# Beyond Typosquatting: An In-depth Look at Package Confusion

Shradha Neupane
*Worcester Polytechnic Institute*

Grant Holmes
*University of Kansas*

Elizabeth Wyss
*University of Kansas*

Drew Davidson
*University of Kansas*

Lorenzo De Carli
*University of Calgary*

## Abstract

Package confusion incidents—where a developer is misled into importing a package other than the intended one—are one of the most severe issues in supply chain security with significant security implications, especially when the wrong package has malicious functionality. While the prevalence of the issue is generally well-documented, little work has studied the range of mechanisms by which confusion in a package name could arise or be employed by an adversary. In our work, we present the first comprehensive categorization of the mechanisms used to induce confusion, and we show how this understanding can be used for detection.

First, we use qualitative analysis to identify and rigorously define 13 categories of confusion mechanisms based on a dataset of 1200+ documented attacks. Results show that, while package confusion is thought to mostly exploit typing errors, in practice attackers use a variety of mechanisms, many of which work at semantic, rather than syntactic, level. Equipped with our categorization, we then define detectors for the discovered attack categories, and we evaluate them on the entire npm package set.

Evaluation of a sample, performed through an online survey, identifies a subset of highly effective detection rules which (i) return high-quality matches (77% matches marked as potentially or highly confusing, and 18% highly confusing) and (ii) generate low warning overhead (1 warning per 100M+ package pairs). Comparison with state-of-the-art reveals that the large majority of such pairs are not flagged by existing tools. Thus, our work has the potential to concretely improve the identification of confusable package names in the wild.

## 1 Introduction

Modern, language-based software ecosystems (LBEs) contain expansive repositories of third-party code that can be conveniently downloaded and installed by developers. The *packages*[1] of code contained in these repositories supply ready-made, diverse functionality to be used as part of a larger codebase. The popularity of package repositories is apparent through their usage: The package ecosystems npm for node.js, RubyGems for Ruby, and PyPI for Python collectively host millions of distinct packages and serve billions of package downloads weekly [66].

Tooling and automation has eased the task of finding and deploying packages. A simple invocation of the install command for the package manager frontend tool can be responsible for the cascading download of hundred of distinct packages, as (transitive) dependencies are discovered, fetched, and installed. The ease of use built into package ecosystems also increases the likelihood of a developer completing the entire installation process on a package that they did not intend to download. Should an error be made when invoking the name of an intended package, a completely different package name will downloaded and deployed. This set of circumstances allows the use of the LBE as a vector for software supply chain attacks. An adversary might publish a malicious package that attacks a developer when the package is installed, or delivers malicious functionality to end-users when the malicious package is used as part of a larger project.

In order to realize the type of incident described above, a victim developer needs to download the malicious package. Thus, the adversary's goal is to carry out a *package confusion attack*, in which a malicious package is created that is designed to be confused with a legitimate target package and downloaded by mistake. Such attacks have been shown to occur in practice [56], effecting developers that mistakenly install the package directly, and any other deployments that includes a malicious package in its transitive dependencies.

Confusion attacks can leverage the long tail distribution of packages. The top 1% most popular packages are responsible for over 99% of downloads [58]. Since LBE package managers fetch a package based on its name, the attacker can upload a package with a name that is easily confused with a legitimate popular package and passively launch the attack

[1] Although various LBEs use specialized names for the units of code that they serve, such as “gems” [2] or “crates” [11], we refer generically to each distinct unit as a package in this paper.

when their uploaded package name is mistakenly fetched.

Previous work [26, 58, 67] is largely based on intuitive notions of the types of confusion that can occur— such as *typosquatting*, which consists of mistypings of package names. However, this does not characterize the myriad other ways in which a developer might be confused by an alternative name. For example, developers may assign connotations to the words in a package name ("guard" vs "watchdog"), or be misled by grammar stems ("swaggerize" vs "swaggerify") upon which the adversary could base an attack.

In this work, we provide a rigorous study into characterizing how package confusion attacks have occurred in the past and may continue to occur in the future. To this end, our contribution consists in answering two main research questions:

**RQ1: What types of package confusion attacks exist among previously discovered attacks?** We evaluate this question with a rigorous, manual evaluation of over 1200 historical package confusion attacks over the past 6 years, ultimately deriving a set of 13 categories.

**RQ2: What types of package confusion exist in the wild?** Through the use of manual and automated analysis, we performed a large-scale analysis of the current state of the npm package repository, and tested the feasibility of algorithmic detection of package confusion in the field.

We describe the existence of a number of attack categories ignored by existing tools. We also show that some of these categories can be identified with high accuracy by the tooling developed as part of this project, while others require additional data to be fully addressed. Thus, we believe that our work provides practical contributions to the understanding and detection of package confusion attacks in the wild.

In the rest of the paper, we present the background necessary to understand the context of package confusion attacks, describe our research infrastructure to explore the above research questions, and detail the results of our measurements.

## 2 Background

### 2.1 Supply-chain Security

Language-based software ecosystems (LBE) are large-scale repositories of software packages written in a single language [61]. Examples include npm for JavaScript, PyPI for Python, RubyGems for Ruby and others. These large LBEs enable rapid prototyping and development of software by enabling developers to quickly and easily import external code in their project in the form of dependencies. npm, the largest ecosystem, alone has **1,826,912** packages as of April 2022.

The biggest strength of these ecosystems, the ability to import dependencies, is also a significant vector for software compromise. Even simple packages may import a large number of dependencies, each of which in turn imports their own; as a result, the dependency tree of a package may be many levels deep, and include hundred of packages [70]. This create a fertile ground for attackers, as the attack surface of a package includes not only the package itself, but all of its dependencies, given rise to *software supply chain attacks*. In such an attack, rather than directly attempting to compromise a popular package/library, an attacker may attempt to inject malicious code into one of its dependencies. This is particularly effective if the dependency is an obscure package, hidden deep in the dependency chain of a popular one, and subject to less scrutiny. Supply chain attacks are rapidly emerging as one of the chief issues in modern software security, originating a large number of incidents and being recognized as critical by industry and government entities [4, 56].

### 2.2 Package Confusion Attacks

As noted in Section 1, a package confusion attack is a form of supply chain attack where an attacker uploads into an ecosystem a malicious package which is confusable with another, benign package. It should be noted that there are documented cases of package confusion happening without malicious intent. A concerned developer may create a confusable package to prevent a threat actor from creating a malicious package with the same name. In this case, the confusable package may be empty, or may simply duplicate the original's package code. This is for example the case of the npm package *loadsh*, which targeted the benign package *lodash* but did not contain any malicious code, and was intended to prevent anyone else from creating a package with the same name [58]. Confusable packages may also be created inadvertently. Developers frequently duplicate packages for various purposes, oftentimes creating confusable package *clones* [66]. Finally, someone may simply create a package whose name happens to be similar to another one, without realizing it.

Nevertheless, importing a confusable but non-malicious package wastes developer time, may cause degraded functionality, and may actually still create security issues. Many such packages lag behind in making security updates, resulting in long-lived latent vulnerabilities [66]. Even if confusable packages can be created by accident, in many documented instances post-facto analysis of incidents did reveal malicious or at least ambiguous intent. Indeed, one of the datasets used in this paper (described in Section 3), aggregated from public sources and covering the last 6 years, contains over 1,200 documented malicious package confusion instances.

In order to ensure consistent terminology without implying malicious intent (as this may not always exist as discussed above), we term an benign popular package as *target*, and a package with a name confusable with that of the target as *confuser*. Furthermore, we mark packages as potential confusers against a target using confusability potential rated by humans, rather than attempting to infer malicious intent.

| Source | #samples |
|---|---|
| ReversingLabs [45] | 721 |
| JFrog [16] | 207 |
| snyk.io [13] | 172 |
| GitHub advisories [7] | 100 |
| Publication [64] | 17 |
| SonaType [55] | 5 |
| npm advisories [9] | 3 |
| ZDNet [23] | 3 |
| BleepingComputer [17] | 2 |
| The Record [24] | 2 |
| **Total** | **1232** |

Table 1: Sources of attacks in dataset

## 3 Datasets

Our work involves analyzing known package confusion attacks (Section 4), and building upon the results to estimate the footprint of package confusion in the wild (Sections 5 and 6). This section describes our datasets.

### 3.1 Package Confusion Attacks Dataset

**Definitions.** Our dataset consists of past documented attacks. Our definition of *documented attack* requires that an established source publishes information calling out a package as a security risk, and explicitly characterizes the attack as involving package confusion. An *established source* may be: a generalist advisory program (e.g., CVE [5]); the advisory feed of one of the ecosystems analyzed (e.g., npm [9]); a peer-reviewed publication (e.g., [64]); a specialized media outlet (e.g., zdnet [23]); a public post or press release from a threat intelligence company (e.g., ReversingLabs [45]). *Characterizing the attack as involving package confusion* means that the description either includes representative keywords (e.g., “typosquatting”), or describes the mechanism of the attack as inducing confusion in developers due to the package name.

**Dataset construction.** The set of attacks was built as follows. First, we compiled a list of well-known generalist and ecosystem-specific advisory programs including CVE/NVD feed [5], snyk.io [13], GitHub advisories [7], npm security advisories [9], the PyPI community-maintained advisory database [10], and RubyGems advisories [12]. We then searched all advisory lists above starting from 2016, using a set of relevant keywords (e.g., “typosquatting”, “name confusion”, etc.) and vetting the results to ensure they matched our criteria for inclusion. Furthermore, we performed a literature search in relevant computer security academic venues to identify papers focusing on supply chain security, which included datasets or mentions of specific attacks. Finally, we carried Google searches with a list of relevant keyphrases (e.g., “npm malicious software”, “PyPI typosquatting”), collecting and manually vetting results. We continued the search process until we determined that new sources failed to contribute attacks not already in our dataset. We note that some commercial sources may have vested interests in framing unrelated attacks as package confusion, to promote their services. We mitigate this risk by choosing a diverse set of sources; and manually inspecting incident descriptions.

**Results.** The result consists of **1232** documented attacks. Table 1 summarizes the source of attacks in our dataset. ReversingLabs contributed a large campaign against the RubyGems ecosystem. JFrog contributed a similar campaign against npm.

### 3.2 Package Dataset

The dataset we used to estimate the prevalence of package confusion in the wild consists of a snapshot of all npm package names, taken in **April 2022**. The snapshot consisted of **1,826,923** package names. We selected npm as our analysis target as it is the largest among popular software ecosystems. As such, we expect the results to be both representative, and relevant to the Open Source developer community.

As we will detail in Section 6.1.1, our analysis logic prefilters package pairs based on popularity (measured as downloads per week). Download counts for each package in the dataset were obtained by using the npm public API. Overall, we collected 1,752,424 package names and their download counts. The remaining 74,499 packages in the snapshot and weekly downloads had no download count. This meant the 74,499 packages that existed in the npm snapshot did not exist in the npm repository when we got the download counts.

## 4 Categorizing Package Confusion Attacks

Virtually all previous investigations of package confusion attacks [58, 64] are based on expert, but observational, understanding of the phenomenon. As a result, existing tools use detection rules that "make sense" to domain experts. While this approach is intuitive, we believe a comprehensive understanding of the phenomenon requires a rigorous assessment of all available data on past package confusion incidents. In turn, we expect that such comprehensive understanding will result in more effective detection procedures.

### 4.1 Methodology

We analyze the attack dataset described in Section 3.1. Our initial goal is to produce a comprehensive categorization of package confusion attacks, to answer RQ1 from Section 1 (*What types of package confusion attacks exist among previously discovered attacks?*) As package confusion is arguably a social engineering attack aimed at confusing humans, we perform categorization using human experts.

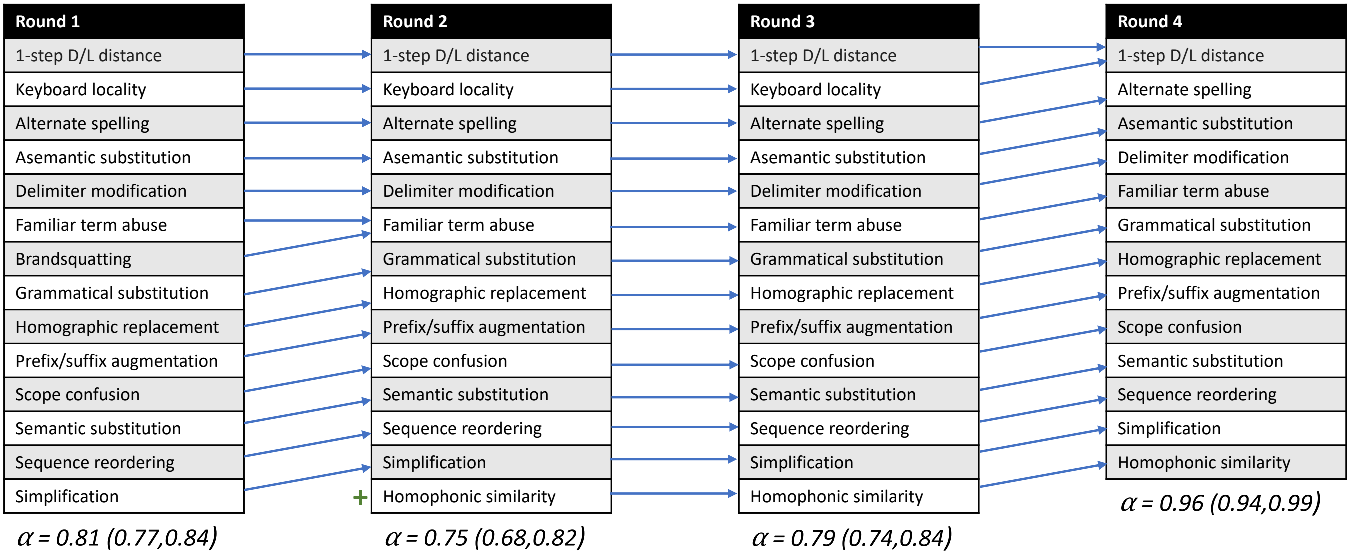


Figure 1: Evolution of codebook and inter-coder agreement per round

Labeling and categorizing textual data according to human interpretation is traditionally the purview of qualitative research methods. Processes such as grounded theory [25] and thematic analysis [59] establish methodology to ensure experimenters follow a repeatable and rigorous process. Thematic analysis is a process for qualitative data analysis due to Braun and Clarke [19], which has found application in the usable security domain (e.g., [37, 44]). We pick this approach to categorize attacks in our dataset due to its emphasis on inductive reasoning, which is necessary in our cases due to lack of prior knowledge on the nature of package confusion attacks. While such methods are typically applied to expressive, long form text (e.g., interview data [27], program snippets [57]), we decided to use it to provide methodological rigour to the process of deriving attack categories.

Our process, established prior to coding, implements a simplified version of thematic analysis as follows. Five members of the research team (two senior researchers, one senior graduate student, and two junior students) review the same sample of 100 incidents, each coming up with an initial set of codes. These are then discussed until everyone agreed on the same set. Subsequently, all coders code the same set of 50 yet-uncoded incidents, using Krippendorff's $\alpha$ to measure inter-coder agreement. This step is repeated until $\alpha \geq 0.8$ (commonly used in literature to denote strong inter-coder agreement). In each round resulting in $\alpha < 0.8$, we follow up by requiring *all* disagreements to be resolved. After reaching $\alpha \geq 0.8$ we split the remaining incidents among the coders.

## 4.2 Coding and Results

We performed coding following the above pre-established methodology, with only two minor changes. First, we noticed that 722 out of 1232 attacks originate from a RubyGems campaign, which batch-generated a large number of package names by altering delimiters. We marked all such cases as *Delimiter modification* without further analysis. Furthermore, the first round of coding resulted in $\alpha = 0.81$, however we agreed that finalizing the set of codes at this early stage would have been premature. Therefore, we continued coding in parallel. In hindsight, this decision appears appropriate as the second and third round of coding resulted in $\alpha$ below 0.8 and various changes to the initial codebook. We terminated parallel coding in round 4, with $\alpha = 0.96$ and a set of codes we collectively deemed stable. Figure 1 displays the evolution of the codebook; it also displays, under each round of coding, inter-coder agreement $\alpha$ and its 95% bootstrapped CI. We generated our categorization of package confusion attacks simply by interpreting each code as an attack category. Table 2 describes the final 13 categories.

## 4.3 Discussion

The main events in the evolution of the codebook are instances of merging two categories into one. In the first instance, we merged *Brandsquatting* — intended as the use of a popular brand in a package name to attract installs — into the more general *Familiar term abuse*. In the second case, we merged *Keyboard locality* — referring to a confuser name with a 1-character difference that can be introduced as the result of a typo — into the more general *1-step Damerau/Levenshtein distance*. The reason to do so is that the potential for typos depends on the specific keyboard layout being used.

The relationship between categories requires further discussion. Every confuser name can theoretically be transformed into the target name by a sequence of edit steps. Under this

| Id | Category name | Description |
|---|---|---|
| 1 | Prefix/suffix augmentation | A prefix or suffix has been added to the target package name. The added term may refers to the ecosystem, language of the package, a specific functionality, or version number. **Examples:** `dateutil` → `python3-dateutil`; `genesis` → `genesisbot`. |
| 2 | Sequence reordering | The target package can be divided into multiple segments of consistent meaning, which have been reordered in the confuser package name. **Example:** `python-nmap` → `nmap-python`; `libhtml5` → `html5lib`. |
| 3 | Delimiter modification | Character based-changes in the target package name where delimiters have been added, removed or swapped with a different type. **Examples:** `workarea-gift_cards` → `workarea-gift-cards`; `active-support` → `activesupport`. |
| 4 | Grammatical substitution | The confuser package substitutes terms in a different grammatical forms of the target package's terms (singular/plural, different verbal form). **Examples:** `serialize` → `serializes`; `learnlib` → `learninglib`. |
| 5 | Scope confusion | The confuser package name is unscoped, but resembles the name of target scoped package. **Example:** `@cicada/render` → `cicada-render`. |
| 6 | Semantic substitution | The confuser package substitutes the target package name (or a segment therein) with terms that have the same meaning. **Example:** `bz2file` → `bzip`; `electron-native-notify` → `electron-native-notification`. |
| 7 | Asemantic substitution | The confuser package substitutes terms in the target package that are familiar to the developers. The replacement terms do not match the original ones in meaning. **Examples:** `discord.js` → `discord.app`; `libcurl` → `pycurl`. |
| 8 | Homophonic similarity | The confuser package and the target packages differ in spelling or letters but sound the same. Similarity is based on homophones. **Examples:** `uglify-js` → `uglyfi.js`; `asinc` → `async`. |
| 9 | Simplification | Removal of prefix/suffix, such that: the target package name can be divided into multiple segments of consistent meaning, and any of the segments have been removed, without altering the overall semantic meaning of the name. **Examples:** `pwdhash` → `pwd`; `urllib3` → `urllib`. |
| 10 | Alternate spelling | Target and confuser package names differ due to different local spellings but there is no change in the meaning of the target package name. **Examples:** `colorama` → `colourama`; `colour-string` → `color-string`. |
| 11 | Homographic replacement | Character based change in the target package name where the exchanged characters look similar. **Examples:** `django` → `diango`; `jellyfish` → `jeilyfish`. |
| 12 | 1-step Damerau/Levenshtein distance | The target and confuser package names have 1-step Damerau/Levenshtein distance. The difference may involve character substitution, omission, addition, or swapping. **Examples:** `crypto` → `crypt`; `express -> experss`. |
| N/A | Familiar term abuse | The confuser package name includes terms that are familiar to the user, such the name of a company or a popular technology. Differently from other attacks, this one does not postulate the existence of a target package. **Examples:** `plutov-slack-client`; `applogger-ruby`. |

Table 2: Package confusion categories

interpretation, every attack is a composition of multiple instances of Damerau/Levenshtein distance (Rule 12 in Table 2); however, this interpretation is not very useful. Instead, we use Rule 12 as a fallback for all cases that cannot be interpret as part of another, more specific category. There are a number of similar instances of the same problem; for example, one can envision attacks which could be classified both as Homophonic replacement and Homographic replacement. We resolve this issue by defining a priority order between categories; categories in Table 2 are listed by decreasing priority. The priorities were attributed empirically, going from what we perceived to be more specific categories to less specific ones. *Familiar term abuse* has no priority as, unlike other categories, it does not assume a target package. We acknowledge that priority assignment is subjective. However, our main goal here is to generate comprehensive categories; the specific choice of priorities is inconsequential to those.

We also allow rules to be composed. The choice of whether labeling a sample as a composition of categories rather than a new category is again based on the coders' perception of the attack. An example is `pymongodb` being mutated into `python-mongo`; we classified this pair as a combination of *Semantic substitution* (`py` → `python`), *Delimiter modification* (add "`-`"), and *Simplification* (`mongodb` → `mongo`).

## 4.4 Nature of attacks

Figure 2 depicts the distribution of category labels across the attack dataset. Note that the number of labels is higher than the number of attacks as composite attacks have multiple labels. Figure 2(a) breaks down all attacks by category and ecosystem. Furthermore, the distribution of attacks is highly dependent on what we term *confusion campaigns* in our dataset. Specifically, we define such a campaign as the

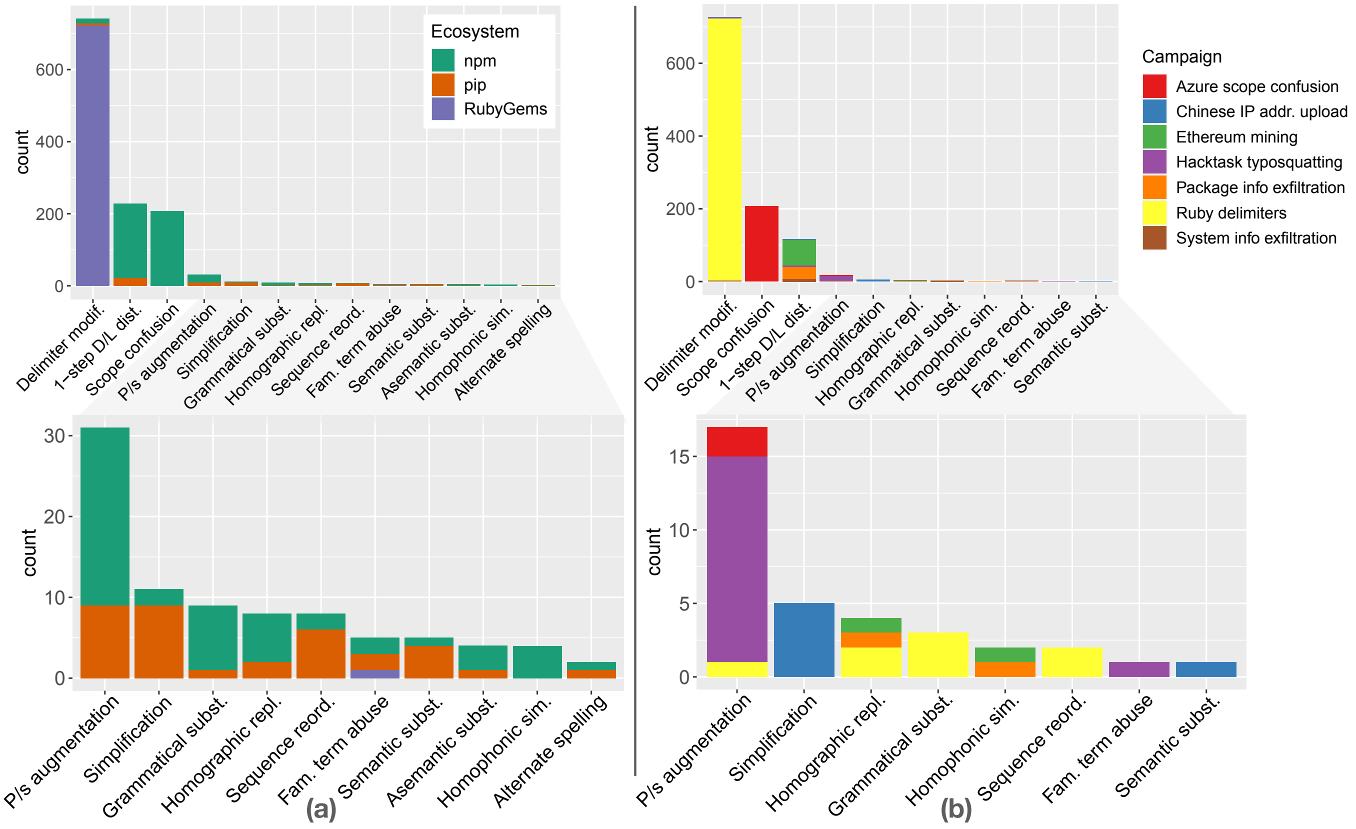


Figure 2: Distribution of manually-generated category labels across ecosystems **(a)** and campaigns **(b)**

coordinate upload of 10 or more packages by the same entity. Our dataset include 7 such campaigns [16, 22, 29–31, 36, 45] (1077 incidents total out of 1232). Figure 2(b) breaks down attacks by campaign. For 5 out of 7 campaign, uploaded packages predominatly fall within one category.

Package confusion is oftentimes referred to as *typosquatting*. This carries the implicit assumption that the confuser package attempts to capture some of the target's package installs by exploiting typing errors. In practice, this is not substantiated by our analysis. A way to interpret our categories is that Rules 1-11 are arguably based on various forms of semantic confusion; while Rule 12 acts as a fallback for cases where no semantic mechanism is apparent. Together, Rules 1-11 make up **82%** of our attack dataset (**43%** when excluding the RubyGems campaign). This suggests that giving consideration to semantic-based confusion attacks is important.

# 5 Detecting Package Confusion

We now turn to RQ2 from Section 1: *What types of package confusion exist in the wild?*. To answer this question, we design *detection rules* to identify instances of package pairs belonging to the categories in Table 2.

Our categories are qualitative and descriptive, and thus cannot be directly used for detecting new instances. Our detection rules are designed to *approximate* the categories while achieving broad performance goals (discussed in Section 5.1). Below we describe the design of the detection rule for each category. We decided not to detect *Familiar term abuse*, since the mere presence of a popular brand or term (e.g., "Twilio", "MariaDB") without additional context, is too weak of a signal to identify cases likely to arise confusion. As our goal is to measure semantic package confusion beyond basic local textual differences, we also excluded the *1-step D/L* category from measurements.

## 5.1 Design process and evaluation

To design the detection rules discussed below, we created initial prototypes which we then iteratively refined. At each iteration, we measured rule precision, recall, and F1-score on the attack dataset (1232 packages). To do so, we interpreted human-generated labels as ground truth.

Consistently with literature [58], we did not strive to maximize an aggregate metric (such as F1 score). Instead, we treat ours as a multi-objective optimization problem, prioritizing precision over recall. The rationale is that confusing packages

| Rule | #TP | #TN | #FP | #FN | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|
| P/s augmentation | 21 | 1193 | 1 | 9 | 0.95 | 0.70 | 0.81 |
| Sequence reord. | 7 | 1215 | 1 | 1 | 0.88 | 0.88 | 0.88 |
| Delimiter modif. | 717 | 483 | 0 | 24 | 1 | 0.97 | 0.98 |
| Grammatical subst. | 7 | 1215 | 1 | 1 | 0.88 | 0.88 | 0.88 |
| Scope confusion | 188 | 1016 | 0 | 20 | 1 | 0.90 | 0.95 |
| Semantic subst. | 2 | 1219 | 0 | 3 | 1 | 0.4 | 0.57 |
| Asemantic subst. | 3 | 1219 | 1 | 1 | 0.75 | 0.75 | 0.75 |
| Homophonic sim. | 3 | 1182 | 38 | 1 | 0.07 | 0.75 | 0.13 |
| Simplification | 7 | 1208 | 5 | 4 | 0.58 | 0.64 | 0.61 |
| Alternate spelling | 2 | 1222 | 0 | 0 | 1 | 1 | 1 |
| Homographic repl. | 7 | 1209 | 7 | 1 | 0.5 | 0.88 | 0.64 |

Table 3: Performance of detection rules

are rare due to the scale of software ecosystems. Thus, poor precision leads to detectors overwhelmingly returning false positives (a form of the *base rate fallacy* [18]). Therefore our goal is to maximize the chances of identifying actually confusable packages, at the cost of missing some instances.

This approach is not always applicable, however, as our dataset is significantly imbalanced; for some categories, only $< 5$ samples exist. In those circumstances, even a small number of false positives can significantly affect precision. For those, we relax our requirement to maximize precision to ensure a recall of at least 33% (a rule that does not detect anything has 100% precision, but it is not useful). We believe this limitation is acceptable, since our analysis of potential package confusion instances in the wild entails manual review of matches (Section 6). Table 3 details true and false positives/negatives, precision, recall, and F1 scores for all rules. Note the total in Table 3 are less than 1232 because they exclude package pairs where the target was undefined or ambiguous (5 cases of *Familiar term abuse*, and 3 cases where the attack report did not disambiguate the specific victim package).

## 5.2 The Language of Package Names

When we set to implement detection rules, one of our earliest findings was that package names tend to largely consist of *jargon*: acronyms (e.g., “html”, “npm”) and other various types of textual tokens (e.g., “js”, “db”) many of which are not valid words in any human language, but assume a valid *connotation* in the context of the language-based software ecosystem. Many package names consist of “sentences” resulting from concatenation of such tokens, sometimes interspersed with English words (e.g., “js-sha3”, “hw-transport-u2f”, etc.), and we found it impossible to productively creating rules without building a model of the jargon first.

Our model consists of a list of jargon tokens, each annotated with its frequency in the corpus. To do so, we run 1.9 million package names from PyPI and npm through a token extraction pipeline. The pipeline splits each name across common delimiters to generate initial tokens, then attempts to further split those using a English word segmenter [33]. This last step serves to split strings such as "easyinstall" into "easy" and "install". To reduce noise, tokens that occur in less than 100 package names are removed. This process results in 10425 unique tokens, that form the “Jargon” corpus. 5110 of these tokens are not present in the standard English corpus. Figure 3 illustrates extensive reuse of a very small percentage of the tokens across many package names. The primary use of this corpus is for delimiterless tokenization.

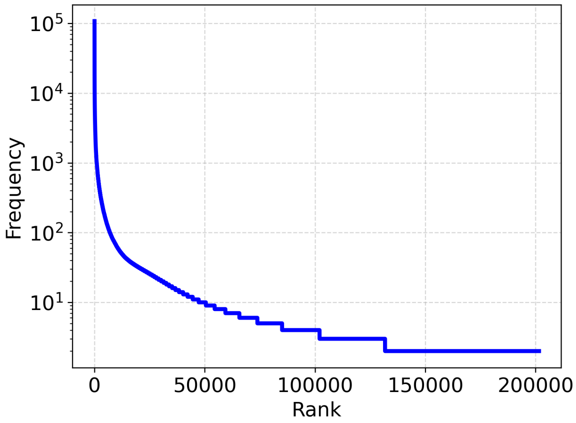


Figure 3: Package token frequency VS rank.

### 5.2.1 Delimiterless Tokenization

A number of the detectors need to transform a package name into a sequence of tokens. However, many package names omit token delimiters; e.g., "html5lib", "easyinstall", "setuptools" and many more. This common practice means that tokenizing a package name requires a more complex approach than simply splitting along a set of known delimiters. This is further complicated as many of the tokens are not present in a standard English corpus, such as "html", "lib", and "dev".

**Algorithm 1** Delimiterless tokenization

```
Input: Word $w$
Input: Set of corpuses $C = \{C_E, C_J\}$
Output: Tokenization $S$
1: $\mathcal{T} = \emptyset$
2: for each $c \in C$ do
3:     $\mathcal{T} = \mathcal{T} \cup ForwardPass(w, c)$
4:     $\mathcal{T} = \mathcal{T} \cup BackwardPass(w, c)$
5: if $\#(Lengths(\mathcal{T})) > 2$ then return $\emptyset$
6: $\mathcal{T}_2 = \{T \in \mathcal{T} \mid \forall t \in T, t \in C_E \cup C_J\}$
7: $\mathcal{T}_3 = \{T \in \mathcal{T}_2 \mid \#T = Mode(Lengths(\mathcal{T}_2))\}$
8: $\mathcal{T}_4 = \{T \in \mathcal{T}_3 | T \supseteq MostCommonSequence(\mathcal{T}_3)\}$
9: return $argmax_{S \in \mathcal{T}_4}(min(Lengths(S)))$
```

Thus, we developed a delimiter-less tokenization algorithm that makes use of both a standard English word corpus from the NTLK framework [8], and our jargon corpus.

**Tokenization algorithm.** Given a string without delimiters, Algorithm 1 returns a sequence of one or more tokens. The algorithm first uses a sliding window matcher to identify substrings from the input word $w$ which match the corpus. The matcher uses such substrings as boundaries to create a tokenization. The process is repeated using both a forward- and backward-sliding window, for both the English corpus $C_E$ and the jargon corpus $C_J$ (lines **1-4** of Algorithm 1). The result is a set of candidate tokenizations $\mathcal{T}$. Next, the algorithm computes the lengths of all $T \in \mathcal{T}$, and returns if there exists more than 2 different lengths (line **5**). This is because we found that widely differing tokenizations is a strong predictor of a non-segmentable word.

Next, the algorithm applies a simple heuristic to filter candidate tokenizations and select the winner (lines **6-9**). First, all tokenizations including substrings that are not present in any corpus are discarded (line **6**), as "gibberish" tokens are a predictor of poor tokenization. Second, only tokenizations whose length is equal to the mode of all remaining tokenization lengths are retained (line **7**). Third, the algorithm computes the most frequent token subsequence within all tokenizations, and only retain tokenizations containing that sequence (line **8**). These two rules reward tokenizations which were identified, with small variations, across multiple passes. Finally, if ties exist they are resolved by picking the tokenization which the longest shortest token length (line **9**). This heuristic is motivated by the empirical observation that poor tokenizations tend to contain many short substrings.

## 5.3 Detection Rules

**Prefix/Suffix Augmentation** describes cases where the confuser package adds a prefix or a suffix to the target. The detector for this rule marks package pairs where the target package is a substring of the confuser package, and all tokens in the target package are *uncommon*. We consider a token uncommon if its frequency in package names is less than 15% that of the most common token. This constraint is due to the fact that augmentation of a very common term is likely not confusable. To limit false positives, this detector only triggers if the length of both the confuser and target name is greater than 3; and the length of the confuser is less than twice the length of the target.

**Sequence Reordering** consists of a confuser package rearranging the order of tokens in the target package. The detector tokenizes both package names into token sequences, and then checks if the sorted sequences are equal.

**Delimiter Modification** consists of a confuser package that modifies delimiters from the target package name. Modification includes the omission, addition, or transformation of delimiter sequences. The detector first tokenizes both package names across their delimiters. Next, for each token sequences, a set is constructed by iterating over the possible concatenations of token subsequences into a single string. A non-empty intersection of these two sets results in a match.

**Grammatical Substitution** involves a confuser making changes to the grammar of words in the target name. The detector tokenizes both package names, then use the NLTK lemmatizer [8] to normalize each token to its root (e.g. "loader"/"loading" to "load"). Depending on how may tokens have the same root format, it then checks if one is a simple plural form of the other token. Eg. *handle* and *handler*. If all tokens in both packages match but one which differs by being a plural form of the other, then the grammatical substitution detector returns positive.

**Scope Confusion** consists of a confuser package which is confusable with a scoped target name. The detector checks for two different cases. In the first, the confuser has the same name as the target, but only the target is scoped. The detector works by removing the scoping markers("@" and "/") from both package names. The package names are split at "/". If both the package names are equal, with the target being scoped and confuser being unscoped, the detector flags the pair as a match (e.g., *@cadl-lang/openapi3* (target) and *openapi3*). In the second case, the detector removes the scoping delimiters and creates a list of all package tokens. The detector then checks if the confuser and target tokens are equal (e.g., *@cicada/render* (target) and *cicada-render*.)

**Semantic Substitution** occurs when a token in the target name is replaced by one of similar meaning in the confuser. We approach it using the pre-trained FastText word embedding model [6,47] to evaluate semantic closeness. The semantic substitution detector tokenizes both package names. It then checks whether at least one token is the same for both the confuser and target package names. For each differing tokens in the target and confuser, word similarity is calculated on all possible token pairs. The similarity ratio is then calculated as *sum of word similarities / number of combinations*. If the similarity ratio is greater than a predefined threshold, then the package pair is flagged. We used a linear search to identify the similarity threshold (**0.65**) leading to the best precision on our attack dataset (**1**) (in doing so, we were limited by the limited number of examples in the dataset).

**Asemantic Substitution** occurs when a token with in the target is replaced with another token, without the tokens being semantically close (e.g., "discord.js" → "discord.dll"). The detector first tokenizes target and confuser names, and checks whether there exist one or more common tokens. It then checks if the remaining tokens' similarity ratio is less than 0.65 using the same word model as semantic subsitution. To limit false positives, we only allow a single token to be substituted. Also the length of the token in the confuser must be less than twice the length of the token in the target.

**Homophonic Similarity** describes instances where the confuser name sound similar to the target's name. The detector checks whether any token in the same position in the sequence are homophonically similar, with the rest of the package names being equal. The similarity between token is assessed by evaluating them against the Metaphone [43] and Soundex [3] algorithms. The two tokens must have the same encodings for both algorithms, as different encodings implies uncertainty. Furthermore, the token coming from the target must be present in either the English or the jargon corpus (a token not present in either is unlikely to be confusable.)

**Simplification** describes a confuser name removing a prefix or a suffix from the target. The detector checks if the confuser package is a prefix or a suffix of the target package, with a restrictions: the length of the target package cannot be greater than three times that of the confuser package.

**Alternate Spelling** describes the existence of different regional spellings of common English words (e.g., "colour-string" and "color-string".) To detect such occurrences, the detector leverages a dataset of 1706 British English to American English translations. The detector looks for package names that are identical except for one or more position-matched tokens being translation of one another. A limitation of this detector is the focus on English.

**Homographic Replacement.** consists of potential confusion caused by similar-looking letters in the package names. To detect such patterns, the detector is given a list of ASCII and Unicode homoglyphs of the keyboard characters *[a-z,  , ., _, -]*. Then for each character in the target name, candidate pairs are created by replacing the character with its homoglyph counterpart, e.g m → n. If the candidate pair matches the confuser package, then the pair is flagged.

**Rule composition.** Detecting instances consisting of the composition of multiple rules is in general unsolvable as it requires to test for the composition of any possible combination of rules, with repetitions. We chose a compromise to detect simple compositions of up to three selected rules. We develop *normalizers* approximating *Delimiter Modification* (converts all common delimiters to "_"), *Sequence Reordering* (alphabetically sorts all tokens separated by delimiters), and *Scope Confusion* (removes all scoping markers). We apply combinations of normalizers to package names prior to piping them to selected other rules, allowing detection of specific cases of the three rules above in combination with others.

## 6 Measuring Package Confusion in the Wild

In this section we aim at detecting the extent to which package confusion occurs in the wild. To do so, we break our high level RQ2: *What types of package confusion exist in the wild?* in three specific sub-questions:

- *How many potential instances of package confusion exist in the npm ecosystem?* Results detailed in Section 6.1 show a significant number of potential instances (**360,333**, with **2,779** matching multiple detection rules)
- *What is the confusability of matches identified by our detection rules?* Results shown in Section 6.2 paint a complex picture, showing that while some high-yield rules consistently identify highly confusing matches, others encompass a high number of matches with low confusion potential.
- *Are existing detectors able to identify our identified attack categories?* Results in Section 6.3 show limited overlapping between state-of-art tools and our tools, suggesting that it would be useful to integrate the high-yield subset of our rules into existing detectors.
- *What is the relationship between package confusion and malicious code?* Results in Section 6.4 show that, although the presence of name confusion by itself is not sufficient to conclude maliciousness, packages whose name is confusable with that of a target have greater likelihood to be flagged as malicious than other packages.

### 6.1 Package confusion instances in npm

In this experiment, we execute our rules against package pairs in the npm snapshot detailed in Section 3.2.

#### 6.1.1 Methodology

Even after excluding *1-step D/L* and *Familiar term abuse*, there are still 11 rules to be applied to $(1.7e6)^2$ package pairs, which is impractical. In order to ensure the measurement terminates in acceptable computation/time, we apply an optimization proposed by TypoGard [58]. Highly impactful confusion incidents arise when a benign *popular* package is

| Rule | #Instance |
|---|---|
| P/s augmentation | 143864 |
| Asemantic subst. | 139160 |
| Simplification | 27743 |
| Homophonic sim. | 24735 |
| Semantic subst. | 9610 |
| Delimiter modif. | 7183 |
| Scope confusion | 4247 |
| Grammatical subst. | 2461 |
| Homographic repl. | 2393 |
| Sequence reord. | 1734 |
| Alternate spelling | 21 |

Table 4: Matches in npm for each category

confusable with a malicious *unpopular* package. The insight is that a popular package is a more damaging target than an unpopular one, and more likely to be chosen as target.

Following the established methodology described above, we partition the npm package set between popular and unpopular packages using a threshold of 15,000 downloads/week, as proposed by TypoGard's authors. As we only consider popular packages as targets and unpopular ones as confusers, the number of matches is reduced to $1,727,553 \times 24871$ making ecosystem-scale analysis practical. We execute the analysis on a SLURM cluster at our institution, consisting of 1000+ x86 CPUs and 8+ TB of aggregated RAM.

### 6.1.2 Results

Overall, our rules flagged **360,333** package pairs, which amount to **0.00087%** of all considered pairs (i.e., one pair every 114,470.) On average, analysis took **0.22ms/pair**[2]. Table 4 shows the number of matches per category. The total number of matches is higher than the total number of packages as there are **2779** pairs matching multiple rules concurrently. The most common rule combinations in composite matches are *Homophonic similarity & Prefix/ suffix augmentation*, *Delimiter modification & Sequence reordering*, and *Delimiter modification & Grammatical substitution*.

While these results suggest a high upper bound on the number of instances of semantic confusion, it is important to point out that the rules by necessity *approximate* the mechanism that may induce confusion in humans. As such, not all the matches may be confusing to humans in practice. The next section explores this issue.

## 6.2 Confusability Analysis of Matches

In this section, we perform an analysis of a random sample of package pairs flagged by our detection rules, to determine their potential to induce confusion in developers. As discussed in Section 2.2, confusable packages may generate security issues even without explicit intent. Thus, we investigate package confusion as a separate consideration from maliciousness, which we explore in Section 6.4. Hence, this section focuses on the confusability potential of analyzed package pairs.

### 6.2.1 Methodology

We designed an online survey assessing perceived confusability of randomly selected package pairs, by participants with software development experience. This methodology has the advantages of being assumption-free (i.e., it focus on package names, without making assumptions on the context were developers may encounter such packages), and efficient. The survey underwent ethics review and approval.

**Participant Recruiting.** To ensure an appropriate population, we advertised the survey via email to computing-related majors at our institutions, further extending reach through snowball sampling. Applicants were then vetted via a prescreening survey. We excluded applicants with no software development experience, and/or who could not correctly name one ecosystem/package. All applicants entered a raffle awarding one $50 gift card. Participants who completed the study also received a $10 gift card.

**Study Protocol.** Participants were directed to a survey presenting the following Likert-scale question: *On a scale of 1 to 6, how likely are you to misremember or mistype the package in column P instead of the corresponding package in column V?*. The column P contains the confuser package names and column V contains the target package names. The question was followed by a list of 50 package pairs, randomly sorted and selected from a sample generated as follows. We included 100 randomly selected samples for each detection rule, except for *Alternate spelling* which only returned 21 matches across the whole ecosystem. At the conclusion of the study, we only retained pairs for which three or more ratings had been received (to guarantee analysis consistency, if more than three ratings had been received, we randomly selected three). Furthermore, we also included limited samples from a control set of 100 package pairs generated by randomly pairing benign package names. Prior to data analysis, we removed incomplete and improperly completed answers (e.g., answers which gave the same score to all package pairs).

### 6.2.2 Results

After data cleanup we retained results for $n = 64$ participants, rating **784** packages. Empirically, we observed that results may still contain noise (e.g., some high confusability ratings to randomly paired and semantically different names). To avoid cherry-picking data, we decided to exclude only obviously incorrect answers (e.g., answers giving the same rating to all entries) and retain everything else. Ultimately, this

[2] We could not analyze 4% of all package pairs due to a SLURM-related issue we were unable to resolve.

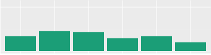
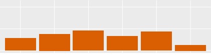
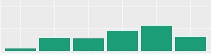
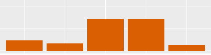
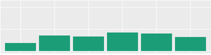
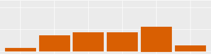
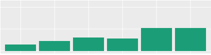
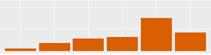
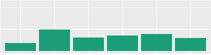
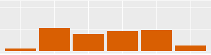
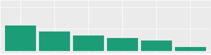
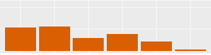
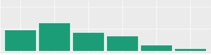
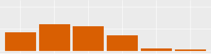
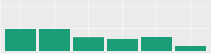
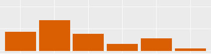
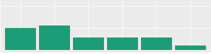
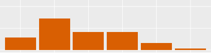
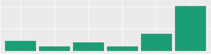
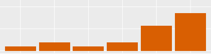
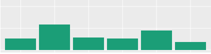
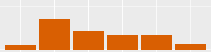
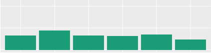
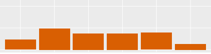
| Rule | Rating Distribution (1-6) | Median Distribution(1-6) | n samples | %(2+r≥ 4) | %(3r≥ 5) |
|---|---|---|---|---|---|
| P/s augmentation |  |  | 79 | 44% | 2.5% |
| Sequence reord. |  |  | 58 | 79% | 10% |
| Delimiter modif. |  |  | 78 | 56% | 7.7% |
| Grammatical subst. |  |  | 77 | 74% | 18% |
| Scope confusion |  |  | 84 | 52% | 4.8% |
| Semantic subst. |  |  | 83 | 31% | 0.0% |
| Asemantic subst. |  |  | 86 | 21% | 0.0% |
| Homophonic sim. |  |  | 78 | 24% | 3.8% |
| Simplification |  |  | 78 | 29% | 1.3% |
| Alternate spelling |  |  | 21 | 81% | 38% |
| Homographic repl. |  |  | 62 | 39% | 6.5% |
| **Overall** |  |  | **62** | **45%** | **6.1%** |

Table 5: Confusability ratings for samples from each category. Column 2 shows the distribution of all ratings per category. Column 3 shows the distribution of medians computed for each packet pair. "%(2+r ≥ 4)" expresses the fraction of samples receiving a confusability rating of 4 or above from at least two rater. "%(3r≥ 5)" expresses the fraction of samples receiving a confusability rating of 5 or above by all raters.

dataset suffices for our main purpose, which is high-level quantitative analysis of our detection rules.

Packages per category vary between **58** (excluding *Alternate spelling*) and **86**. We also obtained **69** single-rater ratings for packages in the control group (scores for such packages remain fairly consistently low, thus we limited their presence in the survey, in the interest of maximizing the number of scores collected for confusable packages). Table 5 presents results. For matches labeled with multiple categories, we map each match to the highest-priority category according to Table 2. For each detection category, we show (i) the rating distribution for each rater, (ii) the percentage of packages marked ≥ 4 by two or more out of three raters (interpreted as being potentially confusing), and (iii) the percentage of packages marked ≥ 5 by all raters (highly likely to be confusing).

#### 6.2.3 Review of results

Three categories exhibit non-negligible ratios of highly confusable pairs (> 10%). Those include *Sequence reordering*, *Grammatical substitution*, and *Alternate spelling*. The same category also exhibit > 70% pairs being rated potentially or highly confusing. *Delimiter modification* and *Scope confusion* return limited matches which meet the "highly confusing" criterion, but > 50% of their matches meet the "potentially confusing" criterion.

Intuitively, it appears that the most effective rules aim at relatively simple textual modifications, which can be accurately modeled by our detectors. To further investigate, we measured Spearman correlation between number of matches $M_P$ meeting the "potentially confusing" criterion, and precision, recall and F1-score of each detection rule on the attack dataset (ref. Table 3). We do the same for number of matches $M_H$ meeting the "highly confusing" criterion. $M_P$ exhibits strong correlation with recall ($r(11) = .71,\ p = 0.014$) and F1-score ($r(11) = .82,\ p = 0.002$). Likewise, $M_H$ exhibits strong correlation with recall ($r(11) = .83,\ p = 0.002$) and F1-score ($r(11) = .71,\ p = 0.014$). This is consistent with our hypothesis that simpler rules that more accurately model the underlying confusion lead to higher-quality matches.

To determine which rules pick up *any* confusability potential at all, we conducted pairwise comparisons between ratings in the control group and each category. To account for the fairly complex dependencies between samples (same rater scoring multiple pairs/categories, multiple raters scoring the same pair), we fit an ordinal regression model to the data, and performed post-hoc pairwise comparison between groups (rules) using Tukey's HSD method (we also performed a simpler permutation test with Holm-Bonferroni correction, with qualitatively identical results). We found that 8 out of 11 categories exhibit a statistically significant difference from the control group, i.e., those related to randomly paired benign packages (details omitted for brevity). The only exceptions are *Semantic Substitution Asemantic Substitution*, and *Homophonic Similarity*, which is consistent with those being the lowest-performing categories. This result suggests that package pairs identified by all but the lowest-yield rules exhibit some degree of confusability.

Finally, we performed a manual review of the lowest-performing rules. The quality of matches seems to depend on factors, such as the word structure of similar-sounding names. For example, in "Homophonic similarity", `babel-jest/buble-jest` meets our "potentially confusing" criterion, but `lazy.js/lice-js` does not. Building a detector that considers such perceptual nuances is future work.

## 6.3 Comparison to State-of-the-art detectors

In this section, we compare our detection rules with two state-of-the-art package confusion detection systems: Taylor et al.'s TypoGard tool [58], and Microsoft OSSGadget's typosquatting detection component [1]. Like our work, these detectors apply checks to package pairs to identify package confusion candidates. However, these tools focus on small syntactic differences for detection, as opposed to our emphasis on semantic confusion. Thus, the goal of our evaluation is not to asses whether our rules subsume these tools. Rather, we consider whether our rules *complements* them by returning matches that those tools fail to identify.

### 6.3.1 Methodology

In order to carry out our comparative analysis, we execute both previous tools on the set of package pairs described in Section 6.1.1. For both tools, we use the public implementation from GitHub. We made minor modifications to OSSGadget's public implementation (which do not affect the detection logic) to adapt it to our evaluation harness. Similar to our tooling, both TypoGard and OSSGadget associate each match to a specific detection category. After execution, we consider the *overlap* between each TypoGard/OSSGadget category and our categories. Overlap is defined as the fraction of matches returned by one of our rules which is also returned by a given TypoGard/OSSGadget rule. Furthermore, we analyze differences in user ratings between matches returned by both our approach and OSS, and by our approach only (TypoGard did not flag any package pair for which we have user ratings).

| Rule | Matches | TG Ov. | OG Ov. |
|---|---|---|---|
| P/s augmentation | 143864 | 96 | 1484 |
| Sequence reord. | 1734 | 407 | 0 |
| Delimiter modif. | 7178 | 300 | 847 |
| Grammatical subst. | 2461 | 169 | 1084 |
| Scope confusion | 4247 | 8 | 0 |
| Semantic subst. | 9610 | 3 | 33 |
| Asemantic subst. | 139160 | 1 | 32 |
| Homophonic sim. | 24735 | 374 | 2680 |
| Simplification | 27743 | 74 | 792 |
| Alternate spelling | 21 | 1 | 8 |
| Homographic repl. | 2393 | 115 | 1913 |

Table 6: Overlap between matches returned by our rules ("Matches"), and matches returned by TypoGard ("TG Ov.") and OSSGadget ("OG Ov.").

### 6.3.2 Results

Overall, TypoGard returned **3690** matches, and OSSGadget returned **31501** matches. Of those, respectively **1924 (52.1%)** and **8386 (26.6%)** are also returned by our rules; however, the majority of matches returned by our rules were not returned by either tool.

Table 6 presents, for each one of our rules, overall number of matches (column 2); matches also returned by TypoGard (column 3); and matches also returned by OSSGadget (column 4). We further discuss relevant overlap instances below.

Figure 4 summarizes the degree of overlap between our rules and individual TypoGard/OSSGadget rules as a heatmap. Our Grammatical substitution rule exhibits a moderate (**25%**) overlap with OSSGadget's *Suffix added*; this is unsurprising as some (but not all) grammatical alterations involve adding a suffix to the base word. The only combination with major overlap is between our Homographic replacement rule and OSSGadget's *Ascii Homoglyph*. Upon close inspection, both rules aim at detecting homographic stand-ins (i.e., different characters that look similar), but are engineered slightly differently, leading to **80%** overlap. We further discuss the implications of these results in Section 7.

Table 7 characterizes user ratings of matches returned by both OSSGadget and our rules, and those returned by our rules only. Overall, these results suggest that the design space

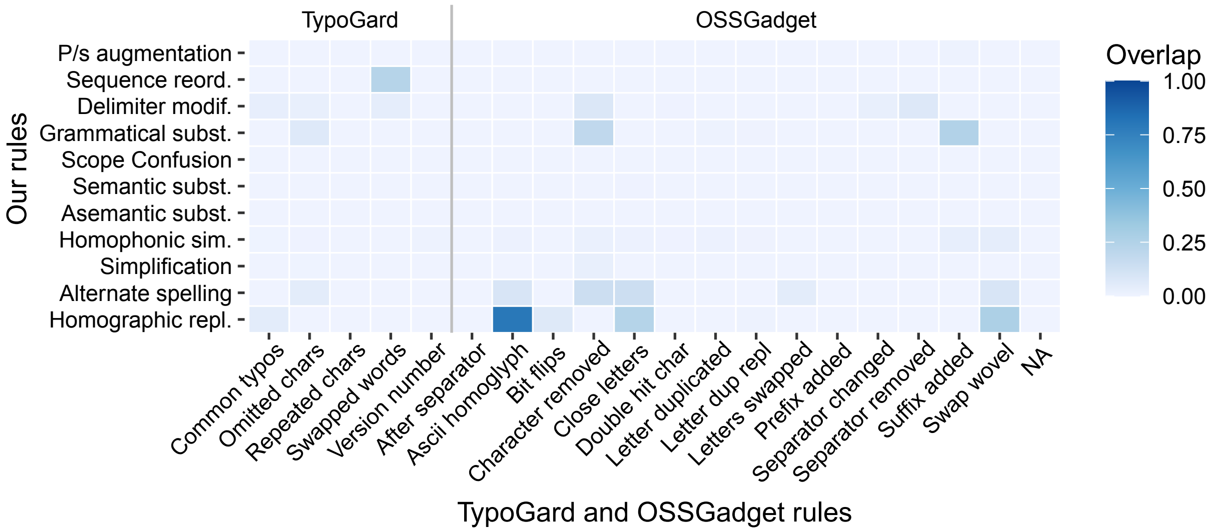


Figure 4: Degree of overlap between our rules (Y-axis) and TypoGard/OSSGadget rules (X-axis). Darker hues corresponds to larger overlaps (i.e., closer to 1.0).

of certain rules may allow trade-offs between precision and recall. OSSGadget focus on simple textual modifications results in high fraction of potentially or highly confusable matches, among overlapping matches. However, for the same reason the tool also fails to capture a significant amount of potentially relevant package pairs. We leave exploring combination of techniques from different tools as future work.

## 6.4 Security analysis of packages

In this section, we perform an analysis of the maliciousness of potentially confusable packages to determine the security risks they pose to developers. We gathered data from sources of known malicious packages, to determine which flagged packages had been taken down as malicious. We then compared malicious package density in the set of packages flagged by our rules, vs the population of not flagged packages.

**Methodology.** We leveraged Snyk, a popular third-party database which is commonly used in related work [15, 69]. We further requested malicious package data sets from a number of previous publications [51, 54, 58]. We matched all the unique perpertrator packages returned by our tool ($n = 210,741$) against the datasets described above, collecting all packages matching any dataset. To provide a control group, we also matched $n = 150,000$ packages not flagged by our tool against the same datasets. Using Snyk poses an internal validity issue as the same resource is part of the data sources used to define the rules, however it is necessary due to the scarcity of open dataset of software supply chain attacks. We consider this limitation acceptable as the purpose of this experiment is to characterize known malicious behavior in confusable packages, rather than measure intrinsic accuracy of our detection rules (the latter would require ground truth for the $210,741$ packages, which is impractical). Packages flagged by Snyk may not appear in our attack dataset for two reasons: (i) they were not present in the Snyk database at the time of collection; or (ii) they were present in the Snyk database but not clearly characterized as package confusion (although they are confusing according to our rules).

Further, we categorized flagged malicious packages according to the taxonomy of Duan et al. [67]. We find it necessary to extend the taxonomy with two new categories: *Cryptotheft*, which steal crypto currencies from wallets, and *Downloader* which downloads malicious payloads on a system. Finally, some packages were flagged as malicious without further description, and we marked those as *Unknown*.

**Results.** Overall, **168** confuser packages flagged by rules 1-11 presented known malicious behavior (**0.080% of all confusers**). Detailed results are showed in Table 8. If we also consider packages matched by *1-step D/L* (not detailed in the table), we obtain **278** malicious packages out of $n = 275,910$ (**0.1%**). Furthermore, **40** packages in the control group where flagged as malicious (**0.027%**). While the numbers are small in absolute terms, they also evidence that a package marked by our rules is **3** times more likely to be flagged as malicious than an unmarked package (**3.8** when also including *1-step D/L*). This suggests that name confusion is a relevant component in supply chain attacks, and both syntax- and semantics-based confusion contribute to it. Thus, we believe identifying confusable package pairs has a place in a broader security strategy, both for detection and for decision support.

## 6.5 Disclosure

Package confusion incidents represent a grey area of security issues when the content of a confuser package is not overtly

| Rule | OSS total | OSS 2+r≥ 4 | OSS 3r>5 | NO total | NO 2+r≥ 4 | NO 3r>5 |
|---|---|---|---|---|---|---|
| P/s augmentation | 0 | N/A | N/A | 79 | 44% | 2.5% |
| Sequence reord. | 0 | N/A | N/A | 58 | 79% | 10% |
| Delimiter modif. | 20 | 85% | 0.0% | 58 | 47% | 10% |
| Grammatical subst. | 42 | 81% | 24% | 35 | 66% | 11% |
| Scope confusion | 0 | N/A | N/A | 84 | 52% | 4.8% |
| Semantic subst. | 0 | N/A | N/A | 83 | 31% | 0.0% |
| Asemantic subst. | 0 | N/A | N/A | 86 | 21% | 0.0% |
| Homophonic sim. | 9 | 78% | 33% | 69 | 17% | 0.0% |
| Simplification | 0 | N/A | N/A | 78 | 29% | 1.3% |
| Alternate spelling | 8 | 75% | 38% | 13 | 85% | 38% |
| Homographic repl. | 48 | 46% | 8.3% | 14 | 14% | 0.0% |

Table 7: Characterization of user ratings between overlapping ("OSS") and non-overlapping ("NO") matches. "Total": total number of matches, "2+r≥ 4": % of potentially confusing matches, "3r>5": % of highly confusing matches

.

| Attack Category | # pkgs | # in attack set |
|---|---|---|
| Stealing | 70 | 48 |
| Backdoor | 9 | 9 |
| Sabotage | 2 | 1 |
| Cryptojacking | 2 | 2 |
| Virus | 1 | 1 |
| Maladvertising | 2 | 1 |
| PoC | 1 | 0 |
| Cryptotheft | 33 | 31 |
| Downloader | 1 | 0 |
| Confusion | 2 | 0 |
| Unknown | 45 | 18 |

Table 8: Breakdown of malicious packages flagged by our rules. *# pkgs* is the number of flagged packages per category, while *# in attack set* is the number of such packages present in the attack dataset upon which the rules are based (Section 3)

malicious. Such a package may still degrade the quality of a project in which it is accidentally used, by unintentionally introducing potentially unmaintained, vulnerable code into a project [66, 70].

We disclosed the results of our package confusion analysis directly to npm so that they could apply an appropriate mitigation. The Trust and Safety team acknowledged our submission, but they have not rendered a final decision by the time of publication. We note that the security issue presented by package confusion does not fit well into the existing reporting and takedown mechanisms of package repositories. The npm staff advised us to use the *Report Malware* feature in our reports, and categorize the incidents as *Typosquatting*. They also noted that investigations rely on determining violations of npm's Open Source Terms, which does not include preclude confusing content. Furthermore, a confuser package may be confusing unintentionally, and may not contain any malcious code. We believe that mitigating package confusion is a complex issue that requires future work.

### 6.6 Data/Artifact Release

Our datasets, results, and tooling are available at `https://osf.io/nfkts/?view_only=b56d63194ef84ce4ba85ec00ee57cd05` and `https://github.com/ldklab/typomind-release`.

## 7 Discussion

In this section, we discuss the results of our previous experiments, and describe their implications.

**Importance of Broad Package Confusion Detection.** The results of our confusability analysis demonstrates that *each* of the categories have potentially-confusing entries in practice and that some methods are highly capable of confusability (*c.f.* Section 6.2.3). This result implies that our categories do capture actual methods by which an adversary might induce a developer to download a package. This data emphasizes that the problem of package confusion attacks are a credible threat, and that our categorization can help to specify how the attack might occur in practice.

**Uniqueness of Package Confusion.** One of the key results of Section 6 is that complex contortions of package names are less effective than simple textual replacement. As such, we tested whether an expanded notion of package confusion is subsumed in practice by state-of-the-art detectors that focus on typosquatting. Our comparative analysis against the TypoGard and OSSGadget's tools indicate that our categories do add a new dimension to package confusion, and that package confusion is not captured by typo detection alone.

**Feasibility of Automated Detection.** Our results have implications for automatic detection of package confusion attacks to alert users. We note that the detectors for *Sequence reordering*, *Grammatical substitution* and *Alternate spelling* work quite well in practice, resulting in an aggregate 77% matches marked as potentially confusing or highly confusing and 18% being extremely confusing. Cumulatively, these detectors also flag less than one package pair per 100M, limiting the risk of warning fatigue [21]. These detectors may be immediately applicable in proactively preventing these types of confusion (whether malicious or unintentional). However, several of the other detectors would require additional data (construed as vetted attack examples) to better refine their design. For example, *Semantic substitution* and *Asemantic substitution* resulted in matches unlikely to generate confusion. This is in a sense unsurprising, as designing complex detectors based on very limited sets of examples is a significant challenge. Nevertheless, we believe our work may serve as a call to arms for future study on incidents and detectors for these categories.

**Deploying Package Confusion Prevention.** The results of our experiment on package confusion instances on npm shows that relatively few of the packages within the ecosystem (0.00087%, *c.f.* Section 6.1.2) are likely to be package confusion attacks. However, as with the narrow threat of typosquatting, package confusion attacks have an outsized impact on the overall security of the repository, especially since such attacks have historically exposed many users to their effect. Based on the low overall alert rate of our rules, deploying such tools in practice is unobtrusive. Since our tooling aims to detect *potential* confusion, it is best served as a warning mechanism rather than a irrevocable blocking policy to prevent package installation.

## 8 Related Work

**Supply-Chain Security:** Our work represents an entry in the growing body of literature on characterizing an detecting supply-chain security issues [28,34,52,53,62,63,65]. Related to our work is also the literature on software bill of materials (SBOM), which seeks to track the developers and components of software [46, 49]. Other relevant works focus on community repository security [39,60], binary transparency [14], and integrity verification [50]. Ecosystem maintainers have also been involved in security, including shipping hardware verification keys to popular projects [32]. The body of work detailed above does not specifically address package confusion as an issue within the software supply chain.

Other works focus specifically on npm security. Xiao et al [67] study vulnerabilities in the Node.js programs through hidden property abusing. Sejfia et al. [54] build a practical automated detection technique to find malicious npm packages. Zahan et al. [68] analyze the metadata of 1.6 million npm packages, identifying various signals of security weakness. Our research complements these investigations by providing a rich, in-depth characterization of package confusion.

Ohm et al. [51] analyze 174 malicious packages in npm, ruby and python libraries and found that almost 64% utilize typosquatting, which is consistent with some of our findings. Zerouali et al. [69] and Alfadel et al. [15] analyze vulnerable packages in PyPi and npm and Rubygems respectively, by leveraging the Snyk database. We perform similar analyses as an evaluation criteria for our results.

**Namesquatting:** The problem of name confusion in other contexts, broadly referred to as namesquatting, is similar in spirit to name-based package confusion attacks. Ladisa et. al [40] define *Create Name Confusion with Legitimate Package* as one of the taxonomy for OSS supply chain attacks. Our work details ways we can detect such confusing package names and tests it in the wild. Moubayed et al. propose a technique for detecting DNS squatting [48], and Hu et al. propose similar work for detecting squatting in mobile app markets [35]. Burt et al. studied brand-name confusion in the context of trademarks, noting that confusion can arise in a variety of circumstances but offering no concrete categorization or automated mitigation/detection [20]. Our research focuses specifically on categorizing known package confusion instances and builds detection rules to detect these categories in the wild and uses domain-specific information, such as a targeted jargon dictionary.

**Package Name Typosquatting Defenses:** Ecosystem maintainers implemented several measures to mitigate the threat of typosquatting, such as PyPI disallowing packages to occupy a name that only replaces underscores with dashes. Several recent works have also attempted to characterize [38,64] or prevent [58,61] typosquatting, intended as limited forms of package confusion such as typos. As stated in Section 1, our work is designed to expand upon these limited protections.

**Cognitive Models:** Moving beyond basic lexical similarity, previous work has examined several cognitive aspects of name similarity as they relate to several areas outside of security. One such area is drug prescribing errors due to drug names being too similar [41,42]. This research described the "look-alike/sound-alike" model (LASA). For the original model, techniques including n-grams, longest common subsequence, and Levenshtein edit distance were employed to detect "look-alike" instances while techniques including Soundex, Phonex, Editex, tapered edit distance, omission key, and skeleton key were utilized in the detection of "sound-alike" instances. Our research utilizes some of these cognitive models in package names to understand package name confusion in developers.

## 9 Conclusion

In this work, we expand upon the study of package confusion attacks, proposing categories that captures historical examples of such attacks. We show that these categories also represent credible means of misleading users of language-based ecosystems. We propose several heuristics, based on these categories, for automatically identifying examples of package confusion in the wild. We show that there are numerous such instances in the npm repository, and encourage future work in the domain.

## Ackowledgments

We thank the anonymous reviewers and our shepherd for their insightful feed- back, that greatly aided us in improving this work. We also thank Ann Barcomb for her help with the survey design, and Sirshendu Ganguly and James Kajon for their help with the package analysis process. This work was partially supported by a generous gift from the Google Open Source Security Team, and by funding from the Worcester Polytechnic Institute Computer Science department.

## References

[1] Microsoft OSS Gadget. https://github.com/microsoft/OSSGadget/tree/main/src/oss-find-squats-lib.

[2] RubyGems: Guide. https://guides.rubygems.org/, last accessed October 2022.

[3] Soundex System. https://www.archives.gov/research/census/soundex, August 2016.

[4] Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, May 2021.

[5] CVE - CVE. https://cve.mitre.org/index.html, October 2022.

[6] fastText. https://fasttext.cc/index.html, oct 2022.

[7] GitHub Advisory Database. https://github.com/advisories, October 2022.

[8] NLTK :: Natural Language Toolkit. https://www.nltk.org/, oct 2022.

[9] npm advisories. https://github.com/advisories?query=type%3Areviewed+ecosystem%3Anpm, oct 2022.

[10] Python Packaging Advisory Database. https://github.com/pypa/advisory-database, September 2022.

[11] Registries - The Cargo Book. https://doc.rust-lang.org/cargo/reference/registries.html, October 2022.

[12] Ruby Advisory Database. https://github.com/rubysec/ruby-advisory-db, October 2022.

[13] Snyk Open Source Advisor. https://snyk.io/advisor, October 2022.

[14] Mustafa Al-Bassam and Sarah Meiklejohn. Contour: A practical system for binary transparency. *CoRR*, abs/1712.08427, 2017.

[15] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *SANER*, 2021.

[16] Andrey Polkovnychenko and Shachar Menashe. Malicious Packages in npm Targeting Azure Developers. https://jfrog.com/blog/large-scale-npm-attack-targets-azure-developers-with-malicious-packages/, March 2022.

[17] Ax Sharma. PyPI removes 'mitm-proxy2' over code execution concerns. https://www.bleepingcomputer.com/news/security/pypi-removes-mitmproxy2-over-code-execution-concerns/, October 2021.

[18] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *ACM CCS*, 1999.

[19] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006.

[20] Jennifer Burt, Kimberley McFarlane, Sarah Kelly, Michael Humphreys, Kimberlee Weatherall, and Robert Burrell. Brand name confusion: Subjective and objective measures of orthographic similarity. *Journal of Experimental Psychology: Applied*, 23, 02 2017.

[21] Rainer Böhme and Jens Grossklags. The security cost of cheap user interaction. In *NSPW*, 2011.

[22] Catalin Cimpanu. Ten Malicious Libraries Found on PyPI - Python Package Index . https://www.bleepingcomputer.com/news/security/ten-malicious-libraries-found-on-pypi-python-package-index/, 2017.

[23] Catalin Cimpanu. Twelve malicious Python libraries found and removed from PyPI. https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/, October 2018.

[24] Catalin Cimpanu. Malicious Python packages caught stealing Discord tokens, installing shells. https://therecord.media/malicious-python-packages-caught-stealing-discord-tokens-installing-shells/, November 2021.

[25] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.

[26] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *IS NDSS*, 2021.

[27] Ceryn Evans and Jamie Lewis. Analysing semi-structured interviews using thematic analysis: Exploring voluntary civic participation among adults. *Sage Research Methods Datasets Part 1*, 2018.

[28] Dan Geer, Bentz Tozer, and John Speed Meyers. Counting broken links: A quant's view of software supply chain security. *For Good Measure*, 2019.

[29] GitHub. Mailicious Package in erquest. https://github.com/advisories/GHSA-4pmg-jgm5-3jg6, 2020.

[30] GitHub. Malicious Package in bufffdr-xor. https://github.com/advisories/GHSA-8549-p68h-m9mc, 2020.

[31] GitHub. Malicious Package in koa-body-parse. https://github.com/advisories/GHSA-wqgq-mfvj-6qxp, 2020.

[32] Dan Goodin. Actors behind pypi supply chain attack have been active since late 2021. https://arstechnica.com/information-technology/2022/09/actors-behind-pypi-supply-chain-attack-have-been-active-since-late-2021/, September 2022.

[33] Grant Jenks. Python Word Segmentation — Word Segment 1.3.1 documentation. https://grantjenks.com/docs/wordsegment/, 2017.

[34] Trey Herr. Breaking trust – shades of crisis across an insecure software supply chain, 2021. USENIX ENIGMA.

[35] Yangyu Hu, Haoyu Wang, Ren He, Li Li, Gareth Tyson, Ignacio Castro, Yao Guo, Lei Wu, and Guoai Xu. Mobile app squatting. In *WWW*, 2020.

[36] Ivan Akulov. Malicious packages in npm. Here's what to do. https://iamakulov.com/notes/npm-malicious-packages/, 2017.

[37] Danielle Jacobs and Troy McDaniel. A survey of user experience in usable security and privacy research. In *HCII*, 2022.

[38] Berkay Kaplan and Jingyu Qian. A survey on common threats in npm and pypi registries. In *MLHat Workshop*, 2021.

[39] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using delegations to protect community repositories. In *NSDI*, 2016.

[40] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Taxonomy of attacks on open-source software supply chains. *CoRR*, abs/2204.04008, 2022.

[41] Bruce L Lambert, William Galanter, King Lup Liu, Suzanne Falck, Gordon Schiff, Christine Rash-Foanio, Kelly Schmidt, Neeha Shrestha, Allen J Vaida, and Michael J Gaunt. Automated detection of wrong-drug prescribing errors. *BMJ Quality & Safety*, 28(11):908–915, 2019.

[42] Bruce L. Lambert, Swu-Jane Lin, Ken-Yu Chang, and Sanjay K. Gandhi. Similarity as a risk factor in drug-name confusion errors: The look-alike (orthographic) and sound-alike (phonetic) model. *Medical Care*, 37(12):1214–1225, 1999.

[43] Lawrence Philips. Hanging on the metaphone. *Computer Language Magazine*, 7(12):39–44, December 1990.

[44] Markus Lennartsson, Joakim Kävrestad, and Marcus Nohlberg. Exploring the meaning of usable security–a literature review. *Information & Computer Security*, 29(4):647–663, 2021.

[45] Tomislav Maljic. Mining for malicious Ruby gems. https://blog.reversinglabs.com/blog/mining-for-malicious-ruby-gems, April 2020.

[46] Robert Alan Martin. Visibility & control: addressing supply chain challenges to trustworthy software-enabled things. In *IEEE SSS*, 2020.

[47] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[48] Abdallah Moubayed, MohammadNoor Injadat, Abdallah Shami, and Hanan Lutfiyya. Dns typo-squatting domain detection: A data analytics & machine learning based approach. In *IEEE GLOBECOM*, 2018.

[49] Éamonn Ó Muirí. Framing software component transparency: Establishing a common software bill of material (sbom). 2019.

[50] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive Software-Update transparency via collectively signed skipchains and verified builds. In *USENIX Security*, 2017.

[51] 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.

[52] 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 *ACM SCORED Workshop*, 2022.

[53] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. On the feasibility of detecting injections in malicious npm packages. In *ARES*, 2022.

[54] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *IEEE/ACM ICSE*, 2022.

[55] Ax Sharma. Sonatype Catches New PyPI Cryptomining Malware. https://blog.sonatype.com/sonatype-catches-new-pypi-cryptomining-malware-via-automated-detection, June 2021.

[56] SonaType. State of the Softwarw Supply Chain. Technical report, 2021.

[57] Sidra Taha. Can a code snippet portal contribute to greater learning outcomes in other fields of science and technology? Master's thesis, UiT Norges arktiske universitet, 2021.

[58] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Defending Against Package Typosquatting. In *NSS*, 2020.

[59] Gareth Terry, Nikki Hayfield, Victoria Clarke, and Virginia Braun. Thematic analysis. *The SAGE handbook of qualitative research in psychology*, 2:17–37, 2017.

[60] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *USENIX Security*, 2019.

[61] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. Security issues in language-based sofware ecosystems. *CoRR*, abs/1903.02613, 2019.

[62] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. Lastpymile: identifying the discrepancy between sources and packages. In *ACM ESEC/FSE*, 2021.

[63] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Towards using source code repositories to identify software supply chain attacks. In *ACM CCS*, 2020.

[64] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Typosquatting and combosquatting attacks on the python ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 509–514. IEEE, 2020.

[65] Zachary Williams, Jason E Lueg, and Stephen A LeMay. Supply chain security: an overview and research agenda. *The International Journal of Logistics Management*, 2008.

[66] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork?: Finding hidden code clones in npm. In *IEEE/ACM ICSE*, 2022.

[67] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the node.js ecosystem. In *USENIX Security*, 2021.

[68] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *IEEE/ACM ICSE-SEIP*, 2022.

[69] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Softw. Engg.*, 27(5), sep 2022.

[70] 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*, 2019.