = 0$, partial installation of dependencies is occasionally observed. These cases form a subset $\mathcal{D}_{\text{partial}} \subset \mathcal{D}_{\text{valid}}$ and are of particular interest, as malicious payloads may persist through successfully installed subcomponents. Additionally, dependency resolution may implicitly introduce malicious variants: a benign package version v_j of n_j may cause the installation of a related version v_j' such that $(n_j, v_j') \in \mathcal{D}_{\text{valid}}$ and $\mathrm{Label}(n_j, v_j') = \text{Malicious}$. To account for such behavioral variability, these cases are retained for analysis.

During the install-time and post-install-time of these packages, eBPF captures kernel and user-level event sequences. post-install-time tracing executes for a fixed duration $\Delta = 120\,\mathrm{s}$, producing a sequence $\mathrm{Trace}(\pi_j, d_k) \in \mathcal{S}^*$, where $\mathcal{S}^*$ denotes the space of trace sequences. The trace extraction function $\mathrm{Extract}(\pi_j, d_k)$ yields $\mathrm{Trace}(\pi_j, d_k)$ if both deployment and installation succeed, i.e.,

Algorithm 2: `QUT-DV25` Dynamic Trace Extraction

Input: Validated dataset $\mathcal{D}_{\text{valid}} = \{(n, v, \text{Label}(n, v))\}_{j=1}^{m}$; Raspberry Pi devices $\mathcal{D}_{\text{RPi}} = \{d_k\}_{k=1}^{n}$
Output: Traces $\mathcal{T} = \{T_j\}_{j=1}^{m}$
Precondition: Each d_k runs an isolated Python 3.8–3.12 environment with eBPF support.
Definitions:
    $f : \mathcal{D}_{\text{valid}} \to \mathcal{D}_{\text{RPi}}$ (uniform random device assignment)
    $\text{Deploy}(\pi_j, d_k) = 1$ iff package π_j is successfully transferred to d_k
    $\text{Install}(\pi_j, d_k) = 1$ iff package π_j installs successfully on d_k
    $\text{Trace}(\pi_j, d_k) \in \mathcal{S}^*$ captures the eBPF event sequence during install-time and post-install-time (120s)
    $\text{Extract}(\pi_j, d_k) = \text{Trace}(\pi_j, d_k)$ if $\text{Deploy}(\pi_j, d_k) = 1 \wedge \text{Install}(\pi_j, d_k) = 1$, else $\emptyset$
for $j \leftarrow 1$ **to** m **do**
    $d_k \leftarrow f(\pi_j)$
    if $\text{Deploy}(\pi_j, d_k) = 0$ ***or*** $\text{Install}(\pi_j, d_k) = 0$ **then**
        $T_j \leftarrow \emptyset$
        continue
    end
    $T_j \leftarrow \text{Trace}(\pi_j, d_k)$
end
return $\mathcal{T} = \{T_j\}_{j=1}^{m}$

$\text{Deploy}(\pi_j, d_k) = \text{Install}(\pi_j, d_k) = 1$; otherwise, it returns the empty set $\emptyset$. If $\text{Extract}(\pi_j, d_k) = \emptyset$, the environment on d_k is reset, and the process repeats with the same package until a valid trace is collected. The resulting valid trace is denoted $T_j = \text{Trace}(\pi_j, d_k)$, and the complete trace set is given by $\mathcal{T} = \{T_j\}_{j=1}^{m}$. This trace set provides isolated and reproducible dynamic behavioral profiles for each package, supporting subsequent analysis (cf. Algorithm 2).

3.3 QUT-DV25 Data Records

This study analyzes a corpus of $|\mathcal{D}_{\text{valid}}| = 14,271$ Python packages, of which $7,127$ are labeled as malicious. Approximately 88% of these packages yielded successful installations, i.e., $\text{Install}(\pi_j, d_k) = 1$, while the remaining packages triggered direct install-time anomalies, such as system crashes, infinite loops, forced shutdowns, or authentication prompts-despite $\text{Install}(\pi_j, d_k) = 0$. To characterize behavioral variability across the dataset, a classification function $\text{Classify}(\pi_j) \in \mathcal{B} = \{\text{normal}, \text{compatibility}, \text{system}\}$ is introduced, mapping each package π_j to a behavioral category based on its install-time and post-install-time outcomes in the isolated environment. Table 2 summarizes the characteristics of packages during install-time and post-install-time analysis.

Table 2: Characteristics of packages during install-time and post-install-time analysis.

Install-time and post-install-time characteristics	**Malicious**	**Benign**
Normal: Successfully installed, metadata issues, setup/wheel issues	6,864	6,905
Compatibility: Mismatch, version issues, auth, naming, module issues	236	202
System: Freezing, infinity waiting, looping, shutdown, prerequisites	27	37
Total:	**7,127**	**7,144**

Feature sets and annotations: To analyze install-time and post-install-time behaviors comprehensively, the system is instrumented using eBPF-based monitoring, which enables real-time capture of both kernel-space and user-space activity for each execution trace $T_j \in \mathcal{T}$. The feature set for each trace is denoted as $\mathcal{T}_j = \{F_j^{(i)}\}_{i=1}^{q}$, where each $F_j^{(i)}$ corresponds to a category of behavioral signals derived from eBPF probes. The observed traces are categorized into six primary trace types $\mathcal{F}$, mapping the raw trace T_j into structured components $F_j^{(i)}$, each capturing a distinct dimension of install-time or post-install-time activity. Table 3 summarizes these eBPF-derived feature sets for each package π_j, describing their analytical focus and relevance to threat detection.

These features collectively define the vectorized representation $\mathcal{T}_j = \Phi(T_j) \in \mathbb{R}^q$, where Φ denotes the eBPF-based feature extraction operator and q represents the dimensionality across all trace types. Unlike static or metadata-based approaches, this approach captures latent behaviors that manifest

Table 3: Definitions of eBPF-based feature sets for package π_j.

Feature Sets	Description
$F_j^{(ft)} = \text{FiletopTraces}(\pi_j)$	File I/O process; detects abnormal file access or missing files.
$F_j^{(it)} = \text{InstallTraces}(\pi_j)$	Dependency logs; indirect malicious installs.
$F_j^{(ot)} = \text{OpensnoopTraces}(\pi_j)$	File open attempts; flags access to protected directories.
$F_j^{(tt)} = \text{TCPTraces}(\pi_j)$	TCP flows; identifies connections to suspicious endpoints.
$F_j^{(st)} = \text{SysCallTraces}(\pi_j)$	System call activity; indicates sabotage or privilege misuse.
$F_j^{(pt)} = \text{PatternTraces}(\pi_j)$	Behavioral sequences; detects loops, or payload triggers.

only during install-time and post-install-time, allowing for the detection of advanced threats such as ransomware, backdoors, and privilege escalation. A detailed description of `QUT-DV25` feature types, along with representative examples, is provided in Appendix Table 6.

4 Technical Validation and Benchmarks of QUT-DV25

This section presents the technical validation and performance benchmarking of candidate ML models for MDS, using the proposed `QUT-DV25` dataset.

Data preparation: The trace set $\mathcal{T} = \{T_j\}_{j=1}^{m}$ underwent preprocessing to ensure compatibility with ML models. Duplicate packages were removed, incomplete traces discarded, and all installations were aligned to a uniform directory structure across devices to eliminate identifier bias. Each trace $T_j \in \mathcal{T}$ was transformed into a feature vector $x_j \in \mathbb{R}^d$, where categorical features were encoded as n-gram [37] frequency vectors $\phi_{\text{cat}}(T_j) \to \mathbb{R}^{d_c}$, and numerical features were scaled via min-max normalization: $x'_{j,i} = (x_{j,i} - \min(x_i))/(\max(x_i) - \min(x_i))$, for all i in numerical features [38]. The final feature vector is $x_j = [\phi_{\text{cat}}(T_j), \phi_{\text{num}}(T_j)] \in \mathbb{R}^d$, suitable for training ML models.

Feature extraction and selection: From the trace set $\mathcal{T}$, 62 candidate features were extracted, forming the set $CF = \{f_i\}_{i=1}^{62}$, where each f_i denotes an attribute derived from the traces. To eliminate redundancy, features with a Pearson correlation coefficient [39] $|r_{ij}| > 0.50$ for any pair $(f_i, f_j) \in CF$ were pruned, resulting in an independent feature subset $IDF \subset CF$ with $|IDF| = 40$. For each feature $f \in IDF$, an importance score $IMS_m(f) \in [0, 1]$ was computed using each model $m \in M = \{\text{RF}, \text{DT}, \text{SVM}, \text{GB}\}$. The selected engineered feature set was defined as $SEF = \{f \in IDF \mid \max_{m \in M} IMS_m(f) > 0.08\}$, yielding $|SEF| = 36$, which corresponds to a 58% reduction in the original feature set. Models were trained with the following hyperparameters: RF with `n_estimators`=100 and `max_depth`=8; DT with `max_depth`=8 and `min_samples_split`=10; SVM with a linear kernel; and GB with `n_estimators`=100, `max_depth`=5, and `learning_rate`=0.1. The dataset $\mathcal{D}$ was divided into training, validation, and testing subsets in the ratio $\mathcal{D}_{\text{train}} : \mathcal{D}_{\text{val}} : \mathcal{D}_{\text{test}} = 70{:}15{:}15$. Five-fold stratified cross-validation was employed for hyperparameter tuning, and final evaluation was conducted on $\mathcal{D}_{\text{test}}$.

4.1 Experiments with ML Models

Each model $m \in M$ was evaluated using accuracy $\mathcal{A}$, precision $\mathcal{P}$, recall $\mathcal{R}$, and F1-score $\mathcal{F}_1$, capturing overall detection correctness, robustness to false positives/negatives, and class imbalance.

Feature set performance analysis: The impact of trace-derived feature subsets $\mathcal{T} = \{T_j\}_{j=1}^{m}$ and their union `CombinedTraces` $= \cup\, \mathcal{T}$ on model performance was evaluated, as presented in Table 4. For each model $m \in M$, features from `CombinedTraces` consistently yielded the highest performance (e.g., $\mathcal{A}_{\text{RF}} = 95.99\%$, $\mathcal{P}_{\text{RF}} = 96.00\%$, $\mathcal{F}_{1,\text{RF}} = 66.47\%$), outperforming any individual trace subset $T_j \in \mathcal{T}$. This improvement is attributed to feature complementarity: `FiletopTraces` captures resource I/O patterns; `OpensnoopTraces`, file access anomalies; `TCPTraces`, suspicious netflows; `SysCallTraces`, syscall anomalies; and `PatternTraces`, multi-stage attack sequences. Although `InstallTraces` alone show limited discriminative power ($\mathcal{A}_{\text{RF}} = 69.45\%$, $\mathcal{F}_{1,\text{RF}} = 66.47\%$) due to overlapping installation metadata, their inclusion in `CombinedTraces` enhanced attack coverage.

For standalone trace evaluation, performance varied across trace subsets $T_j \in \mathcal{T}$. `PatternTraces` demonstrated the highest effectiveness ($\mathcal{A}_{\text{RF}} = 94.62\%$, $\mathcal{F}_{1,\text{RF}} = 94.61\%$), reflecting its capacity

to capture high-level behavioral signatures. `SysCallTraces` achieved strong performance ($\mathcal{A}_{\mathrm{RF}} = 88.51\%$), while `FiletopTraces` and `TCPTraces` showed moderate results ($\mathcal{A}_{\mathrm{RF}} = 92.01\%$ and $\mathcal{A}_{\mathrm{RF}} = 83.74\%$, respectively). `InstallTraces` remained the least informative ($\mathcal{A}_{\mathrm{RF}} = 69.45\%$), reinforcing their limited standalone utility. These findings highlight that combining heterogeneous trace types enables cross-domain behavioral reasoning and maximizes detection capability.

Table 4: Performance of ML models across features: bold indicates the best, ↑ second-best, ↓ third-best, and underline denotes the lowest value.

	Metrics	Filetop	Install	Opensnoop	TCP	SysCall	Pattern	Combined
RF	$\mathcal{A}$	92.01	69.45	93.55	83.74	88.51	94.62	**95.99**
	$\mathcal{P}$	92.10	80.28	93.62	83.74	88.51	94.95	**96.00**
	$\mathcal{R}$	92.01	69.45	93.55	83.74	88.51	94.62	**95.99**
	$\mathcal{F}_1$	92.00	66.47	93.55	83.74	88.51	94.61	**96.02**
DT	$\mathcal{A}$	86.87	69.50	91.35	81.13	88.41	94.62 ↓	94.02
	$\mathcal{P}$	86.87	80.84	91.36	81.21	88.41	94.95 ↓	94.36
	$\mathcal{R}$	86.87	69.50	91.35	81.13	88.41	94.62 ↓	94.02
	$\mathcal{F}_1$	86.87	66.43	91.35	81.11	88.41	94.61 ↓	94.28
SVM	$\mathcal{A}$	89.77	68.65	80.05	80.47	85.56	94.53	95.28 ↑
	$\mathcal{P}$	89.85	80.65	81.65	80.55	85.57	94.87	95.30 ↑
	$\mathcal{R}$	89.77	68.65	80.05	80.47	85.56	94.53	95.28 ↑
	$\mathcal{F}_1$	89.76	65.39	79.79	80.46	85.56	94.52	95.23 ↑
GB	$\mathcal{A}$	87.38	67.16	91.54	80.47	85.61	<u>94.58</u>	94.11
	$\mathcal{P}$	87.42	79.31	91.67	80.72	85.62	<u>94.88</u>	94.42
	$\mathcal{R}$	87.38	67.16	91.54	80.47	85.61	94.58	<u>94.61</u>
	$\mathcal{F}_1$	87.38	63.39	91.53	80.43	85.61	<u>94.57</u>	94.35

Comparison with baseline datasets: An effective MDS dataset must balance accuracy, efficiency, and generalization. The proposed `QUT-DV25` was compared against two ML-based baselines: (i) `MetadataDataset`, based on the method by Halder et al. [17], and (ii) `StaticDataset`, following the approach of Samaana et al. [5]. To ensure fairness, the original models, feature selection strategies, and hyperparameters were applied to features derived from the common corpus. As shown in Table 5, `QUT-DV25` with `CombinedTraces` and RF achieved the highest performance across all metrics: $\mathcal{A}_{\mathrm{RF}} = 95.99\%$ and $\mathcal{F}_{1,\mathrm{RF}} = 96.02\%$. Confusion matrix analysis further confirms its robustness with $\mathrm{TPR} = 96.36\%$, $\mathrm{TNR} = 98.26\%$, $\mathrm{FPR} = 1.74\%$, and $\mathrm{FNR} = 3.64\%$.

Table 5: Performance comparison with existing datasets; bold indicates the overall best values.

Dataset	M	$\mathcal{A}$ (%)	$\mathcal{F}_1$ (%)	TPR (%)	TNR (%)	FPR (%)	FNR (%)
Metadata Dataset [20]	RF	84.44	84.81	82.98	86.10	13.90	17.02
	DT	83.93	84.36	82.26	85.76	14.24	17.74
	SVM	80.47	81.60	77.26	84.59	15.41	22.74
	GB	83.46	84.25	80.52	87.04	12.96	19.48
Static Dataset [22]	RF	95.14	95.24	93.37	97.06	2.94	6.63
	DT	95.14	95.29	92.45	98.20	1.80	7.55
	SVM	95.32	95.30	96.01	94.65	5.35	3.99
	GB	94.90	95.08	92.06	98.19	1.81	7.94
`QUT-DV25`	RF	**95.99**	**96.02**	95.26	96.77	3.23	4.74
	DT	94.02	94.28	90.48	**98.26**	**1.74**	9.52
	SVM	95.28	95.23	**96.36**	94.24	5.76	**3.64**
	GB	94.11	94.35	90.71	98.16	1.84	9.29

In contrast, `MetadataDataset` exhibited high false positives ($\mathrm{FPR} = 13.90\%$), attributable to its dependence on superficial package attributes, leading to poor generalization. Similarly, `StaticDataset` lacked install-time and post-install-time features, resulting in elevated false negatives ($\mathrm{FNR} = 6.63\%$). Both baselines struggled with previously unseen samples. `QUT-DV25` outperforms both meta and static dataset baselines across $\mathcal{A}$, $\mathcal{F}_1$, and confusion matrix dimensions.

This dataset also enables the distinction between benign and malicious system call patterns [40]. By facilitating the differentiation of these patterns, it supports a more robust evaluation of their

discriminative power in classification tasks. Notably, the RF classifier trained on this dataset flagged 6 benign packages $\mathcal{B}_{\text{flagged}} \subset \mathcal{D}_{\text{test}}$ from PyPI as malicious. Manual analysis confirmed malicious behaviors (e.g., data exfiltration, port scanning, socket proxy, remote access), leading to the removal of 4 packages. The remaining 2 exhibited dual-use traits $\phi(T_j) \in SEF$, including socket proxying and NetCat bundling, highlighting `QUT-DV25`'s sensitivity to repurposable behaviors. These findings demonstrate the dataset's real-world effectiveness with low FNR and high precision.

5 Technical Limitations and Other Applications

Technical limitations: The performance of MDS depends on the quality and representativeness of the proposed dataset $\mathcal{D}$, where each sample comprises install-time and post-install-time traces $T_j \in \mathcal{T}$. These traces may include noise that obscures discriminative patterns. To mitigate this, all T_j were collected using eBPF within isolated Linux-based sandboxes, ensuring clean environments but introducing a platform dependency and setup overhead. To prevent inconsistencies due to dependency reuse, each package was installed in a fresh virtual environment. The extracted feature vectors $\phi(T_j) = x_j \in \mathbb{R}^d$ were high-dimensional, increasing dataset processing complexity. Dimensionality reduction techniques were applied to obtain a selected embedding $SEF_j \subset \mathbb{R}^d$, which preserves relevant semantics while improving efficiency. Since $\mathcal{D}$ is collected from PyPI packages, generalization to other ecosystems (e.g., NPM) may require retraining or domain adaptation. Furthermore, reliance on $\phi(T_j)$ may limit detection of delayed or obfuscated threats. To address this, future work will incorporate runtime traces and extend $\phi_{\text{post-install-time}}(T_j^{>120s})$ to enhance dataset detection robustness.

Other applications of QUT-DV25: The proposed dataset enables training ML models for malicious package detection using user and system-level traces and modeling multi-stage attacks such as dynamic payload execution and covert remote access. It also supports feature attribution studies for understanding behavioral indicators of compromise and provides a benchmark for evaluating dynamic detection systems. With eBPF-collected behavioral data from 14,271 PyPI packages, `QUT-DV25` offers a practical foundation for advancing dynamic malware analysis in software supply chains. This study benefits society by enhancing software supply chain security and leading to the removal of four previously undetected malicious PyPI packages. However, techniques like eBPF tracing, while safely handled in controlled environments here, could pose risks if misused for surveillance or exploitation.

6 Safety and Ethical Discussion

All benign packages used in this study were collected from publicly available Python packages in the PyPI repository. and all malicious packages collected from different publicly available websites based on a details security report. No user-generated or private data were included in the dataset $\mathcal{D}$, ensuring compliance with privacy norms and ethical research standards. The dynamic analysis was conducted in isolated, networked-controlled sandboxes to prevent accidental propagation of malicious behavior and ensure containment. The eBPF monitored only user and kernel-level behaviors $T_j \in \mathcal{T}$ within controlled environments, without logging personal or sensitive content. Also, the dataset $\mathcal{D}$ and feature extraction function $\phi(T_j)$ were designed solely for research purposes to advance open malware detection techniques. To discourage misuse, any release of $\mathcal{D}$ or $\phi(T_j)$ will undergo related institutional review and include documentation outlining ethical usage guidelines.

7 Conclusion and Future Works

Existing software supply chain security benchmarks fail to capture evolving threats such as typosquatting, delayed payloads, and covert remote access. To address this gap, `QUT-DV25` is introduced as a dynamic analysis dataset constructed in a controlled sandbox environment using eBPF kernel and user-level probes. The dataset models real-world PyPI packages by capturing 36 features, including system calls, network activity, and installation traces, across 14,271 packages, of which 7,127 exhibited malicious characteristics. In contrast to static and metadata-based datasets that focus on only surface-level attributes, `QUT-DV25` reflects modern attack behaviors observed during install-time and post-install-time. Comparative evaluation demonstrates superior performance in modeling complex, dynamic threat vectors. `QUT-DV25` serves as a modern benchmark for dynamic malware detection and contributes to the advancement of next-gen supply chain threat defenses. Future work includes extending the dataset to other ecosystems and integrating ML frameworks for automated threat hunting across the open-source software supply chain ecosystem.

References

[1] Synopsys Software Integrity Group. 2024 open source security and risk analysis (ossra) report, 2024. Accessed: December 15, 2024.

[2] HIPAA Journal. Open source security risks, 2025. Accessed: January 1, 2025.

[3] Python Software Foundation. Pypi - the python package index, 2025. Accessed: July 2, 2024.

[4] PyPI Stats. Pypi download statistics, 2025. Accessed: July 2, 2024.

[5] Haya Samaana, Diego Elias Costa, Emad Shihab, and Ahmad Abdellatif. A machine learning-based approach for detecting malicious pypi packages. *arXiv preprint arXiv:2412.05259*, 2024.

[6] Snyk Security. Snyk security - vulnerability database and security insights, 2025. Accessed: May 18, 2024.

[7] National Institute of Standards and Technology (NIST). National vulnerability database (nvd), 2025. Accessed: June 1, 2024.

[8] VirusTotal. Virustotal - free online virus, malware and url scanner, 2024. Accessed: July 12, 2024.

[9] Synopsys Software Integrity Group. Black duck software composition analysis, 2024. Accessed: December 15, 2024.

[10] Ori Abramovsky. Detecting malicious packages on pypi: Malicious package on pypi use phishing techniques to hide its malicious intent. https://blog.checkpoint.com/2023/03/18/detecting-malicious-packages-on-pypi-malicious-package-on-pypi-use-phishing-techniques-to-hide-its-malicious-intent/, 2023. Accessed: July 12, 2024.

[11] The Hacker News. Pypi attack exploiting chatgpt and claude, 2024. Accessed: December 8, 2024.

[12] Checkmarx Security Research. Typosquatting attack on 'requests': One of the most popular python packages. https://zero.checkmarx.com/typosquatting-attack-on-requests-one-of-the-most-popular-python-packages-3b0a329a892d, 2024. Accessed: December 6, 2024.

[13] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.

[14] Xiaoyu Zheng, Chao Wei, Shuo Wang, Yuyang Zhao, Peng Gao, Yuhong Zhang, Ke Wang, and Hao Wang. Towards robust detection of open source software supply chain poisoning attacks in industry environments. *arXiv preprint arXiv:2409.09356*, 2024.

[15] Checkpoint Research. Detecting malicious packages on pypi: Example of the 'phpass' package dynamically generating payloads, 2024. Accessed: June 3, 2024.

[16] Lukas Martini. Psa: There is a fake version of this package on pypi with malicious code, 2019. Accessed: June 3, 2024.

[17] Sajal Halder, Michael Bewong, Arash Mahboubi, Yinhao Jiang, Md Rafiqul Islam, Md Zahid Islam, Ryan HL Ip, Muhammad Ejaz Ahmed, Gowri Sankar Ramachandran, and Muhammad Ali Babar. Malicious package detection using metadata information. In *Proceedings of the ACM Web Conference 2024 (WWW '24)*, page 11, 2024.

[18] Jukka Ruohonen, Kalle Hjerppe, and Kalle Rindell. A large-scale security-oriented static analysis of python packages in pypi. *arXiv preprint arXiv:2107.12699*, 2021.

[19] Anusha Damodaran, Fabio Di Troia, Visaggio Aaron Corrado, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *arXiv:2203.09938*, 2022.

[20] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023. Dataset available at `https://github.com/lxyeternal/pypi_malregistry`.

[21] Marc Ohm, Henrik Plate, Andreas Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2020. Dataset available at `https://github.com/cybersecsi/Backstabbers-Knife-Collection`.

[22] DataDog Security Labs. Malicious software packages dataset. `https://github.com/DataDog/malicious-software-packages-dataset`, 2023. Accessed: July 31, 2024.

[23] Tahir Iqbal, Guowei Wu, Zahid Iqbal, Muhammad Bilal Mahmood, Amreen Shafique, and Wenbo Guo. Pypiguard: A novel meta-learning approach for enhanced malicious package detection in pypi through static-dynamic feature fusion. *Journal of Information Security and Applications*, 90:104032, 2025. Dataset available at `https://github.com/tahir-biit/PyPiGuard/blob/main/pypiguard%20dataset.csv`.

[24] Amir Afianian, Salman Niksefat, Hamid Reza Shahriari, and Rasool Jalili. Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys*, 52(6):1–28, 2020.

[25] Asaf Eitani. Detecting ebpf malware with tracee. *Aqua Security Blog*, 2023.

[26] Ethan Bommarito and Michael Bommarito. An empirical analysis of the python package index (pypi). *arXiv preprint arXiv:1907.11073*, 2019.

[27] Kai Gao, Weiwei Xu, Wenhao Yang, and Minghui Zhou. PyRadar: Towards automatically retrieving and validating source code repository information for PyPI packages. *arXiv preprint arXiv:2404.16565*, 2024.

[28] Junan Zhang, Kaifeng Huang, Bihuan Chen, Chong Wang, Zhenhao Tian, and Xin Peng. Malicious package detection in npm and pypi using a single model of malicious behavior sequence. *arXiv preprint arXiv:2309.02637*, 2023.

[29] Jukka Ruohonen, Kalle Hjerppe, and Kalle Rindell. A large-scale security-oriented static analysis of python packages in pypi. In *Proceedings of the 18th Annual International Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2021.

[30] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, pages 421–430, 2007.

[31] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. A benchmark comparison of python malware detection approaches. *arXiv preprint arXiv:2209.13288*, 2022.

[32] Ax Sharma. New 'pymafka' malicious package drops cobalt strike on macos, windows, linux. https://www.sonatype.com/blog/new-pymafka-malicious-package-drops-cobalt-strike-on-macos-windows-linux, 2022. Accessed: June 3, 2024.

[33] Dave Bogle. eBPF: A new frontier for malware. *Red Canary Blog*, 2023.

[34] Danyil Zhuravchak and Valerii Dudykevych. Real-time ransomware detection by using ebpf and natural language processing and machine learning. In *2023 IEEE 5th International Conference on Advanced ICT*, 2023.

[35] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf. *arXiv preprint arXiv:2302.10366*, 2023.

[36] Kosuke Higuchi and Ryotaro Kobayashi. Real-time defense system using ebpf for machine learning-based ransomware detection method. In *2023 Eleventh International Symposium on Computing and Networking Workshops*, pages 213–219, 2023.

[37] M. Z. Masud, S. Sahib, M. F. Abdollah, S. R. Selamat, and R. Yusof. An evaluation of n-gram system call sequence in mobile malware detection. *ARPN Journal of Engineering and Applied Sciences*, 11(5):3122–3126, 2016.

[38] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, New York, NY, 2nd edition, 2009.

[39] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

[40] Sk Tanzir Mehedi, Chadni Islam, Gowri Ramachandran, and Raja Jurdak. Dysec: A machine learning-based dynamic analysis for detecting malicious packages in pypi ecosystem. *arXiv preprint arXiv:2503.00324*, March 2025.

A Supplementary Material

Table 6: Detailed description of QUT-DV-25 features with examples.

Feature Name	Description	Examples
Package_Name	Package Name and Version	1337z-4.4.7, 1337x-1.2.6
	Filetop Traces	
Read_Processes	Processes in reading	pip reads setup.py for metadata
Write_Processes	Processes in writing	writes to site-packages and cached .whl
Read_Data_Transfer	Reading data transfer	pip reads .whl file from PyPI via HTTPS
Write_Data_Transfer	Writing data transfer	pip writes downloaded .whl into the local
File_Access_Processes	Processes in access files	Accesses _init_.py during installation
	Install Traces	
Total_Dependencies	Total number of dependencies	2 (attrs-24.2.0; beautifulsoup4-0.1)
Direct_Dependencies	List of direct dependencies	1 (beautifulsoup4-0.1)
Indirect_Dependencies	List of indirect dependencies	1 (attrs-24.2.0)
	Opensnoop Traces	
Root_DIR_Access	Root directory access	2 (/root/.ssh/authorized_keys)
Temp_DIR_Access	Temp directory access	15 (/tmp/pip-wheel-pzrcqrtt/htaces.whl)
Home_DIR_Access	Home directory access	55 (/home/Analysis/Env/1337z-4.4.7.)
User_DIR_Access	User directory access	226 (/usr/lib/python3.12/lib-dynload)
Sys_DIR_Access	System directory access	12 (/sys/kernel/net/ipv4/ip_forward)
Etc_DIR_Access	Etc directory access	116 (/etc/host.conf, /etc/nftables.conf)
Other_DIR_Access	Access to other directories	17 (/proc/sys/net/ipv4/conf, /.ssh)
	TCP Traces	
State_Transition	TCP lifecycle transitions	{CLOSE -> ->: 15, SYN_SENT}
Local_IPs_Access	Access to local IP addresses	2 (192.168.0.51, 192.168.0.1)
Remote_IPs_Access	Access to remote IP addresses	2 (151.101.0.223, 3.164.36.120)
Local_Port_Access	Access to local ports	3 (52904, 53158, 34214)
Remote_Port_Access	Access to remote ports	3 (443, 23, 6667)
	SysCall Traces	
IO_Operations	I/O operations performed	ioctl, poll, readv
File_Operations	File-related system calls	open, openat, creat
Network_Operations	Network-related operations	socket, connect, accept
Time_Operations	Time-based operations	clock_gettime, timer_delete
Security_Operations	Security-related sys calls	getuid, setuid, setgid
Process_Operations	Process management sys calls	fork, vfork, clone
	Pattern Traces	
Pattern_1	File metadata retrieval	newfstatat→openat→fstat
Pattern_2	Reading data from a file	read→pread64→lseek
Pattern_3	Writing data to a file	write→pwrite64→fsync
Pattern_4	Network socket creation	socket→bind→listen
Pattern_5	Creating a new process	fork→execve→wait4
Pattern_6	Memory mapping	mmap→mprotect→munmap→no-fd
Pattern_7	File descriptor management	dup→dup2→close→stdout
Pattern_8	Inter-process communication	pipe→write→read→pipe-fd
Pattern_9	File locking	fcntl→lockf→close→file-fd
Pattern_10	Error handling	open→read→error=ENOENT→no-fd
Labels	Classification level	[1,0]