Detecting Active and Stealthy Typosquatting Threats in Package Registries

Wenxin Jiang
Purdue University

Berk Çakar
Purdue University

Mikola Lysenko
Socket, Inc.

James C. Davis
Purdue University

Abstract

Typosquatting attacks, also known as package confusion attacks, threaten software supply chains. Attackers make packages with names that resemble legitimate ones, tricking engineers into installing malware. While prior work has developed defenses against typosquatting in some software package registries, notably npm and PyPI, gaps remain: addressing high false-positive rates; generalizing to more software package ecosystems; and gaining insight from real-world deployment.

In this work, we introduce `TypoSmart`, a solution designed to address the challenges posed by typosquatting attacks. We begin by conducting a novel analysis of typosquatting data to gain deeper insights into attack patterns and engineering practices. Building on state-of-the-art approaches, we extend support to six software package registries using embedding-based similarity search, achieving a 73%–91% improvement in speed. Additionally, our approach significantly reduces 70.4% false-positive compared to prior work results. `TypoSmart` is being used in production at our industry partner and contributed to the removal of 3,658 typosquatting packages in one month. We share lessons learned from the production deployment.

1 Introduction

Software package registries (SPRs) have become indispensable in modern development, providing open-source packages which greatly reduce costs and accelerate product cycles [45, 61]. Open-source packages are used by industry and governments [2, 49], including in AI and safety-critical applications [18, 48]. Attackers seek to disrupt or exploit the resulting supply chains by creating packages with deceptive or look-alike names [10, 23, 39]. This practice is commonly called typosquatting or package confusion [20, 35]. Figure 1 shows the threat model.

Researchers have proposed many approaches to address package typosquatting, including Levenshtein distance [59], lexical similarity [35, 50], or imaging approach [44]. However, three gaps remain: (1) high false positive rate, (2) limited coverage across SPRs, and (3) limited insights from SPR perspectives. Prior work — triggering alarms approximately every 200 to 1,000 package installations — falls short of customer expectations, and has limited coverage on only three SPRs [50].

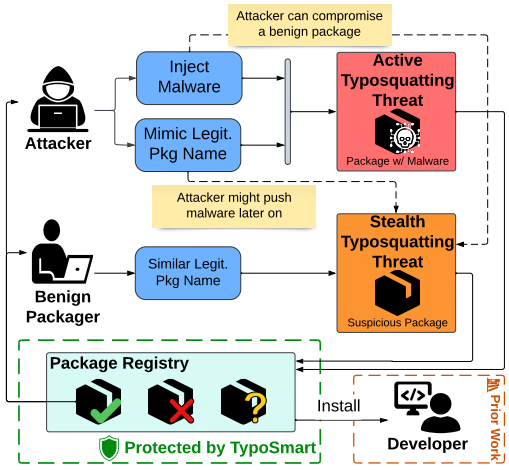


Figure 1: Threat model depicting typical typosquatting attacks involving active and stealth typosquatting threats.

We present `TypoSmart`, the first scalable deployment of a novel typosquatting detection system. Our approach addresses key limitations by leveraging comprehensive package metadata to reduce false-positive rates, integrating enhanced embedding-based name comparisons to scale on production, and deployed to a production environment. Additionally, we refined the traditional definition of typosquatting — moving from merely identifying suspicious package names to incorporating both active and stealth attacks. This approach improved the accuracy of identifying real threats and redefined false-positive criteria, aligning them more closely with practical concerns.

In empirical tests, our `TypoSmart` system improved 73%-91% neighbor search speeds, and reduced false positives by 70.4% through metadata-driven checks. By prioritizing true

positives that practitioners find valuable, our methodology improves the relevance and reliability of detection outcomes. Additionally, this work presents 4 lessons we learned from production deployment from the perspective of SPRs, offering insights into optimal deployment strategies. `TypoSmart` is being used in production at our industry partner, and contributed to the removal of 3,658 typosquatting packages in one month.

To summarize our contributions:

1. Drawing from production deployment experience and analysis of typosquatting-relevant practices, we refined the definition of typosquatting to encompass both active and stealth typosquatting threats.
2. We present `TypoSmart`, an embedding-based detection system, extending state-of-the-art work to SPRs and substantially reducing the false positive rates.
3. Our real-world deployment provides practical lessons for implementing robust supply chain defenses from the perspective of package registries.

2 Background and Related Work

Modern software development depends on ecosystems of third-party libraries and frameworks that facilitate software reuse [61, 65]. We refer to these reusable software artifacts as *software packages*. These packages are distributed by *software package registries (SPRs)* [43, 45]. Table 1 shows the SPRs for six popular ecosystems. Together, these SPRs contain over 7 million software packages that are used by organizations worldwide.

SPRs accelerate engineering practice but their use exposes applications to many software supply chain security attack vectors [11,23,37,39,64] The most common and longstanding supply chain attack vector is called *typosquatting*, or *package confusion*. A recent study indicates that, as of 2024, typosquatting campaigns continue to target developers by exploiting hundreds of popular JavaScript packages, with over 250 typosquatting packages published in total [27]. Previous research has also demonstrated that a single typosquatting package can result in hundreds of user downloads [42]. Our goal is to detect such attacks. In preparation, we next describe the current taxonomy of typosquatting attacks (§2.1) and then examine existing defenses (§2.2).

2.1 Typosquatting Attacks and Taxonomy

Typosquatting attacks on software packages are enabled by permissive package naming policies.[1] In all major SPRs (*e.g.*, those of Table 1), if a name is not already registered, then an attacker can claim it for himself. To carry out a typosquatting attack, an adversary publishes a software package whose name closely resembles that of a legitimate package. The attacker's goal is to cause engineers to accidentally rely on their package instead, allowing the attacker to deliver malware to the developers or users of downstream software [23, 38]. Typosquatting occurs in the AI software supply chain too — on the Hugging Face SPR, attackers publish pre-trained models with names mimicking famous models to deceive engineers into using harmful variants [17, 19, 40].

Table 1: Ecosystems examined in this study, highlighting their primary domains, naming conventions, and popularity metrics. Registry sizes are as of Jan. 2025. An example 1-level naming pattern is PyPI's `requests` module. An example hierarchical name is Hugging Face's `google-bert/bert-base-uncased`.

Registry (# pkgs)	Domain	Name Structure	Pop. metric?
npm (5.1M)	JavaScript	Both	Yes
PyPI (619K)	Python	1-level	Yes
RubyGems (212K)	Ruby	1-level	Yes
Maven (503K)	Java	Hierarchical	External
Golang (175K)	Go	Hierarchical	External
Hugging Face (1.1M)	AI Models	Hierarchical	Yes

Table 2 summarizes the state-of-the-art taxonomy of typosquatting attacks, based on Neupane *et al.* [35]. The variations in naming structure in different SPRs also affect the nature of typosquatting by SPR. In SPRs with 1-level names, typosquats look like the top portion of the table, while in hierarchical SPRs the attacker has a larger naming surface to exploit (bottom portion of table).

Table 2: Common typosquatting techniques with examples. The top section presents the taxonomy proposed by Neupane *et al.* [35]. We added two patterns (bottom, cf. §3.1), generalizing the prior taxonomy to SPRs with hierarchical names.

Technique	Example Transformation
1-step Damerau-Levenshtein Distance	`crypto` → `crypt`
Prefix/suffix augmentation	`dateutil` → **`python3-`**`dateutil`
Sequence reordering	`python-nmap` → **`nmap-python`**
Delimiter modification	`active-support` → `activesupport`
Grammatical substitution	`serialize` → `serialize`**`s`**
Scope confusion	**`@`**`cicada`**`/`**`render` → `cicada-render`
Semantic substitution	`b`**`z2file`** → `b`**`zip`**
Asemantic substitution	`discord.`**`js`** → `discord.`**`app`**
Homophonic similarity	`uglify-js` → `uglifi.js`
Simplification	`pwd`**`hash`** → `pwd`
Alternate spelling	`colorama` → `colo`**`u`**`rama`
Homographic replacement	`d`**`j`**`ango` → `d`**`i`**`ango`
Impersonation Squatting	**`meta`**`-llama/Llama-2-7b-chat-hf` → **`facebook`**`-llama/Llama-2-7b-chat-hf`
Compound Squatting	`@typescript-eslint/eslint`**`-plugin`** → `@typescript-eslint/eslint`**`er`**

[1] We acknowledge that typosquatting plagues all IT endeavors (*e.g.*, DNS [7, 54] and Blocke [33]). We focus on software packages.

2.2 Defenses Against Typosquatting

Typosquat attacks can be mitigated at many points in the reuse process, *e.g.,* by scanning for malware in registries [58] or during dependency installation [24] or by sandboxing dependencies at runtime [4, 8, 57]. Since an ounce of prevention is worth a pound of cure, many works have focused on detecting typosquats prior to dependency installation, *e.g.,* identifying possible typosquats during the dependency selection process. We describe typosquat defenses from academia (§2.2.1) and from industry (§2.2.2), and then describe the weaknesses and knowledge gaps (§2.2.3).

2.2.1 Academic Defenses

Although there are many academic works on software supply chain security [39], there have been relatively few academic works that specifically target the detection of typosquat attacks. Prior typosquatting detection has explored multiple areas, including DNS domains [21, 32], mobile apps [16], blockchain [33], and container registry [25]. The earliest solution to package typosquatting was Taylor *et al.'s* `Spellbound` which was integrated into the installation pipeline to present npm users from potential typosquatting attacks [50]. Vu *et al.'s* concurrent work applied Levenshtein distance to identify PyPI package typosquatting attacks [59]. Their scheme was based on lexical similarity — they detected typosquats by identifying minor textual alterations in package names. To detect more complex attacks, Neupane *et al.* proposed `Typomind`, a system that employs 12 heuristic rules for detecting typosquats [35]. Of particular note is their approach to addressing the "semantic substitution" class of attacks (Table 2). They used FastText embeddings [5] to encode elements of a package name into high-dimensional vectors. By analyzing cosine similarity between these vectors, a wider range of typosquat techniques could be detected.

2.2.2 Industry Defenses

One class of industry tools supports engineers at package installation time. Socket defends against typosquatting by providing real-time detection Stacklok's approach is to identify typosquatting by analyzing package names using Levenshtein distance, evaluating repository and author activity metrics, and assigning a risk score through their platforms [47]. Microsoft's OSSGadget provides a CLI tool to detect typosquats across multiple ecosystems [31].

Some software package registries also seek to detect typosquatting. `npm` uses Levenshtein-based detection to identify and block package names that are deceptively similar to popular packages, thereby preventing typosquats from entering the registry [36]. `PyPI` has an impersonation policy, which prohibits deceptively similar usernames, organization names, and project names, reducing the risk of typosquatting and related attacks [41]. In addition, all major SPRs remove malware (including typosquat packages) when they become aware of it [9, 12, 13]. However, some typosquats are subjective; in the absence of a clear malware signal, human analysts remain necessary to triage reports.

2.2.3 Gap Analysis and Contributions

Our work addresses several gaps across the existing knowledge described in sections 2.2.1 and 2.2.2.

1. **High false positive rate:** Existing tools say that only ∼0.1% of their reports were malware [35] because they do not have a clear definition of typosquatting false-positive. What is the real typosquatting false positive and how to reduce the rate remains an open problem.

2. **Limited registry focus:** So far, the major research papers have focused on only 3 SPRs: npm, PyPI, and RubyGems [35, 51, 59]. As indicated by Table 1, there are many SPRs and these SPRs vary in typosquat-relevant ways — *e.g.,* how names are constructed and whether popularity information is available.

3. **Limited insight from SPR perspective:** Both academic defenses and many industry tools are focused on supporting the individual engineer with engineer-side tooling based on limited local information. Such tooling is certainly useful, but we believe it is a stopgap until SPRs or other ecosystem players can deploy a viable typosquat detection system at scale to proactively prevent typosquat creation or automatically take them down more quickly. False positives are particularly problematic for such a venture because inaccurate notifications erode trust ecosystem-wide. We lack reports from production deployments to understand what challenges arise.

We address these gaps as follows. First, we conduct a novel analysis of typosquat true and false positives in order to improve accuracy over prior work. Second, our work expands on prior work to consider 6 registries, which (to the best of our knowledge) represent all major typosquat-relevant variations across all public SPRs. Third, we have deployed our system in collaboration with an industry partner and share lessons learned.

3 Empirical Study of Typosquats

In this section we report on several complementary investigations into attacker behaviors and benign engineering behaviors. Our findings allow us to refine the definition of typosquatting, extend the state-of-the-art taxonomy, and learn legitimate behaviors that lead to false positives.

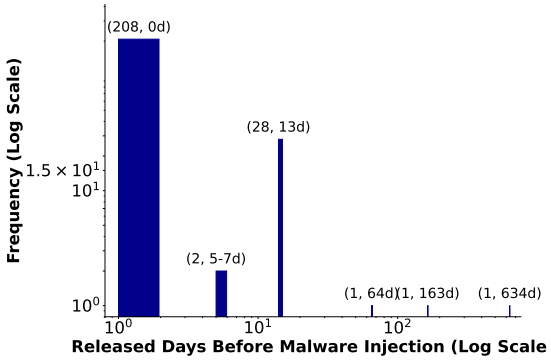


Figure 2: Frequency of released time before malware added to a typosquat package. Over 10% of typosquats are stealthy.

3.1 Analysis of Attacker Practices

Although previous research has proposed various methods for detecting typosquatting attacks, the issue persists across multiple registries. We analyzed attacker practices and identified key shortcomings in prior work.

3.1.1 Stealthy Typosquats

We analyzed the versions of confirmed typosquatting "true-positives", defined as typosquatting attack packages that include malware. These data were collected and reported by Neupane *et al.* [35]. We analyzed the number of days these malicious packages were available before malware was injected. Due to the lack of comprehensive package content (the SPR removed the packages), we estimated the injection time based on the last version updated before removal.

Figure 2 illustrates our findings on the release time before malware injection. **While most typosquatting attacks injected malware within the first day of release, a notable proportion of packages (∼14%) represent stealth attacks**, lying dormant for a period before deploying malware. The stealth typosquatting attack strategy allows attackers to evade detection for extended periods, increasing the likelihood of successful exploitation before the malicious activity is discovered. Prior work on typosquatting did not report this behavior nor take it into account in their system designs.

3.1.2 Extending the Taxonomy to New SPRs

We analyzed the existing taxonomy and real-world attack patterns observed in production data by our industry partner. We found that in registries with hierarchical naming conventions, two additional categories occur:

- **Impersonation Squatting.** Impersonating a legitimate maintainer or organization by registering a deceptively similar `author` or `groupId`. For instance, `meta-llama/Llama-2-7b-chat-hf` vs. `facebook-llama/Llama-2-7b-chat-hf` [40].
- **Compound Squatting.** Making multiple coordinated edits to the hierarchical name, such as altering both scope and delimiters at once. For example, `@typescript-eslint/eslint-plugin` becoming `@typescript_eslinter/eslint`.

3.2 Analysis of Engineering Practices

Prior typosquatting detectors have high false positive rates. To address this, we must know why engineers benignly make packages that resemble typosquats.

Method To understand the causes of such package names, we randomly sampled 665 packages from the original 640,482 false positives reported by Neupane *et al.*, which have suspicious package names while do not include any malware. This sample size was selected to obtain a confidence level of 99% with a margin of error of 5% on the resulting distribution of causes.

To identify the engineering practices in the existing typosquatting data, we began by having two researchers with expertise in the software supply chain analyze 200 of these packages. They analyzed each package's content and metadata (READMEs, maintainers, available source code, etc.). Each analyst independently proposed a list of engineering practices based on this analysis (codebook) and then they discussed this codebook together to reach agreement on terms and definitions. To test valididty, they then independently applied this codebook to the 200 packages and measured agreement using Cohen's Kappa [6]. The initial inter-rater reliability was 0.6 (substantial [29]). The researchers subsequently discussed to resolve discrepancies and refine their analysis. Through discussion, they reached consensus on measurable attributes that could indicate malicious intent or the possibility of a stealthy attack.

Based on the high agreement in this process, one of these researchers analyzed the remaining 465 packages, assisted by a tool developed by both researchers.

Results This analysis identified eight measurable attributes indicative of malicious intent or stealthy attack potential. These attributes include factors such as a distinct purpose, adversarial package naming, and the comprehensiveness of metadata, including README files. Due to space limitations, we have placed the detailed findings in Appendix A.

This manually labeled dataset was also used as the evaluation dataset in §6. We will include the dataset in our artifact (**??**).

3.3 Refined Typosquat Definition

Integrating these findings we obtain:

Definition: Typosquatting Threat

A typosquatting threat is the creation of a software package whose name mimics a popular, trusted resource, with the intent of deceiving developers into installing code that is actively malicious or may become so (stealth).

Under our definition, the current absence of malware does not imply future innocence. In contrast, prior research on typosquatting supposes that typosquat packages always contain malicious code [35, 51].

Without being able to use malware signatures or CVEs to pinpoint active typosquat attacks, our inclusion of intent raises the possibility of high false positive rates. Our analysis of false positives from prior lexical analyses (§3.2) found usable signals to distinguish benign engineering practices from likely-stealthy typosquats.

4 Threat Model and System Requirements

Our goal is to tackle the challenges of high false-positive rates and limited registry coverage while enhancing the ability to detect stealthy typosquatting attempts. This section defines our system and threat model (§4.1), and articulates our system requirements (§4.2).

4.1 System and Threat Model

We describe the system we secure and the threats covered in this work.

System Model: We focus on software package registries, including registries hosting traditional software packages (*e.g.,* NPM) and pretrained AI models (*e.g.,* Huggingface). These registries allow users to publish and share software artifacts (traditional packages and pretrained models) with other users, thereby facilitating software reuse. The number of packages hosted in these registries, as shown in Table 1, demonstrate their popularity across software development communities.

Threat Model: Our threat model focuses on attackers who can publish packages to software package registries and use the published package to deliver malicious code to unsuspecting users and applications. We include some threats and exclude others.

- *In-scope:* We consider attacks where packages with deceptively similar names to legitimate packages are published in the software package registry. The attacker may initially publish with non-malicious content (stealthy) and later introduce malware (active).
- *Out-of-scope:* We do not consider attackers that either directly publish malicious packages with new names or compromise existing legitimate packages. These attacks are mitigated by the broader software supply chain security measures [39].

This threat model is substantially stronger than those of existing typosquat detection techniques [35, 50, 59]. They only consider typosquat packages with malicious code (*i.e.,* active threats, not stealthy ones). Figure 2 demonstrates stealthy threats in the wild, with malicious code introduced months after the original package is published.

4.2 System Requirements

Effective detection systems for package registries must meet both security and operational goals. These requirements ensure robust identification of malicious intent, compatibility with diverse ecosystems, and practical performance for real-world deployment. A production-ready system must address the following key requirements:

$\underline{\text{Req}_1}$: High Accuracy and Low False-Positive Rate A typosquat detection system must achieve high accuracy, effectively capturing true positives while maintaining a low false-negative rate Additionally, to avoid "alert fatigue" [3] and protect the reputation of influential open-source contributors, the system must minimize unnecessary alerts, aiming for a false-positive rate of only a few percent. Packages with similar names, such as legitimate forks [62], should not be flagged unless they exhibit clear signs of malicious intent.

$\underline{\text{Req}_2}$: Efficient and Timely Detection The system must be able to handle large registries, ensuring scalability while providing low-latency, on-demand checks for real-time feedback.

$\underline{\text{Req}_3}$: Compatibility Across Ecosystems The system must provide support for diverse naming schemes, encompassing both one-level and hierarchical naming conventions, along with comprehensive metadata integration.

$\underline{\text{Req}_4}$: Frequent Metadata Updates The system must regularly ingest data from sources like package registries, version histories, and domain checks to maintain up-to-date threat intelligence. This ensures developers are promptly informed about suspicious packages released since the last scan, independent of popularity metrics.

$\underline{\text{Req}_5}$: High Recall of Stealth Typosquatting Attack The system must identify both active and stealth typosquatting packages.

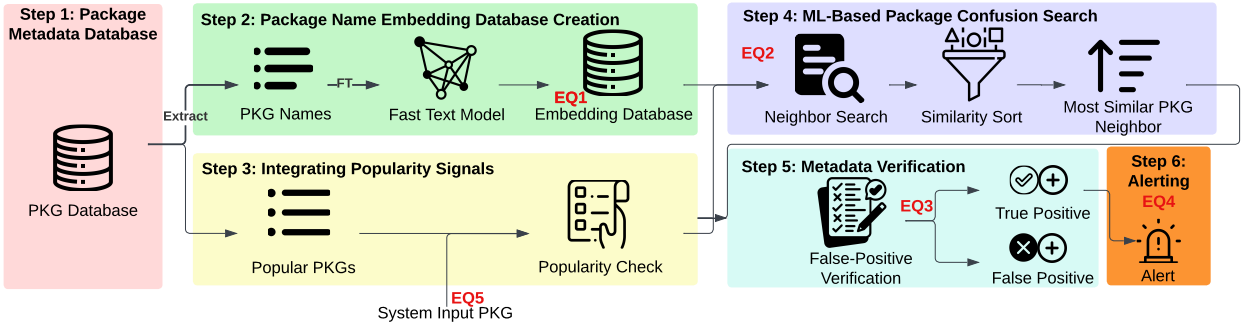


Figure 3: Overview of our typosquatting detection pipeline. The pipeline includes five primary steps, each labeled with a blue circled number. Part of Steps 2 and All Step 3 are also pursued in prior work [35, 50]. In Step 2, we employed an embedding-similarity-based approach, which differs from conventional methods. Additionally, we observed that incorporating Steps 4 and 5 significantly enhances system accuracy. The red circles indicate the evaluation questions (EQs).

5 TypoSmart Design and Implementation

Existing methods often rely on simplistic criteria, such as the absence of current malware activity or name similarity to legitimate packages, to classify packages as benign or potential typosquats (§2.2). These approaches are prone to ambiguity and result in high false-positive rates, failing to account for the range of intents behind suspicious package names [35,51].

We introduce `TypoSmart`, a comprehensive typosquatting detection system designed to reduce false positives while prioritizing the detection of *malicious intent*. Our system integrates embedding-driven name analysis, hierarchical naming checks, and metadata-based verification to enhance threat detection and mitigation. `TypoSmart` is specifically designed for *backend* use in package registries or similar platforms, prioritizing accuracy over low latency to ensure robust and reliable detection.

Our system proactively addresses existing shortcomings by analyzing suspicious package names and leveraging LLMs to detect the existence of malicious intent, thereby flagging harmful naming patterns before they can cause damage.

Figure 3 presents our typosquatting detection pipeline, including five steps:

1. Maintaining an *up-to-date metadata database*, ensuring real-time awareness of new and evolving packages (**Req**$_4$).
2. Generating and storing *fine-tuned embeddings* to capture domain-specific name similarities essential for detecting malicious intent (**Req**$_1$, **Req**$_3$).
3. Integrating *popularity metrics* (where available) so that the system focuses on high-value targets without excluding lower-profile threats (**Req**$_3$).
4. Running *ML-based package confusion searches*, using approximate nearest neighbors (ANN) and quantization to scale efficiently (**Req**$_2$).
5. Verifying *potentially legitimate packages* through metadata heuristics, mitigating false positives and maintaining developer trust (**Req**$_1$, **Req**$_5$).

Each step builds upon the artifacts of the previous ones In the remainder of this section, we first detail the rationale and implementation details of each step (§5.1 – §5.6), as well as the system limitation (§7.2).

5.1 Step ①: Package Metadata Database

5.1.1 Rationale (*Req*$_4$)

Frequent metadata ingestion is essential for early threat detection and aligns with **Req**$_4$, which mandates a reliable feed of newly published packages.

5.1.2 Approach

We employ a *commercially maintained platform* from a software supply chain security company, which consolidates metadata from npm, PyPI, RubyGems, Maven, Golang, and Hugging Face. This database contains package names, version histories, commit logs, license info, and maintainer records, all updated on a near-daily basis. By ingesting these details, we mitigate stale data concerns and ensure our pipeline quickly analyzes newly introduced packages. This step forms the basis for all subsequent steps in the system.

5.2 Step ②: Package Name Embedding Database Creation

5.2.1 Rationale (*Req*$_1$, *Req*$_3$)

Detecting maliciously similar names requires accurately capturing subtle lexical variations. Traditional Levenshtein edit distance methods often fail to account for domain-specific semantic nuances (*e.g.,* `meta-llama` vs. `facebook-llama`), while generic embedding models can introduce inaccuracies for short names, resulting in higher false-positive or false-negative rates. Robust, fine-tuned embeddings address these

shortcomings by providing enhanced semantic sensitivity, reducing erroneous alerts (**Req**$_1$). Moreover, this embedding approach is also generic and unified to support various naming convention of SPRs (**Req**$_3$).

5.2.2 Approach

An *embedding fine-tuned on real package names* enhances semantic sensitivity, enabling the detection of adversarial or suspicious names. This capability is integral to assessing the risk associated with a package and serves as a cornerstone of our *intent-centric* approach.

We build upon FastText [5], starting with the pre-trained model pre-trained on `cc.en.300.bin`, and *fine-tune* it using a list of all (totally ∼9.1 billion) package names extracted from the metadata database in Nov 2024 (§5.1). By fine-tuning, our version can capture:

- **Domain-Specific Subwords**: Frequent domain specific terms in package names are better captured.
- **Hierarchical Structures**: Splitting names in Maven (`groupId:artifactId`) or Hugging Face (`author/model`) to create multiple embeddings per package.

We then use the fine-tuned embedding model to create an embedding vector database utilizing the vector format provided by `pgvector` due to its efficient vector operations for databases [1]. The complete embedding database occupies 24 GB, with each embedding vector corresponding to a single package name (or its `author name` and `package identifier`). This setup not only facilitates rapid query-based lookups but also supports subsequent steps in package neighbor searching. Visualization of our embedding model is available in Appendix B.1.

5.3 Step ③: Integrating Popularity Signals

5.3.1 Rationale (Req$_3$)

Including popularity signals is crucial because attackers often target widely used libraries or models to maximize their impact. Building on insights from prior work [35,51], identifying packages with high download counts or strong community engagement allows us to concentrate detection efforts where they are most needed, minimizing overhead and reducing alert fatigue from infrequently used packages.

5.3.2 Approach

Building on prior work, we prioritize packages based on popularity, treating widely-used ones as more likely to be legitimate, while still scrutinizing all packages for signs of stealth typosquatting attacks to balance resource efficiency and detection accuracy.

For registries like npm, PyPI, RubyGems, and Hugging Face—which offer weekly or monthly download counts. We mark the top packages as *legitimate* packages. Prior works [35, 50] also adopt this strategy, using popularity as a gating mechanism to identify prime attack targets.

Maven and Golang lack consistent download metrics, prompting us to integrate external data sources such as `ecosyste.ms`, which estimate popularity using indicators like stargazers, forks, dependent repositories, and Docker pulls [34]. While the resulting composite `average_ranking` metric is not without limitations, it effectively identifies libraries that are frequently referenced or heavily relied upon.

To address the size of each ecosystem and satisfy Req$_3$, we made the following design decision: For a given input package, we categorize packages into two groups: a popular list and an unpopular list. If the package belongs to the popular list, we compare it only with significantly more popular packages — those with download rates at least 10 times higher (for registries with download metrics) or ranking scores at least twice as high (for registries with relative popularity rankings). If the package is flagged as suspicious, it is removed from the popular list, which is cached in memory for efficient processing. For packages in the unpopular list, we compare them against all packages in the popular list.

5.4 Step ④: ML-Based Package Confusion Search

5.4.1 Rationale (Req$_2$)

Despite the use of robust embeddings (§5.2), scanning entire ecosystems for name-based threats remains computationally intensive. A scalable strategy is essential to identify suspicious package similarities across millions of entries in near-real-time (**Req**2). Moreover, hierarchical naming conventions (*e.g.,* `groupId:artifactId` in Maven) add complexity, as attackers may target authorship (*i.e., author squatting*) rather than solely manipulating package names. Thus, the solution must achieve high recall while maintaining efficiency and minimizing false positives, ensuring developers are not overwhelmed with excessive alerts (**Req**$_2$).

5.4.2 Approach

Package Name Similarity Search To efficiently detect suspiciously similar names, we employ the *Approximate Nearest Neighbors* (ANN) method, utilizing the *HNSW* index [28] in PostgreSQL. We selected HNSW over *IVFFlat* due to its superior benchmarking results, which demonstrate faster search speeds and negligible changes to vectors [52]. The HNSW index partitions the embedding space into multiple clusters, enabling us to focus distance computations on a smaller subset of candidate packages, rather than exhaustively comparing all possible pairs.

Data-Driven Optimization We deployed an initial version of our neighbor search algorithm to production for evaluation. Based on empirical data, we identified opportunities to enhance its performance. To improve accuracy across ecosystems, we tailored the neighbor search query. For example, in Maven, the `artifact _id`, and in Golang, the `domain name`, are lengthy and reduce the effectiveness of embedding similarity. To address this, we relied exclusively on edit distance for author names. Additionally, within our threat model, we determined that changes to both identifiers in Maven, Golang, and Hugging Face were unlikely to significantly confuse users. Consequently, we excluded these from our query.

Quantization for Performance. While ANN indexing accelerates queries, the high dimensionality of fine-tuned FastText vectors can still impose significant computational and storage demands. To address this, we applied the quantization technique to compress model weights by reducing precision, and maintaining high-quality embeddings while reducing model size. We then regenerate and store all package name vectors using the quantized model in PostgreSQL, enhancing both batch and real-time search performance while contributing to **Req**$_3$ by lowering compute costs.

HuggingFace challenges:semantic difference vs. typos

5.5 Step ⑤: Metadata Verification

Despite promising performance in name-based detection, purely string-oriented methods often misclassify harmless or beneficial packages as typosquats. To mitigate these false alarms, we developed a **metadata-driven verification procedure** that filters out legitimate packages based on readily observable heuristics (*e.g.,* active development, overlap in maintainers). This section details our preliminary analysis, the rules derived from it, and how we evaluate the final false-positive (FP) verifier.

5.5.1 Rationale (Req$_1$, Req$_5$).

Although name-based detection (Steps 2–4) successfully flags suspiciously similar packages, purely string-oriented methods often over-trigger on benign forks, relocated projects, or rebranded namespaces. This can overload developers with alerts, undermining trust in the system (**Req**$_1$, which calls for manageable false-positive rates). To mitigate these spurious detections, we introduce a metadata-driven verification procedure that check each flagged package for clear signs of legitimacy (*e.g.,* active development, overlapping maintainers) and classify them into different categories (**Req**$_5$).

5.5.2 Approach

Our analysis on the false-positive data (§3.2) suggested that purely name-based detection can flag numerous legitimate packages, from brand extensions to harmless forks. To address these issues, we iteratively developed nine rules that utilize metadata signals (*e.g.,* version history, maintainers) to distinguish genuinely malicious typosquats from benign look-alikes.

Table 3 provides a summary of the goals and implementation details for each rule. Once our name-based detector flags a package as suspicious, we retrieve its metadata and sequentially apply `R1`–`R8`, stopping as soon as a rule determines the package to be legitimate. Furthermore, during the deployment of our system in production, we identified the need for two additional rules (`R9`, `R10`) based on observed patterns. These new rules were subsequently integrated into the system, along with their corresponding metrics. Implementation detail of our verifier can be found in Appendix B.3.

5.6 Step ⑥: Intervention and Alerting

To mitigate typosquatting attacks, our system issues notifications to relevant stakeholders whenever it detects suspicious packages across different package registries. The goal is to prompt closer scrutiny of packages that exhibit potentially deceptive naming, to determine whether they constitute genuine attacks, and to assess any malicious behavior within them. We envision this alert mechanism as an additional layer of defense, complementing other software-package metrics such as supply chain security, quality, maintenance, vulnerability management, and licensing compliance.

5.7 Implementation

Our system is implemented in Python, with embeddings stored in PostgreSQL. Quantization and embedding generation rely on a modified FastText pipeline (§5.2). `TypoSmart` has over 3K `LOC`, primarily comprising ∼1K `LOC` for embedding creation, ∼500 `LOC` for popularity check, ∼1K `LOC` for neighbor search, ∼300 `LOC` for metadata verification, and ∼100 `LOC` for alerting.

6 Evaluation

To evaluate `TypoSmart`, we pose five Evaluation Questions (EQs) to assess its performance at the component level (**EQ1**–**EQ3**) and the system level (**EQ4**–**EQ5**). At the component level, we measure the effectiveness of novel mechanisms introduced in `TypoSmart`. At the system level, we evaluate its integrated functionality, scalability, and ability to detect real-world typosquatting threats. Additionally, we compare our approach to SOTA methods to benchmark its effectiveness. An overview of the evaluation process is illustrated in Figure 3.

Component-level We assess how individual components contribute effectively to the overall system.

Table 3: Overview of the 10 metadata-based verification rules. Each rule includes a description of its purpose and the specific implementation steps taken to verify flagged packages. The final two rules (R9 and R10) were added as part of our refinement process based on further observed false-positive patterns after we deployed our system in production.

Rule	Description	Implementation
R1: Intentional Naming	Identify brand-related or deliberately extended names, such as `express-plus`, that suggest a legitimate project rather than a malicious clone.	Compare flagged package names to legitimate ones, searching for suffixes like `-plus`, `-extra`, or `-utils`. Presence of these terms strongly suggests legitimate extensions.
R2: Distinct Purpose	Distinguish packages with different functionalities, even if names are superficially similar (*e.g.,* `lodash-utils` vs. `lodash`).	Extract package descriptions and calculate semantic similarity using TF-IDF cosine scores. A score below 0.5 indicates distinct purposes, reducing suspicion of deception.
R3: Fork Identification	Detect benign forks sharing near-identical code or metadata with a popular package.	Compare README files, version histories, and file structures for high overlap. Similarities without malicious edits suggest harmless forks.
R4: Active Development/Maintained	Exclude packages that are frequently updated or actively maintained by multiple contributors, as these are less likely to be malicious placeholders.	Retrieve metadata for the last update, commit history, and version count. Classify packages with recent updates (*e.g.,* within 30 days) or more than five versions as legitimate.
R5: Comprehensive Metadata	Identify packages missing critical metadata elements, such as licenses, maintainers, or homepage URLs, which are typical of legitimate projects.	Check for the presence of licenses, contact details, and repository links. Flag packages missing two or more of these elements as potential typosquats.
R6: Overlapped Maintainers	Distinguish legitimate extensions or rebrands by verifying if the flagged package shares maintainers with the legitimate one.	Match maintainer identifiers (*e.g.,* email, GitHub handle) between flagged and legitimate packages. Overlapping maintainers suggest legitimate intent.
R7: Adversarial Package Name	Filter out name pairs with significant length differences, as these often indicate unrelated projects rather than covert mimicry.	Compare string lengths of flagged and legitimate package names. A difference exceeding 30% indicates likely unrelated naming.
R8: Well-known Maintainers	Trust packages maintained by reputable and recognized authors or organizations.	Leverage knowledge in LLM training data to identify if a maintainer is well-known in the community.
R9: Package Relocation	Account for legitimate package relocations, common in hierarchical registries like Maven.	Parse metadata files (`pom.xml`) for `<relocation>` tags or analogous fields. Identify and ignore renamed or migrated projects.
R10: Organization Allowed List	Prevent false positives by excluding packages published by trusted or verified organizations.	Maintain an allowedlist of approved organizations. If a flagged package is published under an allowed organization (*e.g.,* `@oxc-parser/binding-darwin-arm64`), it should be considered legitimate, comparing to `binding-darwin-arm64`.

- **EQ1: Performance of Embedding Model.** What is the accuracy and efficiency of our embedding model?
- **EQ2: Neighbor Search Accuracy.** How effective is the neighbor search approach? (§5.4)
- **EQ3: Metadata Verification Accuracy.** How much does the FP verifier (§5.5) reduce false positive rates?

System-level We examine the performance of the full `TypoSmart` system and compare to other approaches.

- **EQ4: Discovery of New Typosquats.** Can `TypoSmart` identify previously unknown typosquatting threats?
- **EQ5: System Efficiency and Scalability.** Does `TypoSmart` have practical throughput and latency for deployment? (Throughput is critical for full registry scans, while low latency ensures viability for on-demand API queries.)

All experiments run on a server with 36 CPU cores (Intel Xeon W-2295 @ 3.00GHz) and 188 GB of RAM. Notably, the training and fine-tuning of FastText models do not require GPUs.

6.1 Baseline and Evaluation Datasets

State-of-the-Art Baselines We compare our system to the Levenshtein distance approach, used in [59], and Typomind [35] which is the only embedding-based approach from existing literature.

Evaluation Datasets We evaluate our approach primarily using the dataset from [35] because it summarized all confirmed typosquatting true positives, while include many suspicious packages that does not include malware. This dataset includes 1,239 confirmed typosquats labeled typosquatting packages from npm, PyPI, and RubyGems.

6.2 EQ1: Perf. of Embedding Model

EQ1 Summary: Our embedding model demonstrates superior effectiveness and efficiency, ensuring an acceptable overhead for SPR during the preparation of the embedding database.

In this evaluation question, we measured both the effectiveness and efficiency of our embedding model.

6.2.1 Effectiveness

Method We evaluate the effectiveness of embedding-based similarity detection by comparing three approaches:

1. *Levenshtein-Distance*, calculates the minimum number of single-character edits required to change one package name into another.
2. *Pre-trained FastText* [5] (`cc.en.300.bin`), used in the SOTA work Typomind [35], employs the general-purpose embedding model `cc.en.300.bin` [5] to capture semantic relationships.

3. *Fine-tuned FastText (**Ours**)*, which we have adapted using a corpus specifically composed of package names to better capture domain-specific similarities.

To systematically compare these methods, we construct a balanced test set consisting of both positive and negative pairs, with each category containing 1,239 data points.

- *Positive Pairs*: These are derived directly from the Typomind Ground-Truth dataset [35], which includes labeled typosquatting packages across npm, PyPI, and RubyGems. Each positive pair consists of a known typosquat and its corresponding legitimate package.
- *Negative Pairs*: Created by randomly pairing unrelated package names from registries, ensuring low similarity scores across all tested methods. These pairs are guaranteed not to represent typosquatting relationships.

For each approach, we applied a predefined similarity threshold to classify package pairs as potential typosquats. The threshold was selected via a grid search to optimize Precision and Recall, ensuring effective identification of true typosquats. Pairs with similarity scores above the threshold were classified as positive, while those below were classified as negative. False positives were subsequently filtered using our false-positive verification process in Step ⑤.

We compute Precision, Recall, and F1 scores for each approach to assess their performance:

- *Precision*: The proportion of correctly identified typosquats out of all flagged pairs.
- *Recall*: The proportion of actual typosquats that were correctly identified.
- *F1 Score*: The mean of Precision and Recall, providing a balanced performance metric.

Result Table 4 presents the results comparing our fine-tuned model with baseline approaches. The table shows that the accuracy of the quantized and original versions of our fine-tuned model is nearly identical, indicating minimal loss in performance from quantization. Additionally, the overall F1 score of the fine-tuned model is approximately 1% higher than the pre-trained model, with a notable 2-3% improvement in recall for positive pairs. This improvement in recall is particularly critical for detecting typosquatting attacks, as correctly identifying all positive pairs is essential for comprehensive threat mitigation. At the same time, our approach maintains a relatively high accuracy for negative pairs.

6.2.2 Efficiency

Method To assess the efficiency of embedding database creation, we evaluated three quantized versions of our fine-tuned models. We evaluated the embedding database creation efficiency by measuring the throughput, latency, and overall overhead associated.

Result Our results shows that quantized models substantially increase throughput while decreasing latency. Specifically, both `float16` and `int8` quantization methods enhance throughput by approximately 5× and reduce latency by around 10× compared to the `float32` baseline. However, the performance difference between `float16` and `int8` is minimal, likely due to the additional quantization and processing steps involved.

Overall, the efficiency of quantized models indicates that our embedding models achieve strong performance while optimizing resource usage. Additionally, the implementation of *HNSW* indexing introduces minimal overhead, with each table requiring less than 10 seconds to process. This further enhances overall system efficiency. A detailed results table is available in Table 7 (Appendix B.2).

6.3 EQ2: Neighbor Search Accuracy

EQ2 Summary: Our embedding-based neighbor search algorithm, combined with the sorting algorithm, achieves accuracy comparable to prior work while capturing richer semantic information.

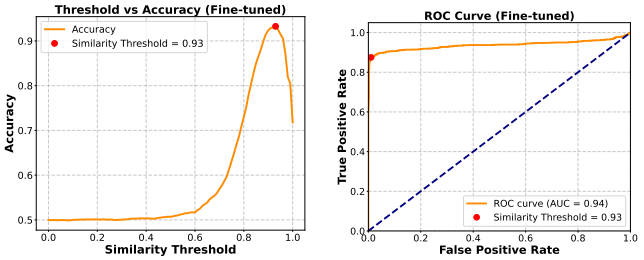
Methods Using the results from EQ1, we determined a threshold of 0.93 through grid search, achieving an optimal balance between precision and recall. To evaluate neighbor search performance, we analyzed suspicious packages identified by `typomind` and additional data provided by our ry partner.

Results Figure 4 illustrates the ROC and accuracy curves for our fine-tuned model. Our approach successfully captured all typosquatting cases previously identified by SOTA methods. The neighbor search algorithm demonstrated the ability to detect 99% of real typosquatting attacks flagged by prior research.

Deployment data revealed that the algorithm effectively identified advanced threats, including compound squatting and impersonation squatting, across platforms such as Maven, Golang, and Hugging Face. For example, our system detected a *compound squatting* attack on `@typescripyt-eslint/eslint-plugin`, where the attacker used both a similar namespace and package identifier (`@typescript_eslinter/eslint`) to mislead users. We found that prior tools were unable to identify this type of sophisticated attack. Additionally, we successfully flagged impersonation squatting attacks targeting hierarchical package names in npm. While no suspicious packages were identified in Golang, Maven, or Hugging Face due to the limited sample

Table 4: Performance of embedding models. We selected the fine-tuned model with a threshold of 0.9 to achieve high F1 scores for positive pairs and relatively high F1 scores for negative pairs. Quantization slightly improves F1 scores because the false positive rate goes up a bit while the false negative rate goes down more substantially.

Model	Quantization	Threshold	Positive Pairs			Negative Pairs			Overall Score
			Precision	Recall	F1 Score	Precision	Recall	F1 Score	
Edit Distance	N/A	4	1.00	0.80	0.89	1.00	1.00	1.00	0.94
Pretrained	float32	0.6	1.00	0.85	0.92	1.00	0.98	0.99	0.96
Pretrained	float16	0.6	1.00	0.85	0.92	1.00	0.98	0.99	0.96
Pretrained	int8	0.6	1.00	0.85	0.92	1.00	0.98	0.99	0.95
Fine-Tuned	float32	0.9	1.00	0.90	0.95	1.00	0.96	0.98	0.96
Fine-Tuned	float16	0.9	1.00	0.90	0.95	1.00	0.96	0.98	0.96
Fine-Tuned	int8	0.9	1.00	0.88	0.94	1.00	0.96	0.98	0.96
Hybrid	N/A	0.5	1.00	0.91	0.95	1.00	0.91	0.95	0.95



(a) (b)

Figure 4: Performance Metrics of the Fine-Tuned Model: (a) Threshold Accuracy and (b) ROC Curve.

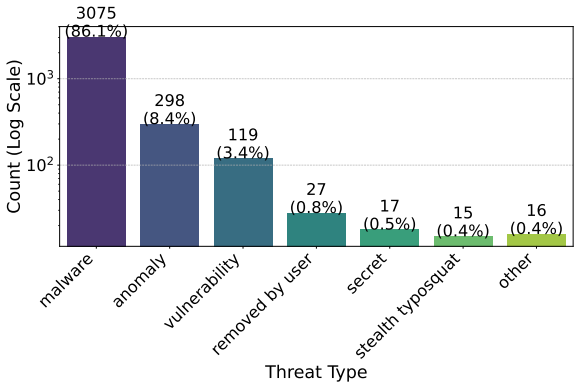


Figure 5: Production data from npm and PyPI showing removed packages flagged by our system within the past month.

size in these registries, we are confident in the system's capability to detect such threats. Notably, it can flag the reported impersonation squatting attacks for Hugging Face from [40], which prior work failed to identify. These findings highlight that our approach achieves SOTA performance in neighbor search accuracy.

6.4 EQ3: Metadata Verification Accuracy

EQ3 Summary: Our metadata verification process significantly lowers the false-positive rate from 75.4% to 5%.

Method Starting with the raw embedding output (*i.e.,* packages flagged for name-based suspicion), we apply the nine metadata checks described in §5.5. We measure false positive rates (FPR) on the manually labeled data, *i.e.,* 665 flagged packages verified in §3.2.

Result Among the 665 false-positive (*i.e.,* suspicious typosquatting package without malware injected), there are 23 packages unavailable. So among the rest 642 packages, our metadata verification step was able to correctly classify 425 as false-positive, *i.e.,* we reduced the false-positive rate from 75.4% to 5%.

These results confirm that supplementary heuristics beyond raw name similarity reduce false positives while retaining high recall for genuinely malicious typosquats. For instance, overlapping maintainer checks (**R6**) and explicit relocation detection (**R9**) proved especially effective in ecosystems like Maven, where hierarchical naming changes are common. Ultimately, this metadata-driven filtering step (**Step 4** in our pipeline) aligns our detection approach with the realities of open-source development and AI model publication, ensuring minimal "false alarms" (Req_3) without missing critical threats.

6.5 EQ4: Discovery of New Typosquats

EQ4 Summary: Our system effectively detected stealthy typosquatting attacks with malicious intent and identified two new attack types—*impersonation squatting* and *compound squatting*—in a production environment. Moreover, it uncovered these threats across three ecosystems that had not been explored in prior research.

Method To evaluate the effectiveness of our typosquatting detection system, we deployed `TypoSmart` in a production environment for one month. During this period, flagged packages were analyzed using a malware scanner and reviewed by

threat analysts for detailed insights.

Result Figure 5 provides a detailed breakdown of typosquatting packages detected by `TypoSmart` which were flagged and removed from registries in Dec, 2024.

Out of 3,658 suspicious package names flagged by our system, the majority (3,075, 86.1%) contained malware. An additional 298 packages (8.4%) were classified as anomalies due to the presence of dangerous functionality (*e.g.,* `eval`, `fs`) that did not appropriately handle user input or exhibited other risky coding practices. A smaller subset, 119 packages (3.4%), was categorized as "vulnerabilities", where the code included multiple dangerous functionalities and mishandled user input in ways that could harm users' systems, such as deleting files outside the `tmp` folder. Furthermore, 27 packages were removed from the SPRs by their authors or maintainers before further analysis, leaving insufficient data to assess their content. In addition, we identified 15 stealth typosquatting threats (0.4%), which did not contain overtly malicious content but were flagged due to their deceptive intent. These packages impersonated legitimate ones to mislead users. Other less common categories, such as spyware-only attacks, were also observed in the dataset.

In addition to these packages that include

Table 5: Typosquats uncovered in randomly selected 5000 packages from each SPR.

Registry	Suspicious	Benign
npm	67	4933
PyPI	29	4971
RubyGem	239	4761
Maven	0	10000
Golang	0	10000
Hugging Face	0	10000

6.6 EQ5: System Efficiency and Scalability

EQ5 Summary: Our system exhibits slower processing speeds due to additional sorting and LLM verification steps but maintains acceptable efficiency and strong scalability for deployment on SPRs.

Method We evaluated the end-to-end system efficiency by measuring the average latency and throughput across 5,000 package inputs per ecosystem. The latency metrics include similarity comparison, neighbor search, false-positive (FP) verification, and overall system latency. Throughput was calculated as the number of packages processed per second.

Result Table 6 summarizes the performance metrics of our system across various registries, demonstrating acceptable latency with variations based on registry size and complexity. For example, npm shows a latency of 14.17 seconds per package, while PyPI achieves a faster latency of 7.22 seconds. RubyGems, with its larger package set and more complex naming structures, has a higher latency of 46.94 seconds per package. Our system also effectively supports previously unaddressed registries like Maven, Golang, and Hugging Face, which exhibit latencies of 5.98, 5.04, and 16.05 seconds per package, respectively. Importantly, it achieves substantial speed improvements in the neighbor search step, outperforming prior work by 73%–91% across all registries.

Although our system prioritizes accuracy over latency, it maintains performance levels suitable for production deployment. The relatively low throughput is primarily due to reliance on the OpenAI API and GPT-4o for metadata verification, which introduces significant inference times (*e.g.,* npm and PyPI require 11.3 and 6.81 seconds per package, respectively, for false-positive verification). However, the inclusion of LLMs greatly enhances detection accuracy, making this trade-off worthwhile. The additional latency from sorting and LLM verification is critical to reducing false positives and ensuring high detection accuracy. Furthermore, the system scales effectively, efficiently managing large registries like npm and PyPI. Future work could explore optimizing LLM inference or incorporating smaller, more efficient models to boost throughput without compromising accuracy.

7 Discussion

7.1 Lessons Learned from Production

Lesson 1: `TypoSmart` prevents real typosquats Over the past two months, we deployed the system in our industry partner's production environment, during which we identified and confirmed 2,153 threats, with 56,025 additional threats under review. For the typosquatting threats we identified on npm, PyPI, and Golang, we ran a commercial malware scanner on those packages. This results in the removal of 3,658 packages within one month.

Lesson 2: False Positives Harm Our Customers. Table 3 represents the most recent metadata verification rules. In deployment, we found additional cases where the package has a very similar name and the README of their package was missing which made our system classify it as suspicious stealth attack, while the package is actually legitimate. That package, specifically served as transitive package and therefore the package owner did not write a README. In such case, from a registry perspective, a false-positive will harm the reputation of the customer in the community if the package is flagged as suspicious. To avoid this case, we added an allowed list so that if the package is from our customers, then it should be considered as legitimate.

Lesson 3: Ontology Matters. Despite these improvements, we have identified ontological limitations in our current ap-

Table 6: System Latency and Throughput Metrics Across Registries: This comparison includes similarity latency, overall neighbor search latency, false-positive verification latency, and system latency. All metrics are evaluated using identical inputs, ensuring fair comparability across systems. Each latency value represents the time taken to compare a given unpopular package with all popular packages in the registry. Prior work does not support Maven, Golang, and Hugging Face because their names are too long to process [35]. The FP verification for Maven, Golang, and Hugging Face was manually triggered (no suspicious neighbors identified).

Registry	# Pop. Pkgs.	*Typomind* Name Comparison Latency (s/pkg)	*Our* Neighbor Search Latency (s/pkg)	FP Verification Latency (s/pkg)	System Latency (s/pkg)
npm	36,333	10.83	2.87	11.3	14.17
PyPI	9,525	4.45	0.41	6.67	8.94
RubyGems	60,695	34.62	1.37	15.73	23.98
Maven	27,831	N/A	5.98	(8.08)	(14.06)
Golang	20,713	N/A	5.04	(12.04)	(17.08)
Hugging Face	13,252	N/A	1.23	(14.82)	(16.05)

proach to categorizing typosquatting threats. Specifically, the typosquatting categories outlined in Table 2 do not provide sufficient meaningfulness for end-users in the alarming system (§5.6). To address this, we propose refining our categorization system to consider the malicious content and intent behind each package, including a risk-level classification. This would allow a more nuanced assessment of each threat.

Lesson 4: We need more sophisticated analysis to discern intent. To bolster a more straightforward blocking policy, there is still a need to integrate additional tools and algorithms. One such tool is a differential scanner, which compares packages to identify discrepancies that may indicate malicious alterations. Additionally, implementing a grey-list system will enable us to place suspicious packages with low-risk levels into a monitored category. These grey-listed packages will undergo continuous surveillance of their behavior and content, ensuring that any emerging threats are promptly detected and addressed. By adopting these strategies, we aim to enhance the robustness and responsiveness of our typosquatting defense mechanisms.

7.2 Limitations and Security Analysis

This section discusses our system limitations and how attackers might bypass `TypoSmart`.

Gaming Metrics Our system relies on software metrics to gauge the likelihood that a package is a typosquat. These metrics might be gamed. There has been little formal study of the feasibility of gaming these metrics, but recent work suggests both the possibility and some real-world examples [15].

Limitations in Neighbor Search One significant limitation of `TypoSmart` lies in its ability to handle short names or acronyms. FastText, the underlying embedding model, struggles with short words (*e.g.,* `xml` vs. `yml`, with a similarity score of 0.7). The model's reliance on character *n*-grams often fails to capture subtle similarities effectively in such cases, providing an avenue for attackers to exploit short package names. Although FastText emphasizes subword representations to improve embeddings, this focus reduces its ability to account for visual ambiguity or phonetic similarity (*e.g.,* `google` and `g00gle` appear less similar in the embedding space). To address this, we implemented a list of potential substitutions to identify cases of visual or phonetic ambiguity. However, this approach introduces additional computational overhead, slowing down the system. To further enhance detection accuracy, we combine embedding similarity with Levenshtein distance. While this hybrid approach improves neighbor search for short names, it increases the computational cost and still does not fully resolve the limitations in representation. Recent research demonstrates the potential of AI tools to generate sound-based squatting attacks [55, 56], further exacerbating threats to the software supply chain.

Bypassing Metadata Verification Our reliance on OpenAI's `GPT-4o` API introduces obvious correctness issues (hallucination). It also entails security risks, as these models are vulnerable to jailbreaking attacks [63]. Adversaries could exploit this by using techniques like prompt injection or model hijacking [26, 66] to manipulate the model's responses. For example, attackers might craft prompts or metadata to trick the model into classifying a malicious package as legitimate, such as misrepresenting a malicious username as belonging to a trusted maintainer or aligning fake functionality with that of a legitimate package. Since the LLM is integral to verifying maintainers and functionality, such attacks could compromise the system's defenses and allow malicious packages to bypass detection.

7.3 Future Directions

Addressing Other Squatting Attacks: Our study focused on typosquatting on package names, but other constructs within SPRs and ecosystems are also viable targets. One example is *command squatting*, where malicious packages mimic command-line arguments (*e.g.,* `npm i help` vs. `npm i --help`). While creating static command lists for each registry would offer a temporary solution, maintaining them would be complex in these decentralized ecosystems.

Enhancing Representations for Typosquatting Detection: Improving the representation of typosquats is crucial for more robust detection. While FastText captures semantic similarity through subword embeddings, it struggles with typographical variations, particularly for short words or acronyms. Fine-tuning FastText or training a more efficient model on domain-specific corpora including both correct and misspelled terms can address these limitations. Augmenting training data with synthetic typos and incorporating typo normalization or correction techniques before embedding generation can significantly reduce errors. Advanced models, such as transformer-based architectures fine-tuned with contrastive learning on typo-specific datasets, present a promising alternative for enhancing detection accuracy and reliability. This approach has proven effective in combating domain typosquatting, but no research has been conducted targeting package typosquatting [22]. One challenge is the limited availability of data for verifying package squatting cases. LLMs might help here [53].

Mitigating LLM Hallucination in Code Generation The increasing use of LLMs for code generation has introduced new challenges, as these models often hallucinate package names or generate commands for nonexistent or maliciously similar packages [46]. These hallucinations pose serious threats to the security of the software supply chain [53, 60]. Addressing this issue requires implementing typo or hallucination correction mechanisms in LLM-based package recommenders. Verifying package legitimacy, detecting typos, and integrating contextual checks can prevent the propagation of incorrect package names, reducing the risks associated with hallucinations.

Meta-Learning for Malicious Package Detection Meta-learning approaches offer significant potential for improving malicious package detection. By leveraging anomaly detection techniques and metadata analysis [14], systems can dynamically adapt to evolving attack strategies. Meta-learning frameworks could analyze patterns across registries and rapidly identify emerging threats, enhancing the scalability and robustness of detection systems. Integrating such frameworks will be key to staying ahead of increasingly sophisticated attackers.

8 Conclusion

We present `TypoSmart`, an embedding-based typosquatting detection system. Based on real-world attack patterns, we refined the typosquatting definition and developed a taxonomy based on engineering practices. `TypoSmart` is being used in production at our industry partner and contributed to the removal of 3,658 typosquatting threats in one month. Compared to SOTA methods, our system is good at capturing additional typosquatting categories, achieves a substantially lower false-positive rate, and maintains acceptable latency, making it well-suited for deployment on SPR backends while remaining effective for frontend on-demand requests. We shared our insights from production experience, customer feedback, the need for an improved ontology, and outlined future directions.

Acknowledgments

Part of this work was completed by Wenxin Jiang while interning at Socket. This work was funded in part by a gift from Socket.

References

[1] pgvector: A vector extension for postgresql. https://github.com/pgvector/pgvector, 2023.

[2] General Services Administration (GSA) 18F. 18f open source policy. https://18f.gsa.gov/open-source-policy/, 2025.

[3] Agency for Healthcare Research and Quality. Alert fatigue. https://psnet.ahrq.gov/primer/alert-fatigue, 2019.

[4] Paschal C Amusuo, Kyle A Robinson, Tanmay Singla, Huiyun Peng, Aravind Machiry, Santiago Torres-Arias, Laurent Simon, and James C Davis. ZTDJAVA: Mitigating software supply chain vulnerabilities via zero-trust dependencies. In *ICSE*, 2025.

[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[6] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[7] Conor Coyle. How google chrome can identify typos in urls to keep you safe online, 2023.

[8] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kastner. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE, May 2021. Place: Madrid, ES.

[9] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *ICSE*, 2021.

[10] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. Investigating package related security threats in software registries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1578–1595. IEEE, 2023.

[11] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. Investigating package related security threats in software registries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1578–1595. IEEE, 2023.

[12] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. Investigating package related security threats in software registries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1578–1595. IEEE, 2023.

[13] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023.

[14] 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 on Web Conference (WWW'24)*, pages 1779–1789, 2024.

[15] Hao He, Haoqin Yang, Philipp Burckhardt, Alexandros Kapravelos, Bogdan Vasilescu, and Christian Kästner. 4.5 million (suspected) fake stars in github: A growing spiral of popularity contests, scams, and malware. *arXiv:2412.13459*, 2024.

[16] Yangyu Hu, Haoyu Wang, Ren He, Li Li, Gareth Tyson, Ignacio Castro, Yao Guo, Lei Wu, and Guoai Xu. Mobile app squatting. In *Proceedings of The Web Conference 2020*, pages 1727–1738, 2020.

[17] Wenxin Jiang, Chingwo Cheung, George K Thiruvathukal, and James C Davis. Exploring naming conventions (and defects) of pre-trained deep learning models in hugging face and other model hubs. *arXiv:2310.01642*, 2023.

[18] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R. Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[19] Wenxin Jiang, Nicholas Synovic, Rohan Sethi, Aryan Indarapu, Matt Hyatt, Taylor R. Schorlemmer, George K. Thiruvathukal, and James C. Davis. An empirical study of artifacts and security risks in the pre-trained model supply chain. In *ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED'22)*, page 105–114, 2022.

[20] Berkay Kaplan and Jingyu Qian. A Survey on Common Threats in npm and PyPi Registries, August 2021. arXiv:2108.09576 [cs].

[21] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero-Gómez, Nikolaos

Pitropakis, Nick Nikiforakis, and Manos Antonakakis. Hiding in plain sight: A longitudinal study of combosquatting abuse. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 569–586, 2017.

[22] Takashi Koide, Naoki Fukushi, Hiroki Nakano, and Daiki Chiba. Phishreplicant: A language model-based approach to detect generated squatting domain names. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 1–13, 2023.

[23] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526, May 2023.

[24] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[25] Guannan Liu, Xing Gao, Haining Wang, and Kun Sun. Exploring the unchartered space of container registry typosquatting. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 35–51, 2022.

[26] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, et al. Prompt injection attack against llm-integrated applications. *arXiv:2306.05499*, 2023.

[27] Jessica Lyons. Ongoing typosquatting campaign impersonates hundreds of popular npm packages. https://www.theregister.com/2024/11/05/typosquatting_npm_campaign/, November 2024.

[28] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[29] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[30] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv:1802.03426*, 2018.

[31] Microsoft. Microsoft ossgadget. https://github.com/microsoft/OSSGadget.

[32] Abdallah Moubayed, MohammadNoor Injadat, Abdallah Shami, and Hanan Lutfiyya. Dns typo-squatting domain detection: A data analytics & machine learning based approach. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[33] Muhammad Muzammil, Zhengyu Wu, Lalith Harisha, Brian Kondracki, and Nick Nikiforakis. Typosquatting 3.0: Characterizing squatting in blockchain naming systems. *arXiv:2411.00352*, 2024.

[34] Andrew Nesbitt. Ecosyste.ms database: A comprehensive dataset for software ecosystem analysis, 2025.

[35] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. Beyond typosquatting: an in-depth look at package confusion. In *USENIX Security '23*, pages 3439–3456, 2023.

[36] npm, Inc. Threats and mitigations. https://docs.npmjs.com/threats-and-mitigations#by-typosquatting--dependency-confusion, 2024.

[37] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, Cham, 2020. Springer International Publishing.

[38] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2020.

[39] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. SoK: Analysis of Software Supply Chain Security by Establishing Secure Design Properties. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 15–24, Los Angeles CA USA, November 2022. ACM.

[40] ProtectAI. Unveiling ai supply chain attacks on hugging face, n.d. Accessed: 2025-01-15.

[41] Python Software Foundation. Acceptable use policy. https://policies.python.org/pypi.org/Acceptable-Use-Policy/, 2024.

[42] ReversingLabs. R77 rootkit: Typosquatting attack in npm ecosystem, 2023.

[43] Taylor R Schorlemmer, Kelechi G Kalu, Luke Chigges, Kyung Myung Ko, et al. Signing in four public software package registries: Quantity, quality, and influencing factors, 2024.

[44] Lee Joon Sern and Yam Gui Peng David. Typoswype: An imaging approach to detect typo-squatting. In *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2021.

[45] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The Emergence of Software Diversity in Maven Central. In *International Conference on Mining Software Repositories (MSR)*, 2019.

[46] Joseph Spracklen, Raveen Wijewickrama, AHM Sakib, Anindya Maiti, and Murtuza Jadliwala. We have a package for you! a comprehensive analysis of package hallucinations by code generating llms. *arXiv:2406.10279*, 2024.

[47] Stacklok. Detecting typosquatting attacks on open source packages using levenshtein distance and activity data. https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html, n.d.

[48] John Stenbit. Open source software (oss) in the department of defense (dod). *Department of Defense, Memorandum*, 2003.

[49] Synopsys, Inc. 2024 open source security and risk analysis report. https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html, 2024.

[50] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Defending against package typosquatting. In *Network and System Security: 14th International Conference (NSS)*.

[51] Matthew Taylor, Ruturaj K Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Spellbound: Defending against package typosquatting. *arXiv:2003.03471*, 2020.

[52] Tembo. Vector indexes in pgvector. https://tembo.io/blog/vector-indexes-in-pgvector, 2024.

[53] Christopher Tozzi. Package hallucination: The latest, greatest software supply chain security threat? *IDC Blog*, April 22 2024.

[54] Marcin Ulikowski. dnstwist: Domain name permutation engine for detecting homograph phishing attacks, typo squatting, and brand impersonation. https://github.com/elceef/dnstwist, 2025.

[55] Rodolfo Valentim, Idilio Drago, Federico Cerutti, and Marco Mellia. Ai-based sound-squatting attack made possible. In *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 448–453. IEEE, 2022.

[56] Rodolfo Vieira Valentim, Idilio Drago, Marco Mellia, and Federico Cerutti. X-squatter: Ai multilingual generation of cross-language sound-squatting. *ACM Transactions on Privacy and Security*, 2024.

[57] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, Andre DeHon, and Jonathan M. Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[58] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. LastPyMile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 780–792, Athens Greece, August 2021. ACM.

[59] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Typosquatting and combosquatting attacks on the python ecosystem. In *EuroS&P Workshops*, pages 509–514. IEEE, 2020.

[60] Vulcan Cyber. Can you trust chatgpt's package recommendations? https://vulcan.io/blog/ai-hallucinations-package-risk, April 17 2023.

[61] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *International Conference on Mining Software Repositories (MSR)*, 2016.

[62] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork? finding hidden code clones in npm. In *Proceedings of the 44th ICSE*, 2022.

[63] Sibo Yi, Yule Liu, Zhen Sun, Tianshuo Cong, Xinlei He, Jiaxing Song, Ke Xu, and Qi Li. Jailbreak attacks and defenses against large language models: A survey. *arXiv:2407.04295*, 2024.

[64] Nusrat Zahan, Tom Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are Weak Links in the npm Supply Chain? In *International Conference on Software Engineering (ICSE)*, May 2022.

[65] Markus Zimmermann, Cristian-Alexandru Staicu, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security Symposium*, 2019.

[66] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv:2307.15043*, 2023.

A Taxonomy of Engineering Practices

Figure 7 presents the taxonomy we created in §3.2.

B More System Implementation Details

B.1 Visualization of Embedding

Figure 6 shows a `UMAP` visualization of 100K npm packages [30]. Clusters often form around minor spelling variations, demonstrating how embeddings capture both lexical and semantic relationships. This grouping is central to detecting maliciously similar names that differ by a single character or switched letters.

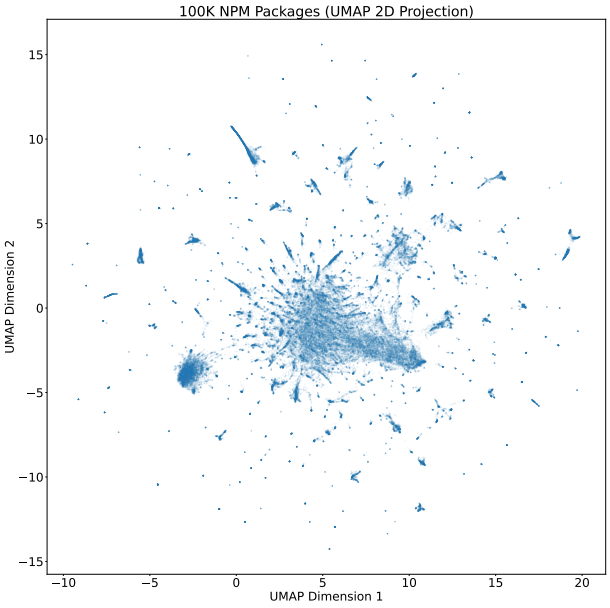


Figure 6: UMAP visualization of embedding space of 10K npm packages.

B.2 Detailed Embedding Efficiency Measurement

Table 7 highlights the efficiency of embedding models across different quantization formats (`float32`, `float16`, and `int8`), indexing overhead, and memory usage. While `float16` and `int8` offer comparable throughput and latency due to similar I/O and storage overheads, `int8` exhibits slightly higher latency in embedding creation due to additional quantization steps. Across ecosystems, `float16` and `int8` achieve significant improvements in processing speed and memory efficiency compared to `float32`, demonstrating their suitability for scalable embedding-based systems.

B.3 Implementation of metadata verification rules

Algorithm 1 shows our implementation of the engineering practice categories. We apply both heuristic rules and LLM-based rules to measure the attributes from Table 2 and use this algorithm to check if a suspicious pair should be classified as true-positive or false-positive.

Algorithm 1 Heuristic Rules for False-positive Verification

Input: typoDoc, legitDoc, registry
Output: Boolean, metrics, explanation
Initialize: Set all keys in metrics to None
for each key in metrics:
    metrics[key] ← None
Populate metrics using helper checks:
    metrics ← _check_package_naming_and_purpose(typoDoc, legitD
    metrics["overlapped_maintainers"] ← _has_overlapped_maintainers(typoDoc, legitDoc, registry)
    metrics["comprehensive_metadata"] ← _has_comprehensive_metadata(typoDoc)
    metrics["active_development"] ← _is_actively_developed(typoDoc)
If ¬metrics["is_adversarial_name"] **and** ¬metrics["is_suspicious"]:
    If metrics["obvious_not_typosquat"] **or** metrics["fork_naming"] **or** isTest **or** metrics["is_known_maintainer"] **or** metrics["distinct_purpose"] **or** metrics["overlapped_maintainers"]:
        return (True, metrics, explanation)
If metrics["is_adversarial_name"] **and** ¬metrics["distinct_purpose"]:
    return (False, metrics, explanation)
If ¬metrics["comprehensive_metadata"] **or** ¬metrics["distinct_purpose"] **and** ¬metrics["is_adversarial_name"] **or** ¬metrics["active_development"]:
    return (False, metrics, explanation)
If metrics["is_adversarial_name"] **and** (¬metrics["distinct_purpose"] **or** ¬metrics["comprehensive_metadata"]):
    return (False, metrics, explanation)
isDeprecated ← _check_deprecated(typoDoc)
If isDeprecated **and** metrics["is_adversarial_name"]:
    return (True, metrics, explanation)
If metrics["is_suspicious"]:
    return (False, metrics, explanation)
Return: (True, metrics, explanation)

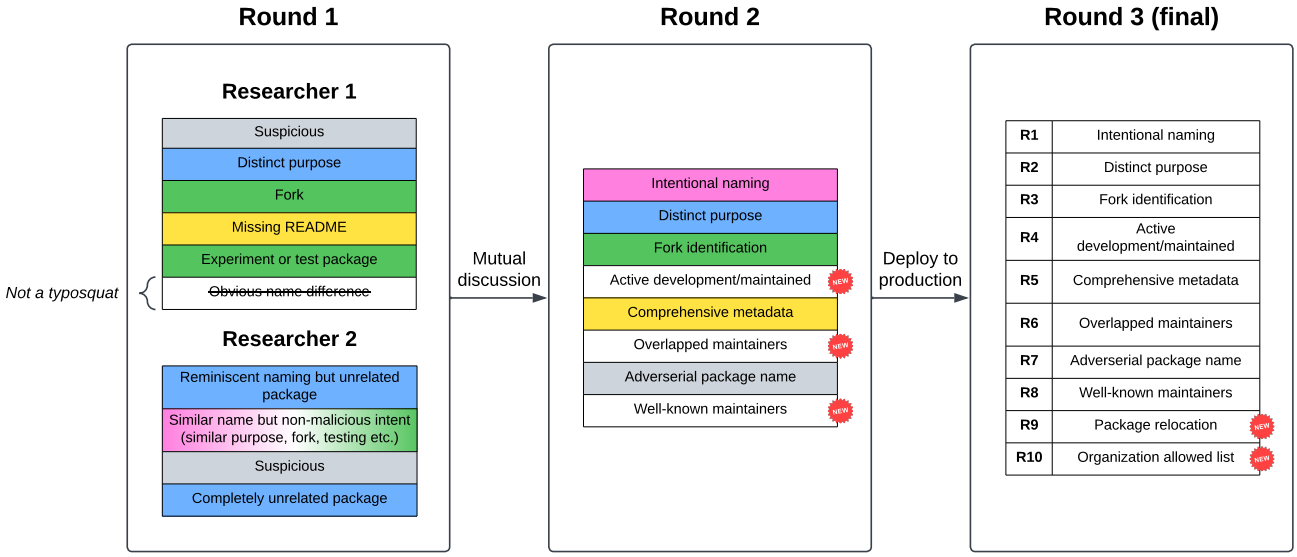


Figure 7: Taxonomy of engineering practices.

Table 7: Evaluation of embedding model efficiency, *HNSW* indexing overhead, and memory usage. The throughput and latency differences between `float16` and `int8` are minimal due to similar overall I/O and storage overheads, as well as the efficient handling of embeddings using `pgvector`. However, some increased latency in `int8` embedding creation is observed due to additional quantization and processing steps required for integer-based representations.

Quantization	Ecosystem	Throughput (batches/s)	Avg Batch Latency (s)	Avg Pkg Latency (µs)	Total Time (s)	PG Table Size (GB)	PG DB Size (GB)	Indexing Time (s)
float32	Hugging Face	4.20	1062.44	2380.68	1941.68	9.25	45.26	4.46
	Maven	4.59	697.16	2180.61	1380.76	6.57	46.97	4.91
	Golang	2.42	3778.33	4135.75	7905.49	21.36	54.81	7.80
	npm	3.73	7463.84	2678.89	13269.07	21.04	60.64	7.42
	PyPI	9.57	312.03	1045.52	602.95	2.62	61.49	4.39
	RubyGems	9.54	116.86	1049.14	220.00	0.94	61.80	4.07
float16	Hugging Face	19.67	203.86	508.61	414.82	3.22	36.73	4.75
	Maven	20.44	156.54	489.42	309.90	2.51	34.74	4.77
	Golang	19.07	520.25	524.55	1002.68	7.56	29.70	4.79
	npm	36.94	669.87	270.72	1340.91	6.69	22.20	5.97
	PyPI	38.33	76.03	261.04	150.54	0.77	21.38	4.23
	RubyGems	37.88	28.09	264.40	55.44	0.28	21.02	4.35
int8	Hugging Face	22.12	192.31	452.20	368.81	3.22	36.73	4.84
	Maven	18.32	161.71	546.01	345.73	2.51	34.72	4.81
	Golang	19.23	499.12	520.22	994.40	7.56	29.70	4.75
	npm	35.28	719.06	283.46	1404.03	6.69	22.20	5.28
	PyPI	37.57	80.66	266.35	153.60	0.77	21.37	4.31
	RubyGems	39.71	26.70	252.35	52.92	0.28	21.00	4.23