Defending Against Package Typosquatting

Matthew Taylor[1], Ruturaj Vaidya[1], Drew Davidson[1], Lorenzo De Carli[2], and Vaibhav Rastogi[3⋆]

[1] University of Kansas, Lawrence, KS, USA
{mjt, ruturajkvaidya, drewdavidson}@ku.edu,
[2] Worcester Polytechnic Institute, Worcester, MA, USA
ldecarli@wpi.edu
[3] University of Wisconsin-Madison, Madison, WI, USA
vaibhavrastogi@google.com

Abstract. Software repositories based on a single programming language are common. Examples include npm (JavaScript) and PyPI (Python). They encourage code reuse, making it trivial for developers to import external packages. Unfortunately, the ease with which packages can be published also facilitates *typosquatting*: uploading a package with name similar to that of a highly popular package, with the aim of capturing some of the popular package's installs. Typosquatting frequently occurs in the wild, is difficult to detect manually, and has resulted in developers importing incorrect and sometimes malicious packages.
We present TypoGard, a tool for identifying and reporting potentially typosquatted imports to developers. TypoGard implements a novel detection technique, based on the analysis of npm and PyPI. It leverages a model of lexical similarity between names, and incorporates the notion of package popularity. It flags cases where unknown/scarcely used packages would be installed in place of popular ones with similar names, before installation occurs. We evaluated TypoGard on both npm, PyPI and RubyGems, with encouraging results: TypoGard flags up to 99.4% of known typosquatting cases while generating limited warnings (up to 0.5% of package installs), and low overhead (2.5% of package install time).

1 Introduction

Package managers automate the complex task of deploying 3rd-party dependencies into a codebase, by transitively resolving and installing all code upon which a given package—which the user wishes to install—depends. One of the most common uses of package managers is in the context of large repositories of code packages based on a single programming language. Package managers are undeniably useful, with open, free-for-all repositories like npm for Node.js, PyPI for Python, and RubyGems for Ruby, collectively serving billions of packages per week. However, they also come with problems.

The open, uncurated nature of these repositories means that any developer can upload a package with a name of their choosing. This circumstance gives rise to *typosquatting*, whereby a developer uploads a "perpetrator" package that is *confusable* with an existing "target" package due to name similarity. As a result the user, intending to install the target package, may accidentally request the confusable perpetrator package. Determining why perpetrator packages are created is a challenging and ill-defined problem, as solving it requires inferring

⋆ Currently employed at Google.

the intent of the package author. The perpetrator may wish to confuse users into installing a malicious payload, seek to increase the visibility of their own benign code, or create a confusable name by happenstance. A perpetrator might even upload a placeholder package to prevent an attacker from leveraging the name. The result is the same: users are confused into importing the incorrect package.

As part of our work, we identified known typosquatting incidents by reviewing security advisories from npm, PyPI and RubyGems. Overall, this yielded 1,002 incidents in the last 3 years. While typosquatting campaigns routinely make the news [24,14,16,13], discovering instances is a laborious process as no tools exist that can warn developers or ecosystem maintainers of potential occurrences. From the point of view of a human analyst, identifying typosquatted packages is difficult and time-consuming, as typosquatting confuses manual inspection by definition. The scope of typosquatting is also far-reaching, due to the inter-dependent nature of packages: not only are the developers that make a typo exposed to unintended code, so are package that *transitively* depend on it.

Undetected typosquatting has numerous detriments, both to developers who integrate a perpetrator package, and to the end-users. An overtly malicious perpetrator may include Trojan functionality that attacks the client [34,17]. Additionally, many package managers invoke configuration hooks bundled with the package at install time, often manifested as shell scripts that run with the privileges of the user. Multiple packages that open reverse shells when installed have been removed from npm [4,33,35]. Even in cases where the perpetrator is not malicious, it can confuse the user and weaken the integrity of the system. Ironically, a well-intentioned perpetrator might clone a victim to keep it out of the hands of an attacker but allow the clone to fall behind as the target is updated.

In this work, we develop TypoGard, a novel typosquatting detection technique to discover and prevent incidents of typosquatting before they can damage the user. TypoGard can be used to detect typosquatting incidents before they happen, or to detect possible perpetrator packages within a package repository. TypoGard is designed to work client-side, and requires no special cooperation on the part of repositories. We also find that our detection techniques are highly generalizable, and show high recall across several repositories. TypoGard's core insight is to identify unpopular packages whose name is close, according to a notion of lexical similarity, to the name of a popular package. It this condition is met, TypoGard issues an alert before the package is fetched and suggests the likely-correct victim package name. During the course of our experiments, TypoGard also identified a popular and previously unknown instance of typosquatting: the npm package *loadsh*, which was typosquatting *lodash*, providing an outdated version of the code vulnerable to prototype pollution. After our report, the npm security team confirmed our finding and deprecated *loadsh*.

Our work makes the following contributions:

- We present TypoGard, an enhancement to the package manager front-end which protects users against typosquatting.
- We evaluate the performance and efficacy of TypoGard, showing that it has a high TPR of **88.1%**, while being non-intrusive.

	npm	**PyPI**	**RubyGems**
Packages	1,221,705	221,041	157,410
Weekly Downloads	17,872,179,641	997,624,343	154,954,144
Avg. Dep. Tree Size	57.27	4.58	9.61

Table 1: Repository statistics for npm, PyPI, and RubyGems.

– We show that, while our design is based on npm and PyPI, it generalizes well: TypoGard achieves **99.4%** TPR on RubyGems typosquatting attacks.

2 Background

2.1 Package Repositories

Package repositories are very popular: they encourage code reuse, and allow well-vetted, expertly-written codebases to be deployed by more developers. For these reasons, successful repositories have grown to enormous size. The first two rows of Table 1 show the current size and weekly download counts for npm, PyPI, and RubyGems. As the table shows, they contain hundreds of thousands (PyPI, RubyGems) to millions (npm) of publicly available packages. The total number of weekly downloads ranges from hundreds of millions to many billions.

Much of the complexity of package management is due to the interdependence of packages. For example, the popular npm package *webpack-dev-server* (6.6M weekly downloads) declares 33 dependencies, each requiring further packages, for a total of 391 transitive dependencies. These packages span many development teams, each of which may update out of step with one another.This is in line with the general trend of code reuse in software development [32]. Given the bulk of code existing in dependencies, it is infeasible to expect developers to manually vet every package that they integrate into their project.

Package manager front-ends automate the complex and tedious task of fetching, configuring, and updating a package and its transitive dependencies. When a user issues a command like `npm install webpack-dev-server`, the front-end relies on the package's metadata to build a spanning tree of the package dependency graph (or *dependency tree*), and then installs each package. Similarly, the command `npm update` updates the package dependency tree for the current set of packages.The third row of Table 1 shows that there is significant interdependence among packages in the package managers we study.

While package managers save users a significant amount of time, they do not help with the herculean task of vetting imported code; if anything, they complicate it. By design, they tend to obscure the provenance of dependencies, complicating the task for developers to decide whether to assign trust to such dependencies.

Characterization of Package Downloads: Based on the self-reported repository download counts, we classified the popularity of packages across npm and PyPI. Figures 1 and 2 show the distribution and dramatic imbalance of downloads across npm and PyPI. A majority of the packages are downloaded relatively infrequently. The top 1% of packages in both repositories receive essentially all

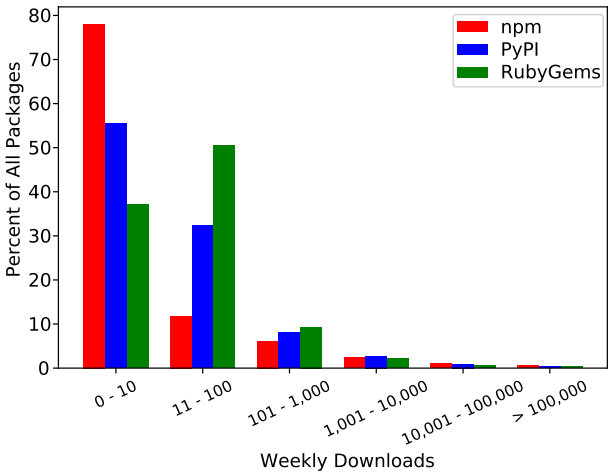

Fig. 1: Percentage of npm, PyPI, and RubyGems packages by weekly downloads.

downloads, as shown in Figure 2. Locating desired packages in this ocean of unpopular packages without assistance is challenging.

2.2 Factors Contributing to Typosquatting

We propose that the following aspects of package repositories contribute to the threat of typosquatting:

- The open-source nature of repositories means that any user can upload a package, and it will be given equal trust to any other package.
- The provenance of a package is opaque to the user, and the interdependence between packages makes their behavior difficult to vet manually.
- The distribution of packages means there are a small number of "juicy" typosquatting targets, and a large number of packages from which a typosquatting attack could be launched.

We now review select cases of historical typosquatting.

2.3 Historical Package Typosquatting

More than one thousand historical typosquatting attacks have been documented [37,42,24]. However, the precise degree to which typosquatting has historically occurred is difficult to capture, due in part to the highly subjective nature of what constitutes typosquatting. In practice, most packages that are flagged by repository maintainers exhibit malicious functionality, and are retroactively deemed typosquatting perpetrators by qualitative manual analysis.

As an example of the complexities of determining typosquatting and its intent, consider the *js-sha3* typosquatting campaign. On October 25th, 2019, 25 packages were simultaneously identified by Microsoft Vulnerability Research and taken down by the npm security team. Upon close inspection, all those packages were determined to have malicious intent, and all package names were close, according to Levenshtein distance, to the victim package *js-sha3*. However, not all

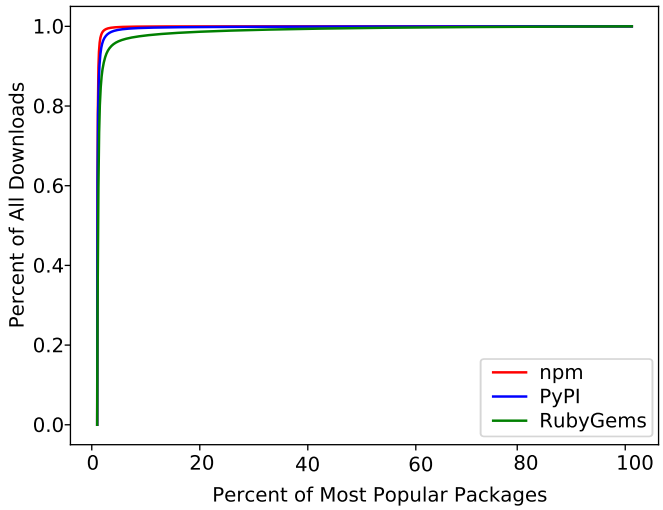

Fig. 2: Cumulative download distribution for npm, PyPI, and RubyGems.

names were likely to confuse the user. For example, *js-sxa3* requires replacing the "h" with an "x". It is unlikely that a developer would misremember *js-sha3* (an implementation of the SHA-3 algorithm) as *js-sxa3*. A typo is equally unlikely on a QWERTY keyboard, given the distance between "h" and "x". As discussed in Section 3.3, we take the stance of only flagging cases where there is strong likelihood that name similarity may confuse the user. While this causes us to ignore some cases (as *js-sxa3* above), it has the advantage to avoid generating an excessive number of warnings that would ultimately stem from flagging pairs of packages with a Levenshtein distance of one. As discussed above, it is unlikely that the packages we ignore could be confused for the correct ones.

One may also be tempted to always attempt to identify malicious intent, regardless of typosquatting. In practice, this is currently impossible to achieve reliably. JavaScript is a particularly difficult target to analyze, and malicious code can be automatically obfuscated to appear syntactically indistinguishable from benign code [18]. Furthermore, the highly dynamic nature of JavaScript means that malicious functionality may not appear until the code is deployed.

Currently, the standard technique for removing typosquatted packages is manual and reactive. Users who discover malicious typosquatting can file a report to the repository maintainers, who will then investigate the claim and remove the package. This approach does little to prevent the installation of malicious packages and fails to protect users from the consequences. Despite these shortcomings, hundreds of package takedowns have been issued that involve package names similar to a popular target. We believe this number to be a lower bound on the total number of typosquatting attempts. The differential of effort favors the attacker, which can automatically generate and upload an arbitrary number of typosquatted variants (e.g., the 25 packages reported by Microsoft above). Many of the reported incidents were actives for months to years before take-down.

2.4 Consequences of Typosquatting

Attacks against end-users: The most subtle typosquatting attack is when an adversarial uploader delivers a malicious payload as part of the dependency code, which is subsequently used as part of a user-facing application. This impacts end-users of the application. Two highly-publicized examples involved a malicious payload that exfiltrated sensitive information such as credit card numbers [17] or cryptocurrency [35]. A stealthy adversary may attempt to obscure the payload by cloning the target package and adding the malicious functionality as a Trojan.

Attacks against developers using a package: An adversary may also target the developer who mistakenly requests the perpetrator package at install time. All three of the repositories analyzed here allow packages to invoke shell scripts— running with the privilege of the user— at install time. Since packages can be installed system-wide, the user may be the administrator, opening a vector for an adversary to take control of the developer's machine. A common choice for malicious package creators is to open a reverse shell on the victim machine [36].

Degradation of functionality: Even when perpetrator does not deploy malicious code, they may still hinder operations by wasting developers' time in diagnosing and remediating package confusion.

Latent vulnerabilities: If a perpetrator package is not detected immediately upon installation, it may remain latent in the victim's codebase for a significant time. For example, developers have typosquatted a target with a payload that is a clone of the current version of the package. While the victim experiences no initial consequences from using the wrong package, the perpetrator may become outdated as it does not receive the same updates as the right package.

An illustrative case is *loadsh*, mentioned in Section 1, which typosquats *lodash*. *loadsh* does not include any malicious functionality - the perpetrator package is an exact snapshot copy of *lodash* version 4.17.11 (*lodash* is currently at version 4.17.15). Nevertheless, the perpetrator still has a negative impact; because the perpetrator package has not been updated, its victims were effectively using an outdated version of *lodash*. We confirmed that *loadsh* was being used unintentionally by emailing the maintainers of packages that used *loadsh*. Three *loadsh*-dependent package maintainers responded to our email, all of whom acknowledged that they had intended to install *lodash* instead. Many of the packages using *loadsh*, including those maintained by our respondents, had been victims for over a year. In the case of this example, the older version contains known prototype pollution vulnerabilities [41], effectively leaving victims of *loadsh* open to attacks that have already been patched in *lodash*.

Misattribution: Even if a perpetrator replicates all of the target functionality, it nevertheless fragments the popularity of the target package. Thus, one minor consequence of typosquatting is that the target will not get as much credit as they would without the perpetrator. Misattribution can be found in packages like *asimplemde* on npm. In addition to typosquatting, this package contains identical functionality to *simplemde*. References attributing credit to the original author are the sole omissions from the duplicate package.

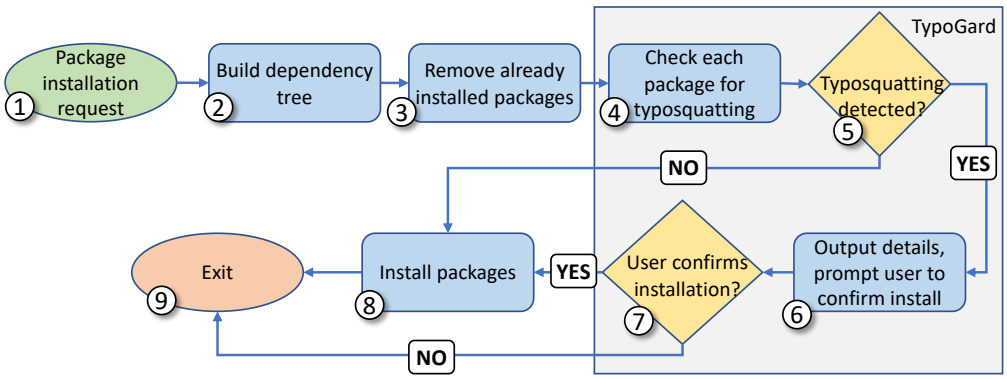

Fig. 3: Modified package installation process with typosquatting protection.

3 Detecting Typosquatting

Motivated by the number of historical instances, the ease of execution, and the severity of the consequences, we created TypoGard, a tool to detect typosquatting in package repositories. At a high level, TypoGard compares a given package name to a list of popular package names. If the given package name matches at least one of the popular packages after a set of allowed transformations, or *signals*, then it is considered to be a typosquatting suspect and the user is alerted.

3.1 TypoGard Workflow

The primary way in which we expect TypoGard to be deployed is as a user-facing utility that integrates with the package manager front-end. Figure 3 depicts the workflow of TypoGard. Algorithm 1 presents the typosquatting detection algorithm (steps 4 through 7 in the figure).

The user initiates the process by triggering a package's installation from the command line, e.g., `npm install loadsh` (**step 1**). The package manager computes the dependency tree of the package (**step 2**), discard from the tree the packages that are already installed (**step 3**), and begins installing the rest. At this point, the workflow triggers TypoGard's logic.

First, TypoGard considers each package queued to be installed (**steps 4-5**, lines 1-3 in Algorithm 1). A package is considered suspicious if its *popularity score* (Section 3.4) is below a tunable threshold T_p, and there exists a popular (popularity $\geq T_p$) package with a similar name (similarity is discussed in Section 3.3). If this is the case, TypoGard flags the package and warns the user (**step 5-6**, lines 4-6). If the user decides to ignore the warning, the package is installed (**step 8**, line 5), otherwise the process is terminated. Note that *AbortInstallation()* in line 6 terminates the process for all queued packages, not just the one which was the object of the warning. In line 7, any package which does not raise suspicion is directly installed without prompting the user.

3.2 TypoGard Batch Analysis

While we anticipate the workflow in Figure 3 to be the most common application of TypoGard, we also envision repository maintainers may want to periodically

apply the same analysis in batch fashion to the *entire package repository*. Our current implementation also supports this. In batch mode, TypoGard receives as input the list of all package names, and returns a list of candidate perpetrators, ranked by decreasing download count. Indeed, the *loadsh* package was identified in this way; TypoGard's batch analysis ranked it as the seventh most popular candidate matching a specific signal discussed in the next subsection.

3.3 Typosquatting Signals

At its core, TypoGard relies on string similarity; however, precisely defining the notion of similarity is challenging in this context. Initially we experimented with Levenshtein distance, a common measure of string similarity. However, we found that this approach is overly simplistic and fails to capture the elaborate typos used in past typosquatting attacks.

After extensively exploring alternative approaches, we devised a typosquatting detection scheme by analyzing and categorizing string transformation patterns that have been used in past typosquatting attacks. We refer to the presence of each of these patterns as a typosquatting *signal*. The insight behind our typosquatting detection scheme is that if a pair of packages exhibits one of these signals (i.e. one package name can be transformed in the other using one of the identified transformations), then one package in the pair is a potential typosquatting perpetrator. The signals are:

Repeated characters: the presence of consecutive duplicate characters in a package name. For example, *reequest* is typosquatting *request*.

Omitted characters: the omission of a single character. For example, *comander* is typosquatting *commander* and *require-port* is typosquatting *requires-port*.

Swapped characters: the transposition of two consecutive characters. For example, *axois* is typosquatting *axios*.

Swapped words: this signal depends on the presence of delimiters in a package name, where a delimiter is a period, hyphen, or underscore. It checks for any other ordering of delimiter-separated tokens in the package repository namespace. For example, *import-mysql* is typosquatting *mysql-import*.

Common typos: character substitutions based on physical locality on a QWERTY keyboard and visual similarity. For example, *requeat* is typosquatting *request*, *1odash* (with the number one) is typosquatting *lodash* (with the letter L), and *uglify.js* is typosquatting *uglify-js*. Users may overlook visually-similar package names during manual analysis, especially in transitive dependencies.

Version numbers: the presence of integers located at the end of package names. For example, *underscore.string-2* is typosquatting *underscore.string*. Note that *underscore.string-2* was previously undiscovered and TypoGard led us to find a latent vulnerability.

3.4 Package Popularity

In order to successfully implement TypoGard, we also necessitate a formal notion of *package popularity*. This requirement stems from a fundamental belief that we posit, which is that only unpopular packages can be typosquatting perpetrators and only popular packages can be typosquatting targets. Popular packages

Algorithm 1 TypoGard typosquatting detection.

Input: List I of packages to be installed
Input: Package graph G
Input: Popularity threshold T_P
for each $p \in I$ **do**
  if Popularity(p) $< T_P$ **then**
    if $\exists p'$ s.t. Popularity(p') $\geq T_P$ **and** Similar(p, p') **then**
      $R \leftarrow$ UserConfirm?(p, p');
      if $R = True$ **then** Install(p);
      else AbortInstallation();
  else Install(p);

are, by our definition, incapable of perpetrating typosquatting attacks. Next, we believe that there exists no incentive for an adversary to typosquat a package which receives an insignificant amount of attention. If a negligible number of users download that package, then an even smaller number of people could potentially misspell the name of that package and fall victim to the attack. By this token, a package which is downloaded thousands, millions, or even tens of millions of times per week, is a far more rewarding target.

The two main possibilities for quantifying package popularity are the number of downloads and the number of dependents. We decided to use the number of downloads because we believe it is a more indicative measure of true package usage. The public number of dependents counts only the number of other packages that directly depend on a given package. Download count, on the other hand, counts the number of users who have downloaded that package either directly or indirectly through some arbitrarily long chain of dependencies.

Popularity based on download count requires the definition of a threshold to distinguish between popular and unpopular packages. This threshold is of crucial importance. An exceedingly low threshold results in many typosquatting packages being considered popular, thus making their detection impossible. Conversely, an exceedingly high threshold may miss packages which are frequently downloaded and are victims of typosquatting. We use a data-driven approach, discussed in Section 4, to determine the threshold.

4 Analysis and Evaluation

In this section, we perform an in-depth analysis of TypoGard's tunable parameter, the popularity threshold, and we evaluate TypoGard's effectiveness in flagging suspicious package installs. Our goal is to answer the following questions:

1. Is it possible to determine an optimal popularity threshold? What is the impact of varying this threshold? (Section 4.2).
2. What is the effectiveness of TypoGard's typosquatting signals in identifying suspicious packages? (Section 4.3).
3. How well does TypoGard generalize to ecosystems different from those for which it was designed? (Section 4.3).
4. Is the latency introduced by TypoGard acceptable? (Section 4.4).

Ecosystem	#packages	Source
npm	259	npm Security Advisories
PyPI	18	Snyk Vulnerability DB
RubyGems	725	ReversingLabs

Table 2: Database of typosquatting perpetrator packages.

4.1 Dataset

For package data, we consider the entire package graphs for npm, PyPI, and RubyGems. Copies of the exact datasets and download counts that were used have been made available for review[4]. A high-level quantitative summary of the repository snapshots is given in Table 1. Our dataset of known typosquatting perpetrator packages is summarized in Table 2 and was obtained from vetted public vulnerability report and—where possible—security advisories from ecosystem maintainers. Since some feeds do not explicitly distinguish typosquatting incidents from the rest, we reviewed such feeds and isolate entries which meet both the following conditions: (i) the word "typosquatting" is present in the advisory; and (ii) the language unambiguously identifies the incident as typosquatting.

4.2 Popularity Threshold

Download counts bear an obvious relationship to the popularity of a given package. Precisely understanding this relationship however requires careful analysis, due to the fact that download counts for a package represent more than the number of people who have installed a package. Packages are regularly downloaded by repository mirrors and bots which download all packages for analysis. Based on estimates made by the creators of npm, a package can be downloaded up to 50 times per day without ever being installed by an actual developer [25].

Therefore, we use 350 weekly downloads as lower bound for package popularity as packages with fewer than this number of downloads may have never been downloaded by an actual user. As seen in Figure 1, a majority of packages across all three repositories receive fewer than 350 weekly downloads. This lower bound removes more than 90% of the packages in each repository from consideration. As an upper bound, we consider packages with more than 100,000 downloads per week to be unquestionably popular; packages above this upper bound make up less than 1% of each repository and account for a vast majority of all downloads. The analysis we describe in this section aims at finding an appropriate threshold to separate popular packages from unpopular ones between these two bounds. We emphasize that, in order to study generalizability, we conducted the analysis on npm and PyPI first, and designed TypoGard exclusively based on these ecosystems. We later incorporated RubyGems in order to determine whether our analysis, and therefore TypoGard, is effective beyond npm and PyPI.

Effect of threshold on number of perpetrators: The first analysis aims to determine how the number of typosquatting targets influences the number of perpetrators. A package is considered to be a typosquatting perpetrator if it, or

[4] *https://www.dropbox.com/sh/wrkz2l3njol0ecw/AAAqbv9hN83Cfdq2CGy6bBjma*

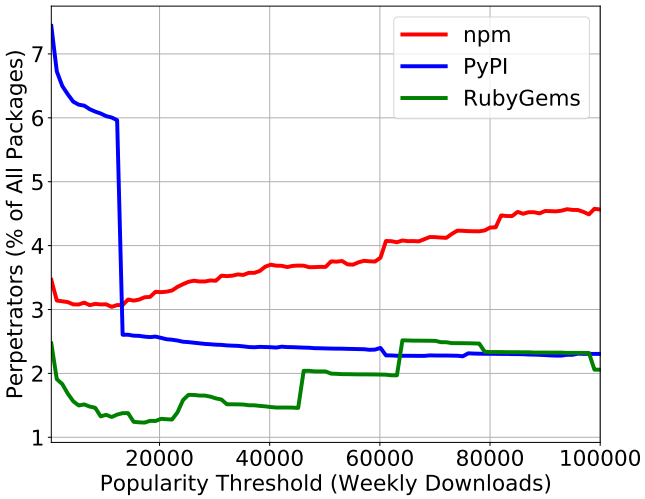

Fig. 4: Popularity threshold vs percent of repository typosquatting.

any package in its dependency tree, fits our definition of typosquatting. Doing this emulates real-world conditions, as users typically would not install a package without installing its dependencies. The results of this analysis is depicted in Figure 4. Interestingly, the curve corresponding to PyPI is fundamentally different from the other two. As the popularity threshold increases, the number of popular packages decreases. With this decrease in typosquatting targets, one would initially expect the number of typosquatting perpetrators to decrease. The trend for PyPI is consistent with this behavior.

However, the curves corresponding to npm and RubyGems see gradual increases. The number of typosquatting perpetrators grows in spite of the fact that the number of targets shrinks. This highlights an interesting phenomenon: a significant amount of package name similarity between popular packages. This phenomenon causes the unintuitive increase in perpetrators seen in Figure 4. As the threshold grows, it surpasses the weekly download count of the less popular package, allowing the package to be considered a perpetrator. Ultimately, this process increases the number of perpetrators as the number of targets decreases. We believe the threshold should be set low enough to avoid flagging reasonably popular packages as perpetrators, in order to reduce the false positive rate, and ultimately, the number of warnings that TypoGard users will experience.

Based on the analysis above, we have chosen to select a popularity threshold of 15,000 weekly downloads. A popularity threshold of 15,000 weekly downloads is the lowest threshold which keeps the number of flagged typosquatting packages reasonably low for both repositories. **We estimate that 3% of the packages on npm and PyPI, and 1.5% of the packages on RubyGems, are potential typosquatting perpetrators at this threshold.**

Effect of threshold on frequency of warnings: We now examine the frequency with which TypoGard will intervene during the package installation process, by flagging a package as potential typosquatting. Keeping this frequency

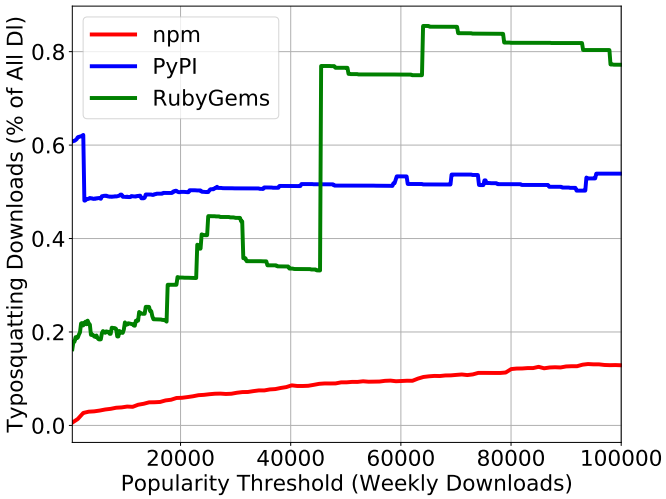

Fig. 5: Popularity threshold vs % of dls containing a typosquatting package.

low is important, because frequently interrupting a developer's workflow risks incurring in the phenomenon of warning fatigue [8]. Also, it is reasonable to expect that the number of packages imported by mistake is a relatively small fraction of the overall number of packages imported by a developer. Therefore, a very high number of warning is likely to consist overwhelming of false positives [3].

This analysis, like the first, is transitive. Results are shown in Figure 5. With the proposed popularity threshold of 15,000 weekly downloads, **the estimated portion of package downloads which will result in a warning from TypoGard is approximately 0.05% for npm, 0.5% for PyPI, and 0.25% for RubyGems**. TypoGard generates on average a warning every 200 to 2000 package installs, which we consider an acceptable burden for the developer.

4.3 Typosquatting Signal Detection Rates

This section aims at estimating the true-positive rate (TPR), or sensitivity, of TypoGard. We do so by measuring what fraction of past typosquatting perpetrators would have been detected by TypoGard. We use the set of attacks detailed in Table 2, which includes more than 1,000 instances of typosquatting.

Meaningfully evaluating the performance of a typosquatting detector is complicated by the fact that typosquatting occurrences are inherently rare events due to the sheer size of the ecosystem. For example, based on known occurrences we can infer that there exist hundreds of perpetrators among more than one million packages on npm. Due to this, most packages flagged by even an extremely accurate detector will be false positives (an instance of the well-known base-rate fallacy [3]). Rather than false positives, we believe that a good typosquatting detector should attempt to minimize the rate at which alerts are generated (as discussed in Section 4.2), while maximizing the true-positive rate (TPR).

First, we measured TypoGard's TPR on our dataset of known perpetrators (Table 2). Each package name was passed to TypoGard to simulate installation,

resulting in successful detection of **88.1%** of historical typosquatting perpetrators. We further note that the observed 11.9% of false negatives is inflated by two main factors: stochastic/exaggerated versions of our signals and combinations of multiple signals. For example, the npm Security Team technically specifies that `ruffer-xor`, `bufner-xor`, and `bwffer-xor` target `buffer-xor`. Furthermore, the malicious Python package `pwd` was accused of typosquatting `pwdhash`. These string modifications are not considered in our set of typosquatting signals, as confusion is unlikely. Extending our signals to capture these instances would essentially revert to a Levenshtein distance form of typosquatting detection, inevitably increasing false positives and presenting end users with frequent alerts. Likewise, checking for packages which utilize multiple signals simultaneously (like `cofee-script`, which targeted `coffeescript`) would exponentially increase the time required to determine if a given package name is typosquatting.

Interestingly, TypoGard confirmed **99.4%** of the typosquatting perpetrators discovered by security researchers on RubyGems, despite the fact that we did not consider any aspect of the RubyGems ecosystem while designing TypoGard. This encourages us to conclude that the underlying algorithm generalizes well beyond npm and PyPI (for which TypoGard was originally designed).

The analysis of the rate at which alerts are generated is likewise encouraging. As presented in Section 4.2, alerts can be expected on about 0.5% of PyPI package installations, 0.05% of npm package installations, and 0.25% of RubyGems package installation, which we believe to be sufficiently low not to disrupt developers' workflow, while providing protection against typosquatting.

4.4 TypoGard Overhead

The goal of our final analysis of TypoGard is to determine the temporal overhead it imposes on the package installation process. To quantify the performance of TypoGard, one thousand npm packages were selected at random, weighted by popularity (to simulate the downloading pattern of an actual user). Once selected, the contents of these packages were locally cached, and installation times for each package were measured using npm's official package manager and a version modified to incorporate TypoGard. The official npm package manager had an average installation time of 2.604 seconds, while TypoGard resulted in an average installation time of 2.669 seconds, meaning TypoGard imposes an **average temporal overhead of about 2.5%**. We believe this result is reasonable and the slowdown incurred by TypoGard is effectively unnoticeable.

Batch mode performance: TypoGard's batch mode (Section 3.2) can analyze the entire npm package set in **11 minutes**. This result suggests that TypoGard could be run frequently (e.g., once per day) allowing quick identification of unknown typosquatting cases.

5 Discussion

5.1 Extensions and Customizations to TypoGard

The goal of TypoGard is to decrease the chances of an incorrect package installation due to confusion. However, it is beyond the scope of this work to model all

of the ways in which a user might confuse package names. For example, confusion may stem from misremembering a name, or hearing it incorrectly. Similarly, the particular keyboard layout used by a developer influences typos that that developer is likely to make when typing in the package name. Collectively, these differences may justify personalizing the typosquatting detection scheme.

TypoGard relies on the concept of popularity. It is possible to define alternative notions of popularity (e.g. using the number of dependent packages). Exploring these is future work, but does not require major modifications.

5.2 Server-Side Protection Mechanisms

Our technique successfully detected typosquatting that was active in popular package repositories for over a year, leading to effective remediation. Consequently, we feel that TypoGard could aid server-side security teams in scanning their entire repository to discovered latent typosquatting instances. As discussed in Section 3, repository maintainers can run TypoGard in batch mode to identify suspicious packages that have already been uploaded. We also consider some additional mechanisms that may help to combat the typosquatting problem.

Preemptive takedown: TypoGard could be used to check every newly uploaded package, effectively disallowing the existence of too-similar package names. This approach is a natural extension to the case-insensitive and delimiter-based naming restrictions currently in place on npm and PyPI [27,28,39]. It further limits potential perpetrators from gaining traction and crossing the popularity threshold, thus achieving legitimacy through the confusion of users.

Variant-insensitive package names: A repository could also map all variations of a package name to the canonical version of the package. This approach means that the perpetrator would be unable to upload their package, since the system would consider the name to be taken. Furthermore, it would address the typo by suggesting the correct target. Some repositories already implement some limited form of this behavior [39]. A potential concern with this approach is that it crowds the set of possible names. We note that npm already incorporates a typo-safe mechanism to allow similar package names, called *scoped packages* [26]. With this mechanism, package names can be declared to exist within a scope, and will not conflict with packages with similar or identical names that exist outside the scope. For example, versions of many popular packages deployed using TypeScript (a typed superset of JavaScript) are available under the @types/ namespace (e.g. @types/node). Scoped packages can be used to alleviate the concern that a repository's names may become too crowded.

5.3 Defensive Typosquatting

One tactic currently used to prevent package typosquatting is to preemptively register confusable variants alongside the canonical package name, so that the variants cannot fall under the control of a typosquatter. We refer to this tactic as *defensive typosquatting*. We observed instances of defensive typosquatting in both npm and PyPI. The placeholder behaviors that we observed are as follows:

Transparent inclusion of target package functionality: One approach is to transparently provide the functionality of the target package to the user within the placeholder package. This can be accomplished by making the legitimate package a dependency of the placeholder. We observed this behavior in the npm package *buynan*, which (defensively) typosquats the legitimate package *bunyan*. One limitation of this defense is that it is indiscernible from a case of a malicious Trojan package; at any point a 3rd-party owner of a placeholder could change the redirect to a malicious payload. Furthermore, a less sophisticated method for transparently including target package functionality is to clone the code of the target. However, if the placeholder fails to stay up-to-date with the package it defends, it can actually expose the user to latent vulnerabilities (e.g., *loadsh*).

User alerts: One possible option is to make the placeholder issue an informative alert with directions to change to the legitimate package. This approach has been extensively used within the PyPI repository [6,5]. In this case, placeholder packages utilize the install hook mechanism of PyPI to issue a message at install time that directs users to the packages they likely had in mind.

Package Deprecation: One mechanism used in practice to alert users is deprecation. This mechanism allows a package maintainer to indicate that the package should no longer be used. When a deprecated package is installed, the user is presented with an alert. One limitation of this technique is that deprecation is used for a variety of purposes, which may lead to confusion in the user.

Defensive typosquatting will continue to have a place as a stopgap mechanism to protect against package name confusion, to deal with context-dependent corner-cases that cannot be detected automatically. Nevertheless, tools like TypoGard can alleviate the limitations of placing placeholder packages.

6 Related Work

Typosquatting Defenses: Tschacher's Bachelor thesis [44] demonstrates a successful controlled typosquatting attack. It also briefly outlines defenses based on forbidding names similar to those of popular packages, but does not implement or evaluate them, and does not consider involving developers in the decision. The creators of npm and PyPI have taken basic countermeasures to combat typosquatting. Both platforms have incorporated restrictions on capitalization and punctuation-based differences [27,39]. User-led defense campaigns exist that aim to create placeholders for potential typosquatting names [5,7]. Limitations of these approaches are discussed in Section 5.

Domain Name Typosquatting: Domain name typosquatting has long been a popular attack vector, allowing cybercriminals to hijack web communications [38] and potentially emails [40] for financial gain. This is accomplished by registering a domain name similar to a popular one. In particularly serious cases, regulations allow ICANN to seize typosquatted domains [1]. Such legal framework does not exist for package typosquatting, and this approach may be difficult to apply due to the fast-evolving nature of software ecosystems. Furthermore, not all instances of package name typosquatting come from attacks.

Software Ecosystem Security: Most past efforts focused on vulnerabilities of package managers themselves [10,2], or potential attack strategies enacted by malicious packages [29]. Both analyses are orthogonal to ours, and none of these works reviewed actual incidents or measured the extent of the problem. Other works more specifically analyze security risks arising from the presence of malicious packages in highly interconnected software ecosystems [21,49]. [49] also identifies typosquatting as one of multiple possible avenues for attack, but it provides no in-depth analysis of the phenomenon, nor describes solutions.

General Characterization of Software Ecosystems: Literature presents many other analyses of software ecosystems. While these works present useful information for understanding these complex objects, they do not focus on typosquatting or other potential security-related issues. Examples include [19,31,47].

Mobile Ecosystems: This work has direct parallels to the work done by Hu et al. [22] in the context of mobile applications. They search for typosquatting applications available on the Google Play store using techniques similar those we used in package repositories.

Another related line of work is the study of mobile application markets such as the Google Play store [45,12,46,11]. These works are concerned with applications used by consumers, rather than packages that are specific to the language ecosystem and used by developers. Characterizations of app markets (and defenses proposed against malicious apps) are orthogonal to our work. The closest work is in the detection of *cloned* applications, which has been done via code similarity metrics [20] or behavior [15]. In contrast, our approach is based on the package metadata and an analysis of the properties of the package repository.

Supply Chain Vulnerabilities: Others have looked at the related problem of *supply chain vulnerabilities*, i.e., vulnerabilities in the open-source applications on which a software package depends [43,9,48,30,23]. These works typically discuss identification or impact of potential upstream vulnerabilities. While an attacker could attempt to introduce such a vulnerability via typosquatting, analyzing this possibility is outside the scope of our work.

7 Conclusion

Typosquatting attacks in package repositories are frequent, and can have serious consequences—but have received little attention. In this paper, we have shown that defending against these attacks is both practical and efficient. By comparing the name in the requested package's dependency tree to a list of probable targets, our proposed solution can protect developers from typosquatting attacks. With an average overhead of 2.5%, a warning-to-install ratio of up to only 0.5%, and third-party confirmation of flagged packages, our solution imposes a negligible burden on developers while protecting package creators and end users alike.

References

1. Senate Report 106-140 - THE ANTICYBERSQUATTING CONSUMER PROTECTION ACT (Aug 1999), https://www.govinfo.gov/content/pkg/CRPT-106srpt140/html/CRPT-106srpt140.htm

2. Anish Athalye, Rumen Hristov, Tran Nguyen, Qui Nguyen: Package Manager Security. Tech. rep., https://pdfs.semanticscholar.org/d398/d240e916079e418b77ebb4b3730d7e959b15.pdf
3. Axelsson, S.: The base-rate fallacy and its implications for the difficulty of intrusion detection. In: Proceedings of the 6th ACM conference on Computer and communications security - CCS '99. pp. 1–7. ACM Press (1999)
4. Baldwin, A.: Malicious package report: destroyer-of-worlds - snyk.io (May 2019), https://snyk.io/vuln/SNYK-JS-DESTROYEROFWORLDS-174777
5. Bengtson, W.: Defensive typosquatting packages created by pypi user wbengtson (Jan 2018), https://pypi.org/user/wbengtson/
6. Bommarito, E., Bommarito, M.: An empirical analysis of the python package index (pypi). arXiv preprint arXiv:1907.11073 (2019)
7. Bullock, M.: Python module: pypi-parker (Oct 2017), https://pypi.org/project/pypi-parker/
8. Böhme, R., Grossklags, J.: The security cost of cheap user interaction. In: Proceedings of the 2011 workshop on New security paradigms workshop - NSPW '11. ACM Press (2011)
9. Cadariu, M., Bouwers, E., Visser, J., van Deursen, A.: Tracking known security vulnerabilities in proprietary software systems. In: SANER (2015)
10. Cappos, J., Samuel, J., Baker, S., Hartman, J.H.: A look in the mirror: attacks on package managers. In: CCS (2008)
11. Chakradeo, S., Reaves, B., Traynor, P., Enck, W.: Mast: Triage for market-scale mobile malware analysis. In: Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks. pp. 13–24. WiSec '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2462096.2462100, http://doi.acm.org/10.1145/2462096.2462100
12. Chatterjee, R., Doerfler, P., Orgad, H., Havron, S., Palmer, J., Freed, D., Levy, K., Dell, N., McCoy, D., Ristenpart, T.: The spyware used in intimate partner violence. In: IEEE Symposium on Security and Privacy. pp. 441–458. IEEE Computer Society (2018)
13. Cimpanu, C.: Twelve malicious python libraries found and removed from pypi (Oct 2018), https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/
14. Claburn, T.: This typosquatting attack on npm went undetected for 2 weeks (Aug 2017), https://www.theregister.co.uk/2017/08/02/typosquatting_npm/
15. Crussell, J., Gibler, C., Chen, H.: Andarwin: Scalable detection of android application clones based on semantics. IEEE Trans. Mob. Comput. **14**(10), 2007–2019 (2015)
16. Denvraver, H.: Malicious packages found to be typo-squatting in python package index (Dec 2019), https://snyk.io/blog/malicious-packages-found-to-be-typo-squatting-in-pypi/
17. Duan, R.: Malicious package report: device-mqtt - snyk.io (Aug 2019), https://snyk.io/vuln/SNYK-JS-DEVICEMQTT-458732
18. Fass, A., Backes, M., Stock, B.: HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In: CCS. ACM Press (2019)
19. German, D.M., Adams, B., Hassan, A.E.: The evolution of the r software ecosystem. In: CSMR (2013)
20. Gonzalez, H., Stakhanova, N., Ghorbani, A.A.: Droidkin: Lightweight detection of android apps similarity. In: SecureComm (1). Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 152, pp. 436–453. Springer (2014)
21. Hejderup, J.: In Dependencies We Trust: How vulnerable are dependencies in software modules? Master's thesis, Delft University of Technology (May 2015)
22. Hu, Y., Wang, H., He, R., Li, L., Tyson, G., Castro, I., Guo, Y., Wu, L., Xu, G.: Mobile app squatting. In: Proceedings of The Web Conference 2020. pp. 1727–1738 (2020)
23. Kula, R.G., Roover, C.D., German, D., Ishio, T., Inoue, K.: Visualizing the Evolution of Systems and Their Library Dependencies. In: IEEE VISSOFT (2014)
24. Lakshmanan, R.: Over 700 malicious typosquatted libraries found on rubygems repository (May 2020), https://thehackernews.com/2020/04/rubygem-typosquatting-malware.html

25. npm Maintainers: The npm blog - numeric precision matters: how npm download counts work (Jul 2014), https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts
26. npm Maintainers: npm-scope — npm documentation (Aug 2015), https://docs.npmjs.com/using-npm/scope.html
27. npm Maintainers: New package moniker rules (Dec 2017), https://blog.npmjs.org/post/168978377570/new-package-moniker-rules
28. npm Maintainers: The npm blog - 'crossenv' malware on the npm registry (Aug 2017), https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry
29. Pfretzschner, B., ben Othmane, L.: Identification of dependency-based attacks on node.js. In: ARES (2017)
30. Plate, H., Ponta, S.E., Sabetta, A.: Impact assessment for vulnerabilities in open-source software libraries. In: ICSME (2015)
31. Raemaekers, S., van Deursen, A., Visser, J.: The maven repository dataset of metrics, changes, and dependencies. In: MSR (2013)
32. Security, C.: Contrast labs: Software libraries represent just seven percent of application vulnerabilities. (Jul 2017), https://www.prnewswire.com/news-releases/contrast-labs-software-libraries-represent-just-seven-percent-of-application\-vulnerabilities-300492907.html
33. npm Security Team: Malicious package report: browserift - snyk.io (Jul 2019), https://snyk.io/vuln/SNYK-JS-BROWSERIFT-455282
34. npm Security Team: Malicious package report: comander - snyk.io (Oct 2019), https://snyk.io/vuln/SNYK-JS-COMANDER-471676
35. npm Security Team: npm security advisory: babel-laoder (Nov 2019), https://www.npmjs.com/advisories/1348
36. npm Security Team: npm security advisory: sj-tw-sec (Nov 2019), https://www.npmjs.com/advisories/1309
37. npm Security Team: npm security advisories (May 2020), https://www.npmjs.com/advisories
38. Spaulding, J., Upadhyaya, S., Mohaisen, A.: The Landscape of Domain Name Typosquatting: Techniques and Countermeasures. In: 2016 11th International Conference on Availability, Reliability and Security (ARES). pp. 284–289 (Aug 2016)
39. Stufft, D.: Pep 503 – simple repository api (Sep 2015), https://www.python.org/dev/peps/pep-0503/#normalized-names
40. Szurdi, J., Christin, N.: Email typosquatting. In: Proceedings of the 2017 Internet Measurement Conference. pp. 419–431. IMC '17, Association for Computing Machinery, London, United Kingdom (Nov 2017)
41. Team, S.S.: Prototype pollution in lodash — snyk (Jul 2019), https://snyk.io/vuln/SNYK-JS-LODASH-450202
42. Team, S.S.: Vulnerability db (May 2020), https://snyk.io/vuln
43. Tellnes, J.: Dependencies: No Software is an Island. Master's thesis, The University of Bergen (Oct 2013)
44. Tschacher, N.P.: Typosquatting in Programming Language Package Managers. Bachelor, University of Hamburg, Hamburg (Mar 2016)
45. Viennot, N., Garcia, E., Nieh, J.: A measurement study of google play. In: ACM SIGMETRICS Performance Evaluation Review. vol. 42, pp. 221–233. ACM (2014)
46. Wermke, D., Huaman, N., Acar, Y., Reaves, B., Traynor, P., Fahl, S.: A large scale investigation of obfuscation use in google play. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 222–235. ACM (2018). https://doi.org/10.1145/3274694.3274726, https://doi.org/10.1145/3274694.3274726
47. Wittern, E., Suter, P., Rajagopalan, S.: A look at the dynamics of the javascript package ecosystem. In: MSR (2016)
48. Younis, A.A., Malaiya, Y.K., Ray, I.: Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In: HASE (2014)
49. Zimmermann, M., Staicu, C.A., Pradel, M.: Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In: USENIX. p. 17 (2019)