International Journal of Computer Applications (0975 - 8887)
Volume 187 - No.154, November 2026

Slopsquatting-Aware Detection of Malicious PyPI Packages via Interpretable Multi-Signal Risk Fusion

Mohanad Ali Al-Alwan
Department of Cybersecurity
Aqaba University of Technology
Aqaba, Jordan

Ahmad Hweishel A. Alfarjat
Department of Cybersecurity
Aqaba University of Technology
Aqaba, Jordan

Wa'el Ibrahim A. Almazaydeh
Department of Cybersecurity
Aqaba University of Technology
Aqaba, Jordan

Rajesha N.
Central Institute of Indian Languages
Mysore, Karnataka
India

ABSTRACT

PyPI accepts uploads from anyone, and the code in a package runs as soon as the package is installed. Attackers have used this for years. More recently, coding assistants based on large language models (LLMs) have created another opening, because they sometimes recommend packages that do not exist. If an attacker registers such an invented name (slopsquatting), developers who follow the model's advice install the attacker's package. This paper describes a static detector for malicious PyPI packages that looks at three kinds of evidence in each upload: the code that runs at install time, the metadata stored in the archive, and the risk carried by the package name, which includes a check against names that LLMs are known to hallucinate. The three scores are merged with learned weights that cannot be negative and must add up to one, so for any alert an analyst can see how much each kind of evidence contributed. For testing, 1,569 malicious packages were taken from Datadog's public collection, and 1,313 benign ones were downloaded from PyPI, some popular and some rarely used. Five-fold cross-validation gave F1 = 0.969 (bootstrap interval 0.964 to 0.976) with 1.6% of benign packages flagged. When the threshold was set so that only one benign package in a thousand is flagged, 89.7% of the malware was still detected, compared with 40.3% for code analysis alone. On the same 533 packages, the threat rules of the GuardDog scanner gave an FPR of 23.8% against 2.1% for the proposed score (McNemar test, $p < 10^{-12}$). Performance fell to an F1 score of 0.862 for attacks first reported in 2025–2026, and only 3 of 17 compromised releases of legitimate libraries were detected.

General Terms

Security, Software supply chain, Machine learning

Keywords

Software supply chain security, malicious package detection, PyPI, slopsquatting, package hallucination, large language models

1. INTRODUCTION

A Python package can execute code on the developer's machine before any of its functions are called. A `setup.py` script may open a network connection, read a browser profile, or download another program during `pip install`, and PyPI does little to vet new publishers. Many malicious uploads of this kind have been reported [1, 2, 3]. A few attacks are long-running and targeted; in the 2024 xz-utils case, the attacker spent years gaining maintainer access before planting a backdoor [4]. Most cases are much simpler. Someone creates a new package that steals credentials or tokens and gives it a name that people are likely to type or copy by mistake [3, 5]. This paper deals with that second group.

LLM-based coding assistants add a further route for such packages. Spracklen et al. [6] generated more than 500,000 code samples with sixteen models and found that about one in five samples from open-source models named a package that did not exist. If a developer runs `pip install` on one of these names, the package installed is whatever an attacker has meanwhile registered under it; the same authors called this slopsquatting. Later work shows that the problem persists. In 2026, five current models were found to hallucinate 127 identical names [7], and in 2024 a harmless test package uploaded under the invented name `huggingface-cli` was downloaded tens of thousands of times [8].

Most published detectors rely on one source of evidence. Name-based tools compare new names with popular packages [9], code-based tools inspect what the package does [10, 11], and learned models classify sequences of operations [12]. In the tools surveyed for this work, hallucinated names were never used as a feature. Some tools also return just one number. If a package is taken down on that basis and later turns out to be harmless, the registry has little to point to.

In the proposed detector the three sources are scored separately and then merged, and the share of each remains visible in the result (Section 4). It was evaluated on 2,882 real packages (Sections 5 and 6). In addition to standard cross-validation, the evaluation covers detection at FPR values as low as 0.1%, the effect of the

Table 1. Related approaches and their main limitations

Approach	Signal	Limitation addressed here
Typosquat defence [9]	Name edit distance	Blind to code; ignores LLM-hallucinated names
Source and artifact diff [14]	Repository vs. package	Needs a linked repository
Static and dynamic analysis [10]	API and system calls	No metadata or name signal
Code classifiers [11]	Code features	One signal family
Behaviour sequence [12]	Learned sequences	Opaque score; no naming risk
GuardDog [15]	Rules and heuristics	Uncalibrated; high FPR (Section 6.6)
This work	Code, metadata, and name (including hallucination)	Interpretable calibrated fusion

threshold, a split by discovery date, ablation of single signals, and the contribution of each signal to the alerts. GuardDog is run on the same packages and tested for significance, LAMPS, a recent LLM-based detector, is compared at the design level, and compromised legitimate libraries serve as a stress test. Section 6.7 describes a short script that checks hallucinated names against the live registry. The cases in which the detector fails are examined in Section 7.

2. BACKGROUND AND RELATED WORK

Datasets and taxonomies. The first curated dataset of real malicious open-source packages was built by Ohm et al. [1], who also described how attackers inject and trigger malicious code. Ladisa et al. [2] then classified supply-chain attacks into a taxonomy of attack vectors, and Guo et al. [3] examined malicious code on PyPI in detail. For experiments, researchers mostly rely on two public sources, the Datadog dataset [5] and the OpenSSF Malicious Packages repository [13], and both keep growing.

Code-based detection. Duan et al. [10] applied static and dynamic analysis to npm, PyPI, and RubyGems. Vu et al. [14] proposed LastPyMile, which compares the files of a published package with those in its source repository; files present only in the package point to injection at release time. Sejfia and Schäfer [11] found that simple classifiers on code features already work well for npm. Zhang et al. [12] trained one model on sequences of malicious behaviour that can be applied to both npm and PyPI. GuardDog, which combines Semgrep rules with metadata checks, is widely used to screen new uploads [15].

Name-based detection. Typosquatting defences usually flag a new name when it is within one or two edits of a popular one [9]. That works for `reqeusts`. It does not work well for hallucinated names, which are rarely misspellings. A model that has seen `import opentelemetry` many times can easily suggest `pip install opentelemetry`, yet the packages that actually exist are `opentelemetry-api` and `opentelemetry-sdk` [6, 7].

The gaps are listed in Table 1. No earlier detector known to the authors puts code, metadata, and hallucination-aware name checks into one score that can be explained, and tests it on real malware.

3. THREAT MODEL

The attacker in this study publishes a brand-new package with harmful intent. Its name is picked so that people install it. Sometimes it is one or two letters off a popular package (typosquatting); sometimes a popular name with a word added (combosquatting); and sometimes a name that LLMs keep inventing. The payload is triggered either by `pip install` or by the first `import`. The defender, a registry or a scanning tool, receives only the archive and has to classify it statically. Takeovers of existing projects through stolen credentials are excluded here and treated separately in Section 6.8.

4. APPROACH

All analysis is static. Each archive is unpacked into a temporary directory and parsed; no package code is ever installed or run. Fig. 1 gives an overview.

Static sub-score $S(p)$. Every Python file is parsed into an abstract syntax tree (AST), and a visitor counts constructs often seen in malware. Counts are kept for `setup.py`, for `__init__.py`, and for the whole package. The counted items are calls to `exec`, `eval`, and `compile` (with a separate count when the argument is another call, e.g. `exec(b64decode(...))`), decoding and decompression calls, process creation, network imports and calls, and reads of environment variables or host details. In addition, the visitor records subclasses of setuptools install commands, top-level calls in `setup.py` that run at install time, long or high-entropy strings, URLs containing raw IP addresses, strings such as `webhook` or `token` or browser credential paths, and bundled binary files.

Metadata sub-score $M(p)$. These features are read from the `PKG-INFO` or `METADATA` file in the archive. They include the description length, an empty summary or one that only repeats the name, a missing author, e-mail address, homepage, repository link, or licence, an author address at a free-mail provider, a trivial version string such as `0.0.1` or `1.0.0`, and the number of classifiers and declared dependencies. Registry history, for example the age of the publisher account, is not used. PyPI removes malicious projects, so this history is missing for the malware, and using it only for the benign class would leak the label.

Name-risk sub-score $H(p)$. Names are first normalised according to PEP 503. The detector then computes the smallest Damerau–Levenshtein distance between the name and the 5,000 most downloaded PyPI projects (the package itself excluded) and checks whether a popular name appears inside the new one. Exact matches with disclosed hallucinated names [7, 8] are flagged, and a few lexical features, such as long digit sequences, are added.

Calibration and fusion. S, M, and H are produced by three separate logistic regressions with L2 regularisation, each fitted on its own feature group, so each output lies between 0 and 1. The fused score is

$$R(p) = \sum_{i=1}^{3} w_i \, s_i(p), \qquad w_i \geq 0, \quad \sum_{i=1}^{3} w_i = 1, \qquad (1)$$

with $(s_1, s_2, s_3) = (S, M, H)$. Writing the weights as $w = \mathrm{softmax}(\theta)$ keeps them non-negative with a sum of one; θ is found by minimising the log-loss of a logistic function of $R(p)$. To avoid overfitting, θ is estimated from out-of-fold sub-scores produced by an inner five-fold split of the training set. τ is simply the cut-off with the best training F1. Because the weights are non-negative and sum to one, an alert can be read directly, for example as 60% code evidence, 30% metadata, and 10% name.

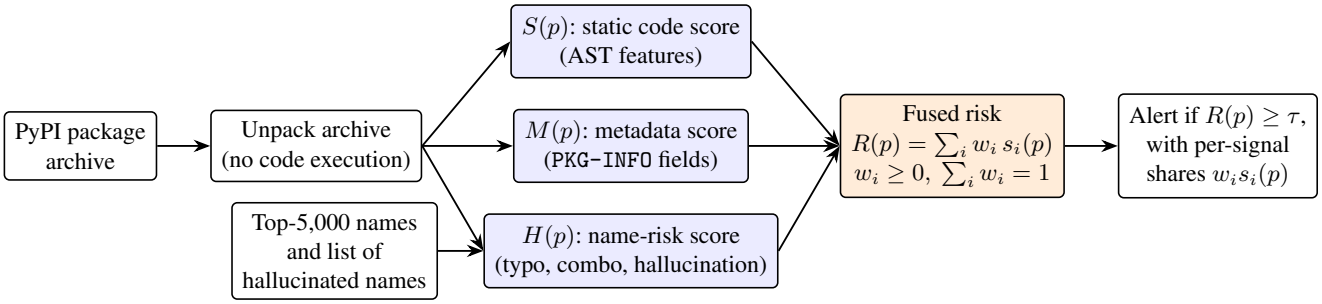

Fig. 1. Overview of the detector. Each sub-score is a calibrated probability, and the constrained weights make every alert decomposable into the contributions of code, metadata, and name.

5. EXPERIMENTAL SETUP

Malicious data. Malicious samples were taken from the PyPI part of the Datadog malicious-software-packages dataset [5], where each sample was reviewed by a person. A package often appears in several versions. To prevent almost identical copies from ending up in both training and test folds, only the newest sample of each package was kept, giving 1,569 packages published with malicious intent and discovered between 2022 and 2026. The 17 compromised legitimate packages in the dataset were set aside for a separate test.

Benign data. For the benign class, the latest source release of 1,313 projects was downloaded. Of these, 655 were sampled from the 1,000 most downloaded packages and 658 from ranks 5,001–15,000 of the October 2026 download ranking. The second sample was included on purpose, since small legitimate projects often have incomplete metadata; a model trained only on well-known libraries might learn to separate polished from unpolished packages instead of benign from malicious ones. Five archives (0.2%) could not be parsed and were dropped.

Baselines. $R(p)$ is compared with each sub-score on its own, with $R(p)$ using equal weights, with a Random Forest of 400 trees trained on all features together (a common machine-learning baseline), and with GuardDog v3.2.0 [15]. GuardDog was run locally on a random subset and evaluated in two ways: counting any finding, and counting only findings from its `threat-*` rules.

Protocol and metrics. AUC, average precision (AP), precision, recall, F1 score, FPR, and the Matthews correlation coefficient (MCC) are reported. Table 2 shows their means over stratified five-fold cross-validation. Some analyses (counts of flagged packages, recall per year) require one set of predictions, so for these the out-of-fold scores of all 2,882 packages were pooled and a single threshold of $\tau = 0.48$ was applied. Confidence intervals were estimated from 1,000 bootstrap samples of the pooled set. In the temporal test, the 1,364 malicious packages discovered before 2025 were used for training and the 205 discovered in 2025–2026 for testing; benign packages were split 60/40 at random.

6. RESULTS

6.1 Overall detection effectiveness

Results of the cross-validation are given in Table 2 and Fig. 2. With learned weights the fused score has an F1 score of 0.969, an AUC of 0.992, and an FPR of 1.6%. The standard deviation across folds is 0.010 for F1 and 0.004 for AUC. Both fused variants exceed every single signal on F1 and FPR, while the Random Forest trained on all features reaches F1 = 0.980.

Table 2. Five-fold cross-validation, mean over folds (1,569 malicious, 1,313 benign)

Method	AUC	Prec.	Rec.	F1	FPR
$H(p)$ only	0.650	0.572	0.940	0.711	84.1%
$M(p)$ only	0.974	0.960	0.924	0.942	4.7%
$S(p)$ only	0.989	0.952	0.971	0.961	5.9%
$R(p)$, equal weights	0.991	0.983	0.952	0.967	2.0%
$R(p)$, learned weights	0.992	0.986	0.953	0.969	1.6%
Random Forest (all features)	0.996	0.990	0.970	0.980	1.2%

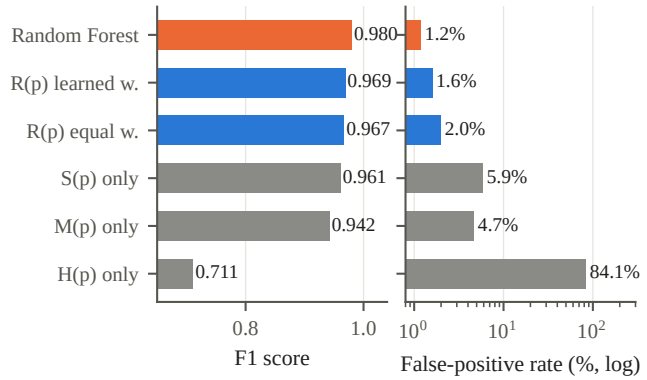

Fig. 2. F1 score and false-positive rate of each method under five-fold cross-validation (data of Table 2). Blue bars are fusion variants of the proposed score.

Table 3 lists the pooled counts. $R(p)$ produces 29 false positives among 1,313 benign packages, against 69 for $S(p)$ and 81 for $M(p)$, and 64 false negatives among 1,569 malicious packages. Its MCC is 0.935. The 95% bootstrap intervals are 0.964–0.976 (F1), 0.989–0.995 (AUC), and 1.5–3.0% (FPR). The pooled FPR of 2.2% exceeds the fold mean of 1.6% in Table 2 because a single threshold is applied to all folds instead of one threshold per fold.

6.2 Performance at low false-positive rates

In registry use only the left part of the ROC curve matters. A false alarm delays the release of a legitimate package, so a practical FPR must be well under 1%. Fig. 3(a) uses a logarithmic FPR axis for this reason, and Table 4 reports the true-positive rate at five FPR levels. With one false alarm per 1,000 benign packages, $R(p)$ detects 89.7% of the malicious packages, $S(p)$ 40.3%, and $M(p)$

Table 3. Confusion matrix and further metrics of each score (pooled out-of-fold predictions, F1-optimal threshold)

Score	τ	TP	FN	FP	TN	AP	MCC	Bal. acc.
$S(p)$	0.44	1,516	53	69	1,244	0.992	0.915	0.957
$M(p)$	0.35	1,479	90	81	1,232	0.980	0.880	0.940
$H(p)$	0.26	1,502	67	1,146	167	0.688	0.154	0.542
$R(p)$	0.48	1,505	64	29	1,284	0.995	0.935	0.969

Table 4. True-positive rate at five fixed FPR levels

Score	0.1%	0.5%	1%	2%	5%
$S(p)$	40.3%	83.7%	90.4%	91.0%	96.3%
$M(p)$	19.0%	41.6%	87.0%	89.5%	92.7%
$H(p)$	1.5%	3.0%	4.2%	13.7%	18.8%
$R(p)$	**89.7%**	**93.9%**	**94.6%**	**95.7%**	**97.2%**

19.0%. The difference shrinks at larger budgets; at 5% all scores that include code features exceed 92%. Fusion thus helps most at strict thresholds, which are the ones a registry would choose. The precision-recall curves (Fig. 3(b)) agree with this. Precision for $R(p)$ is at least 0.988 up to a recall of 0.95, and its AP (0.995) is the highest.

6.3 Score distribution and threshold sensitivity

Fig. 4(a) shows the two classes pulling apart. Half of the benign packages (48.7%) score below 0.1, and well over half of the malicious ones (58.9%) score above 0.9; the class medians are 0.10 and 0.91. Few packages land anywhere near the threshold. That is why its exact value matters little: according to Fig. 4(b), any τ from 0.38 to 0.64 keeps F1 within 0.01 of the best value. Inside that range, raising τ removes a handful of false alarms and adds a handful of misses. An operator can therefore set it to match the number of alerts the review team can handle, with no need to retrain.

6.4 Contribution of each signal

Averaged over folds, the learned weights are $w_S = 0.47$, $w_M = 0.39$, and $w_H = 0.14$, with standard deviations of 0.03, 0.02, and 0.05. $S(p)$ has the highest AUC of the three single scores (0.989). Metadata, however, adds almost as much: combining it with code lowers the FPR from 5.9% to 1.6% (a reduction of 73%). On its own the name score is weak (AUC 0.650). This is not surprising, as plugins and forks such as `pytest-cov` or `django-rest-framework` reuse popular names legitimately. It still receives a non-zero weight and improves the fused result. Most of the gain is already obtained with equal weights, so it comes mainly from combining different evidence rather than from the fitted weights.

Since Eq. (1) is additive, each alert can be divided into the shares $w_i s_i(p)/R(p)$. Fig. 5 shows these shares averaged over three groups. In true positives, code provides 51% of the score and metadata 40%; code is the largest share in 1,436 of the 1,505 cases and metadata in the remaining 69. In the 29 false positives code accounts for 60%, which is in line with what Section 6.6 finds for GuardDog: a library that launches subprocesses or pulls files from the web looks like malware to a code check. Missed packages are a separate story. On average they score 0.31 on code and 0.24 on metadata, and with so little evidence overall the name ends up supplying about a third. Typically these packages have complete metadata and a short or well-hidden payload, which static analysis finds difficult.

Table 5. Temporal split (train: attacks before 2025; test: 2025–2026)

Method	AUC	Prec.	Rec.	F1	FPR
$S(p)$ only	0.940	0.876	0.761	0.815	4.2%
$M(p)$ only	0.946	0.816	0.888	0.850	7.8%
$R(p)$, learned weights	0.954	0.904	0.824	0.862	3.4%
Random Forest (all features)	0.966	0.932	0.873	0.902	2.5%

Table 6. Recall by year of discovery (pooled out-of-fold scores; 9 packages from 2022 omitted)

Year	Packages	$S(p)$	$M(p)$	$R(p)$
2023	965	0.987	0.951	0.974
2024	390	0.977	0.944	0.972
2025	122	0.877	0.926	0.893
2026	83	0.819	0.855	0.831

6.5 Robustness over time

Table 5 and Fig. 6(a) give the results of the temporal split, in which the model is trained on attacks reported before 2025 and tested on attacks reported in 2025–2026. $R(p)$ obtains F1 = 0.862, AUC = 0.954, and FPR = 3.4%. The fitted w_H increases from 0.14 to 0.23 (Fig. 6(b)), which may indicate that naming plays a larger role in recent attacks.

Table 6 and Fig. 6(c) break recall down by year of discovery. Packages reported in 2023 and 2024 are detected at a rate of about 0.97; for 2025 the rate is 0.893 and for 2026 it is 0.831. Every fold had some 2025 and 2026 packages in its training data, so the drop is not due to their absence. Static and metadata scores decline in much the same way. This points to a change in how attacks are built, and the fusion step does not seem to be the cause. For deployment, the model would need to be retrained at regular intervals as new labelled samples become available.

One possible worry is that the model has learned to separate established projects from new ones. The data do not support this. On popular benign packages its FPR is 2.3% (15 of 655), and on long-tail packages 2.1% (14 of 658). $S(p)$ shows the same balance, at 5.3% and 5.2%.

6.6 Comparison with GuardDog

GuardDog and $R(p)$ were compared on the same 533 packages (298 malicious, 235 benign), using out-of-fold scores for $R(p)$. Table 7 and Fig. 7(a) summarise the outcome. GuardDog raised at least one finding on 77.4% of the benign packages. Restricting it to its threat rules lowers the FPR to 23.8%, still about ten times the 2.1% of $R(p)$. On these 533 packages, $R(p)$ was right where GuardDog's threat rules were wrong 79 times, and the opposite happened 12 times; an exact McNemar test rejects equal error rates ($p < 10^{-12}$). The comparison is, if anything, tilted towards GuardDog, because GuardDog found most of the samples in the dataset in the first place [5].

The sources of GuardDog's false alarms are shown in Table 8 and Fig. 7(b). Its capability rules match behaviour that is common in legitimate libraries. For example, the file-read rule fires on 66.8% of benign packages but on 22.8% of malicious ones, so it is actually more typical of benign code. The rule for writing executable files fires on 17.0% and 4.4% respectively. A threat rule that detects

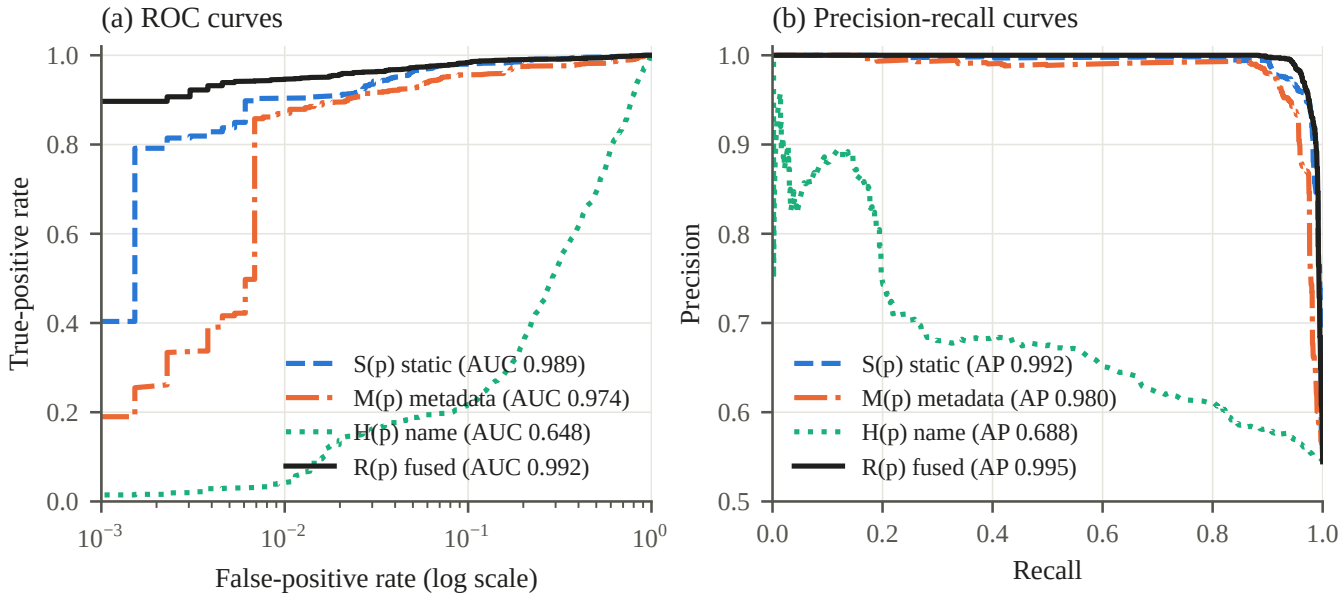

Fig. 3. (a) ROC curves of the out-of-fold scores on a logarithmic FPR axis. (b) Precision-recall curves. Line styles distinguish the scores in grey-scale print.

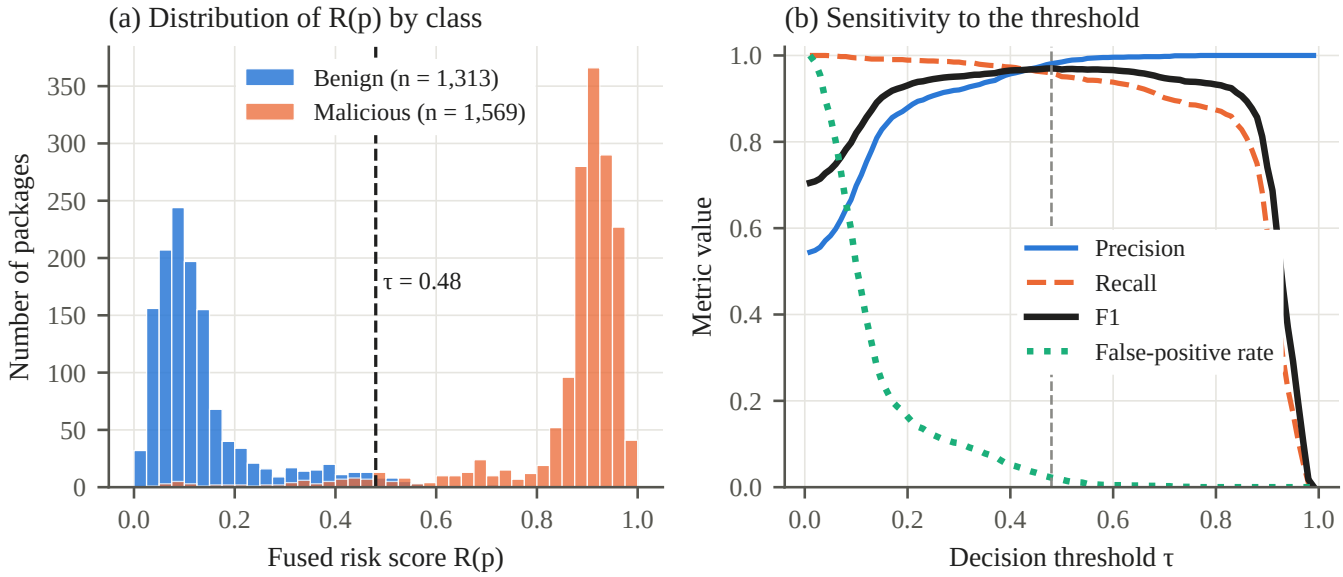

Fig. 4. (a) Distribution of the fused score $R(p)$ for benign and malicious packages; the dashed line marks the threshold $\tau = 0.48$. (b) Precision, recall, F1 score, and false-positive rate as functions of the threshold.

Table 7. Matched comparison with GuardDog on the same 533 packages

Method	AUC	Prec.	Rec.	F1	FPR
GuardDog, any finding	0.863	0.618	0.987	0.760	77.4%
GuardDog, threat rules	0.886	0.827	0.899	0.862	23.8%
$R(p)$ (this work)	0.997	0.983	0.953	0.968	2.1%

reading of environment variables fired on 9.8% of benign packages and on no malicious package at all. In a trained model, features of this kind get small weights and therefore do little harm.

6.7 Live exposure of hallucinated names

For each disclosed hallucinated name [7, 8], the PyPI JSON API was queried. On 13 October 2026 all nine names returned HTTP 404: `aws-cdk`, `rest-framework`, `objc`, `opentelemetry`, `tencentcloud`, `opengl`, `openstack`, `mpl-toolkits`, and `huggingface-cli`. A 404 response means that no project with that name exists, but it does not prove that the name can be

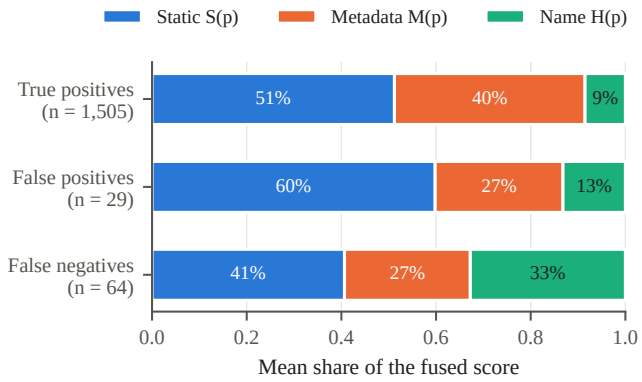

Fig. 5. Mean share of the fused score contributed by each signal, $w_i s_i(p)/R(p)$, for true positives, false positives, and false negatives at $\tau = 0.48$.

Table 8. Firing rates of frequent GuardDog rules

GuardDog rule	Benign	Malicious
`capability-filesystem-read`	66.8%	22.8%
`capability-process-spawn`	46.0%	75.2%
`capability-network-outbound`	39.6%	40.6%
`capability-filesystem-delete`	32.8%	10.7%
`capability-network-download`	17.4%	28.2%
`capability-filesystem-write-executable`	17.0%	4.4%
`threat-runtime-environment-read`	9.8%	0.0%

registered, since PyPI blocks certain names. The check needs only a few lines of code, so a registry could run it continuously and treat an upload under a listed name as strong evidence for $H(p)$.

6.8 Stress test: compromised legitimate packages

Seventeen compromised releases were held out, `litellm`, `ultralytics`, and `lightning` among them; $R(p)$ caught 3. In these cases the name and metadata are genuine, and the injected code amounts to a few lines inside thousands of ordinary ones. Features summed over a whole package simply wash it out.

6.9 Comparison with a published detector

LAMPS [16] is a recent peer-reviewed detector for malicious PyPI packages. It coordinates four LLM agents and pairs a fine-tuned CodeBERT classifier with LLaMA-3 reasoning, and its authors report 97.7% accuracy on a balanced set of 6,000 `setup.py` files and 99.5% accuracy and balanced accuracy on 1,296 files with natural class imbalance. The pooled results in Table 3 put $R(p)$ at 96.8% accuracy and 96.9% balanced accuracy. LAMPS was not re-run on the present data, and the datasets differ, so a direct comparison of these figures would be misleading. Table 9 compares the designs instead.

LAMPS reports higher accuracy. Because it reasons about the meaning of code, it is also likely to handle obfuscated payloads better than count-based features. $R(p)$ differs in other respects. It uses metadata and name features, including hallucinated names, runs on a standard CPU, was evaluated on a temporal split and against GuardDog, and its score can be decomposed by signal. The two methods could be chained, with $R(p)$ applied to all uploads and LAMPS applied only to those that $R(p)$ flags.

Table 9. Design comparison with LAMPS

Aspect	LAMPS [16]	This work
Unit of analysis	Files (`setup.py`; multi-file set)	Whole package (code, metadata, name)
Model	Four LLM agents, CodeBERT and LLaMA-3	Three logistic sub-scores, constrained fusion
Reported accuracy	97.7% (D1), 99.5% (D2)	96.8% (pooled cross-validation)
Naming and slopsquatting	Not modelled	Typo-, combo-squatting, hallucinated names
Temporal test	Not reported	Yes (F1 0.862)
Industry baseline	RAG and single-agent LLMs	GuardDog, matched subset
Explanation	Agent verdicts (text)	Additive per-signal shares
Compute	GPU and LLM inference	CPU, static parsing only

7. DISCUSSION

Interpretability vs. accuracy. The Random Forest reaches an F1 score of 0.980, against 0.969 for $R(p)$. This difference is small compared with the practical value of an explainable score. A wrongful takedown harms the package owner, and $R(p)$ shows whether code, metadata, or name caused the alert (Section 6.4). Nothing stops a registry from using the forest to rank and $R(p)$ to explain.

Implications. Fewer false alarms leave less for people to check by hand. Taking 10,000 benign uploads as an example, the rates measured here mean about 220 alerts to dismiss with $R(p)$ and about 2,400 with GuardDog's threat rules. Against slopsquatting in particular, registries could use $H(p)$ together with monitoring of names hallucinated by several model families, or reserve those names in advance. LLM providers could also reduce the risk by checking that a package exists before recommending it.

Limitations and threats to validity. (i) Most of the malicious samples were originally found by GuardDog [5]. The dataset may therefore contain more attacks of the kind its rules detect, and the results may not transfer to attacks that have not yet been found. (ii) The benign set consists of the latest releases of ranked projects, whereas a real upload stream contains many more low-quality but harmless packages. (iii) Obfuscated or dynamically loaded code can get past a purely static check, and no evasion experiments were run. (iv) Hardly any of the malware uses a known hallucinated name, so the hallucination part of $H(p)$ rests on the registry check rather than on a measured detection rate. (v) A lot of newer legitimate projects build from `pyproject.toml` and ship no `setup.py`: only 45.6% of the benign packages had one, against 90.6% of the malicious ones. Some of what the static score picks up may therefore be packaging age, not intent. (vi) Compromised releases of legitimate libraries are mostly missed. (vii) Only PyPI was studied.

8. CONCLUSION AND FUTURE SCOPE

A static detector for malicious PyPI packages has been presented. It merges a code score, a metadata score, and a name-risk score using constrained learned weights, and, as far as the authors know, it is the first detector that uses LLM-hallucinated names as a risk factor. On 2,882 real packages the F1 score was 0.969 at an FPR of 1.6%.

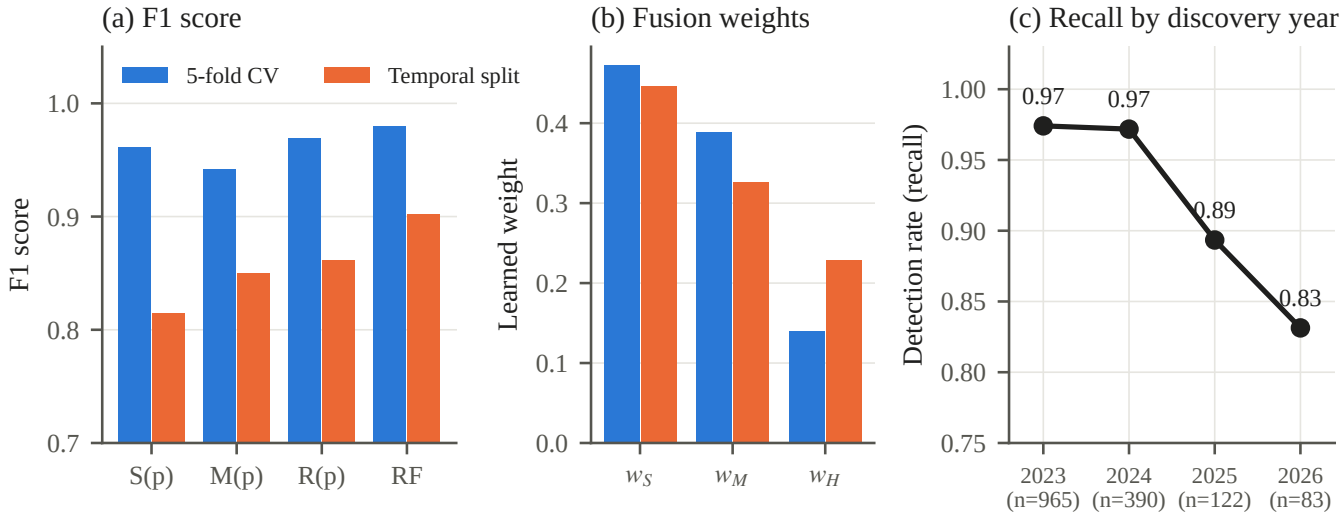

Fig. 6. (a) F1 score under cross-validation and under the temporal split. (b) Learned fusion weights in both settings; the name weight w_H grows on recent attacks. (c) Recall of $R(p)$ by year of discovery (pooled out-of-fold scores).

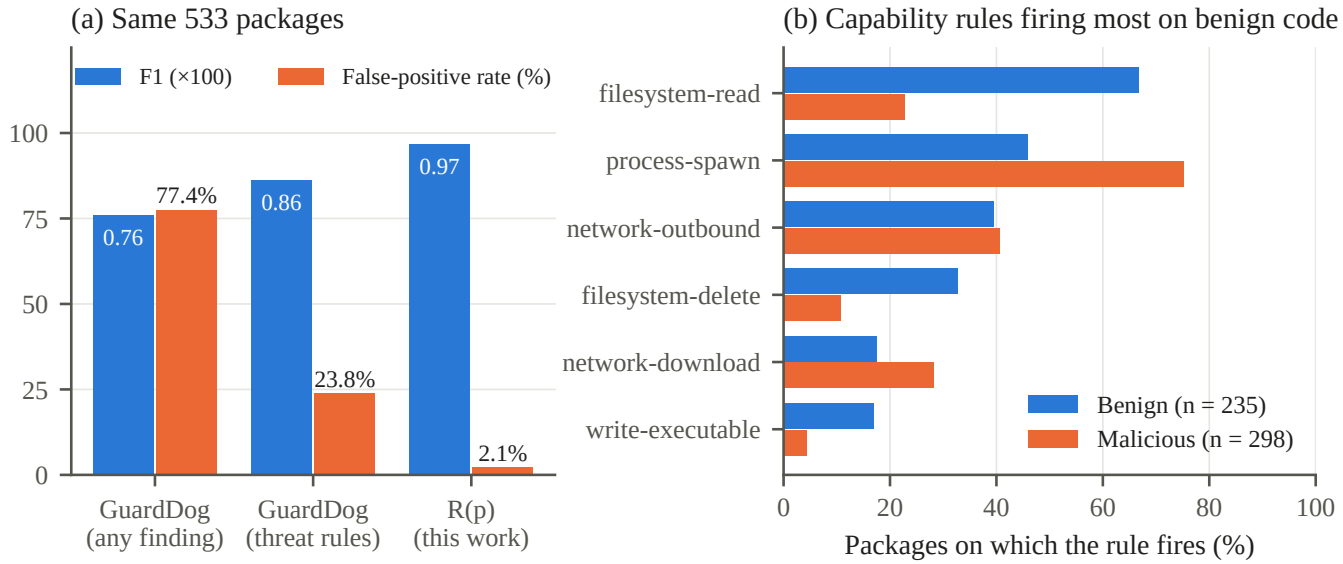

Fig. 7. (a) GuardDog and $R(p)$ on the same 533 packages (data of Table 7). (b) The six GuardDog capability rules that fire most often on benign packages, with their firing rates on malicious packages for comparison.

With the FPR limited to 0.1%, the detector still found 89.7% of the malicious packages, more than twice the rate of code analysis alone. F1 changes by less than 0.01 for thresholds between 0.38 and 0.64, and the score of every alert can be split into code, metadata, and name parts. GuardDog's threat rules gave about ten times more false positives on the same 533 packages ($p < 10^{-12}$). The detector is less accurate on recent attacks and has to be retrained regularly, and it misses most compromised releases of legitimate libraries.

Future work will start with version diffing. Comparing a release with the previous release and with the repository, as LastPyMile does [14], can isolate the small amount of code added to a compromised package; whole-package features miss exactly these cases. A dynamic score $D(p)$ could come from installing packages in a throw-away sandbox and watching the network; as a fourth term in Eq. (1) it would cost more per package but should catch some obfuscated payloads. Robustness to evasion techniques such as identifier renaming, string encoding, and control-flow flattening has not been measured and should be tested.

$H(p)$ would benefit most from better data. One option is a common list of names that several model families hallucinate, run jointly with the registries and refreshed whenever new models come out; with it, $H(p)$ could be checked against real slopsquatting uploads, and the names could be blocked early. Recall dropped for 2025 and 2026, so it makes sense to track how scores on new uploads drift and to retrain either on a timetable or when drift shows up. Further work includes support for npm and RubyGems, an LLM-based

second stage for flagged packages, and a field trial on live uploads to measure the actual reduction in review effort.
The code and the evaluation scripts will be published.

Acknowledgments

An AI assistant (Claude, Anthropic) was used while writing parts of the feature-extraction and evaluation code and while drafting and editing parts of the text. The authors re-ran the code, verified all results and references, and accept full responsibility for the content.

9. REFERENCES

[1] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Proc. DIMVA*, LNCS, vol. 12223. Springer, 2020, pp. 23–43.

[2] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of attacks on open-source software supply chains," in *Proc. IEEE Symp. Security and Privacy (S&P)*, 2023, pp. 1509–1526.

[3] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, "An empirical study of malicious code in PyPI ecosystem," in *Proc. IEEE/ACM ASE*, 2023.

[4] A. Freund, "Backdoor in upstream xz/liblzma leading to ssh server compromise," oss-security mailing list, Mar. 29, 2024. CVE-2024-3094.

[5] Datadog Security Labs, "Open-source dataset of malicious software packages," 2023–2026. [Online]. Available: `https://github.com/DataDog/malicious-software-packages-dataset`

[6] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We have a package for you! A comprehensive analysis of package hallucinations by code generating LLMs," in *Proc. USENIX Security Symp.*, 2025.

[7] A. Churilov, "The range shrinks, the threat remains: Re-evaluating LLM package hallucinations on the 2026 frontier-model cohort," arXiv:2605.17062, 2026.

[8] B. Lanyado, "Diving deeper into AI package hallucinations," Lasso Security, 2024.

[9] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "Defending against package typosquatting," in *Proc. Int. Conf. Network and System Security (NSS)*, LNCS, vol. 12570. Springer, 2020, pp. 112–131.

[10] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Proc. NDSS*, 2021.

[11] A. Sejfia and M. Schäfer, "Practical automated detection of malicious npm packages," in *Proc. IEEE/ACM Int. Conf. Software Engineering (ICSE)*, 2022.

[12] J. Zhang, K. Huang, B. Chen, C. Wang, Z. Tian, and X. Peng, "Killing two birds with one stone: Malicious package detection in NPM and PyPI using a single model of malicious behavior sequence," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2025.

[13] OpenSSF, "Malicious packages repository." [Online]. Available: `https://github.com/ossf/malicious-packages`

[14] D. L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, "LastPyMile: Identifying the discrepancy between sources and packages," in *Proc. ACM ESEC/FSE*, 2021, pp. 780–792.

[15] Datadog, "GuardDog: CLI tool to identify malicious PyPI and npm packages," v3.2.0. [Online]. Available: `https://github.com/DataDog/guarddog`

[16] M. U. Zeshan, M. Ibiyo, C. Di Sipio, P. T. Nguyen, and D. Di Ruscio, "Many hands make light work: An LLM-based multi-agent system for detecting malicious PyPI packages," *J. Syst. Softw.*, 2026, arXiv:2601.12148.