Defending Against Typosquatting Attacks In Programming Language-Based Package Repositories

©2020

Matthew Taylor

B.S. Computer Engineering, University of Kansas, 2019

Submitted to the graduate degree program in the Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering.

Committee members

Dr. Drew Davidson, Chair

Dr. Bo Luo

Dr. Alex Bardas

Date defended: May 14, 2020

The Thesis Committee for Matthew Taylor certifies
that this is the approved version of the following thesis :

Defending Against Typosquatting Attacks In Programming Language-Based Package Repositories

---

Dr. Drew Davidson, Chair

Date approved: May 14, 2020

Abstract

Program size and complexity have dramatically increased over time. To reduce their workload, developers began to utilize package managers. These packages managers allow third-party functionality, contained in units called packages, to be quickly imported into a project. Due to their utility, packages have become remarkably popular. The largest package repository, npm, has more than 1.2 million publicly available packages and serves more than 80 billion package downloads per month. In recent years, this popularity has attracted the attention of malicious users. Attackers have the ability to upload packages which contain malware. To increase the number of victims, attackers regularly leverage a tactic called typosquatting, which involves giving the malicious package a name that is very similar to the name of a popular package. Users who make a typo when trying to install the popular package fall victim to the attack and are instead served the malicious payload. The consequences of typosquatting attacks can be catastrophic. Historical typosquatting attacks have exported passwords, stolen cryptocurrency, and opened reverse shells.

This thesis focuses on typosquatting attacks in package repositories. It explores the extent to which typosquatting exists in npm and PyPI (the de facto standard package repositories for Node.js and Python, respectively), proposes a practical defense against typosquatting attacks, and quantifies the efficacy of the proposed defense. The presented solution incurs an acceptable temporal overhead of 2.5% on the standard package installation process and is expected to affect approximately 0.5% of all weekly package downloads. Furthermore, it has been used to discover a particularly high-profile typosquatting perpetrator, which was then reported and has since been deprecated by npm. Typosquatting is an important yet preventable problem. This thesis recommends packages creators to protect their own packages with a technique called defensive typosquatting and repository maintainers to protect all users through augmentations to their package managers or automated monitoring of the package namespace.

Contents

1 Introduction **1**

2 Background **5**

2.1 Terminology 5

2.2 Dependencies 5

2.3 How Package Managers Work 7

2.3.1 Installing Packages 7

2.3.2 Creating Packages 8

2.4 Historical Domain Name Typosquatting 9

2.5 Historical Package Typosquatting 10

2.6 Past Typosquatting Defenses 11

2.6.1 Legal Domain Name Typosquatting Defenses 11

2.6.2 Defensive Domain Name Typosquatting 12

2.6.3 User-led Package Typosquatting Defenses 12

2.6.4 Maintainer-led Package Typosquatting Defenses 13

3 Review of Related Literature **15**

3.1 Domain Name Typosquatting 15

3.2 Package Repository Vulnerabilities and Defenses 15

3.3 General Software Repository Defenses 16

4 Threat Model **18**

4.1 Terminology 18

4.2 Motivation 19

4.3 Capabilities 19

5 Technical Details **23**

5.1 Typosquatting Detection Scheme 23

5.2 Quantifying Popularity 25

5.3 Package Namespace Analysis 27

5.4 Package Manager Integration 29

5.5 Implementation and Infrastructure 32

6 Evaluation **34**

6.1 Results 34

6.1.1 Download Distribution 34

6.1.2 Popularity Threshold 36

6.1.3 Runtime Overhead Analysis 39

6.2 Discussion 39

6.2.1 Undiscovered Typosquatting 40

6.2.2 Limitations 40

6.2.3 Typosquatting Signal Efficacy 41

7 Conclusion **43**

List of Figures

2.1 A labelled portion of the dependency tree for the `express` npm package. 6

2.2 The benign package workflow from the perspective of the package consumer. . . . 8

5.1 Transitive analysis of the repository namespace. 28

5.2 The modified package installation process. 31

5.3 Sample prompts warning users against both direct and indirect typosquatting. . . . 32

6.1 The download distributions of npm and PyPI. 35

6.2 Cumulative download distribution for npm and PyPI. 35

6.3 Percentage of all packages considered typosquatting perpetrators as a function of popularity threshold. 37

6.4 Percentage of all weekly downloads containing a potential typosquatting perpetrator as a function of popularity threshold. 37

List of Tables

2.1 A comparison between npm and PyPI repository and average dependency tree sizes. 6

6.1 Instances where typosquatting targets and perpetrators are both popular. 38

Chapter 1

Introduction

As software capabilities grow, the time and effort required to implement increasingly elaborate projects have both risen. Developers have employed *package managers* to help ease this strain. These package managers allow for third-party source code, which has been bundled into self-contained units called *packages*, to be quickly imported into a project. These packages are traditionally stored in an online database referred to as a *package repository*. To install packages, users interact directly with the package manager. Users request the desired package explicitly by name, and usually as a command line argument. The package manager then queries its corresponding repository, gathering the requested package along with any other packages that the requested package depends on. These *platforms*, which consist of packages, a package manager, and a package repository, are typically built to serve a specific language or framework. Examples of popular package repositories include npm (for JavaScript/Node.js), PyPI (for Python), crates.io (for Rust), and the NuGet Gallery (for Microsoft's .NET Framework). The package managers that correspond to these repositories are npm (the package manager and the repository share the same name), pip, cargo, and NuGet, respectively.

Over time, these platforms have become remarkably popular, and rightly so. Packages from expert programmers can serve as reusable building blocks upon which useful projects can be rapidly created. The current largest package repository, npm, has over 1.2 million publicly available packages and serves more than 80 billion package downloads per month [25]. The next largest repository, PyPI, contains over 220 thousand packages and accrues more than 4 billion monthly package downloads [40].

The benefit of an open code repository like npm is clear. Those who utilize packages created

by others, or *package consumers*, can utilize it to swiftly bring advanced functionality into their own projects. Those who create packages for others to use, or *package creators*, on the other hand, can use the repository to distribute their code. Unfortunately, package repositories can also serve as a medium through which attackers can spread malware. Anybody can upload packages to these repositories. As a result, the provenance of a package is completely opaque repository users must implicitly trust the creators of any and all packages they install. To make matters worse, there are few restrictions on package contents. With these conditions, one can naturally expect adversaries to upload malicious packages. Unsurprisingly, this has been done repeatedly in past years. Repository maintainers could choose to restrict who has the ability to upload packages or the allowed contents of a package, but these solutions would directly violate the principles which originally made package repositories so useful.

Regardless of the effect the payload has on its targets, a common tactic used to spread malicious packages is typosquatting. Typosquatting has existed in other domains. However, despite various defenses which have helped mitigate typosquatting attacks against domain names, little has been done to combat typosquatting attacks against packages. Because of inherent differences between domain names and packages, defenses which protect one against typosquatting do not directly translate to the other.

Typosquatting attacks require a low level of sophistication from an adversary. Attackers can freely upload a malicious package under a name similar to that of a popular package. Then, if other users make a typo when attempting to install the targeted package, they will be served the malicious package and fall victim to the attack (e.g. the packages `raect` and `reeact` could be used to typosquat the popular package `react`). Typos of this style are better described as *miskeys*, or genuine typing mistakes. Other typos exist beyond simple miskeys. Adversaries can also employ *visual confusion*, wherein characters with visual similarity are interchanged. An example of this style is replacing the lowercase letter 'L' with the number one (e.g. `1odash` could target `lodash`). The final kind of typo explored in this thesis is that of *semantic confusion*, wherein components of names are transposed (e.g. `typed-array` could target `array-typed`) or superfluous portions are

added (e.g. `requests.js` could target `requests`).

Typosquatting attacks against packages rely on mistakes which can be made very easily. Interestingly, they can even target those who make no typos whatsoever. Despite the ease with which the attack can be orchestrated, the consequences can be devastating. Historical malicious packages have exported private information like passwords and credit card numbers, stolen cryptocurrency, and opened reverse shells, giving the attacker access to the victim's computer [33, 34, 35]. Typically, these malicious packages are manually discovered by either users or repository maintainers, then investigated, and finally dealt with. Some automated defenses against malicious packages do exist. However, these defenses may be circumvented through obfuscation or storing the payload remotely, hiding the package's true functionality. Even with the currently employed defenses, typosquatting attacks continue, and the packages can remain available for up to months [10]. Therefore, stronger defenses against adversaries that target package repositories are warranted, and an automated defense against typosquatting specifically could be remarkably advantageous.

This thesis focuses on exploring the extent to which typosquatting exists in popular language repositories and developing a defense against these attacks. Two repositories, namely npm and PyPI, are used as case studies due to their popularity and history of typosquatting attacks. After uncovering exactly how widespread this issue is and developing a proactive defense against these attacks, the proposed solution's practicality and efficacy is quantitatively measured. The analysis performed has revealed that, despite past attempts to resolve the issue, typosquatting still exists in popular package repositories. Fortunately, automated defenses are able to help mitigate the problem in an efficient manner. The defense proposed in this thesis is an extension to the package manager front-end that warns users prior to installing packages which could be typosquatting. Users are given an opportunity to investigate the claim, decide whether or not the requested package was a mistake, and if desired, abort the installation before any harm can occur. Temporal analysis of this implementation has shown that the analysis of package names incurs a 2.5% overhead upon the typical package installation process. Furthermore, transitive analysis of the package repository namespace shows that, with reasonable parameters, this solution effects upwards

of 0.5% of all package downloads, depending on the repository. The implementation presented here was even used to discover a high-profile typosquatting perpetrator with thousands of weekly downloads. The perpetrator was reported to npm's internal security team, resulting in the package ultimately being seized by repository owners. This work shows the efficacy of programmatic defenses against typosquatting attacks which target package repositories. It shows that automatically detecting potentially malicious packages through lexical criteria could realistically prevent typosquatting attacks before they are given the opportunity to affect developers.

Chapter 2

Background

2.1 Terminology

Package managers were created to combat the ever-growing complexity associated with software development. They are programs which automate the installation, maintenance, and removal of self-contained units of source code called *packages* (the exact term differs slightly between platforms, but throughout this thesis they are universally referred to as packages). These packages can range in functionality, from time-saving tools to complete frameworks. Regardless of functionality, they are primarily created with the intention of providing others with extensible tools that can ease the strain of development. Packages are stored online in a public database referred to as a *package repository*, or sometimes a *package registry*. Package managers serve as the intermediary between users and package repositories. *Package creators*, or *package developers*, use package managers to upload their code to the repository. Conversely, *package consumers* use package managers to install packages from the repository.

2.2 Dependencies

Packages are allowed to rely on other packages in order to function. This relationship is known as a *dependency*. When a package consumer requests a package, the package manager begins by creating a *dependency tree*, which is a hierarchical data structure showing every package upon which a given package depends. A dependency tree obviously contains *direct dependencies*, or packages which are explicitly depended upon. Dependency trees also show any and all *transitive dependen-*

cies. A transitive dependency, or *indirect dependency* is simply a dependency of a dependency. Transitive dependencies are usually never explicitly requested, or even seen, by the user. The total number of transitive dependencies can be immense. Explicitly requesting only a single package can often install dozens, or even hundreds, of distinct packages, which are often created by at least that many developers. Figure 2.1 shows an illustration of a dependency tree and Table 2.1 contains high-level information regarding the average version-independent dependency tree size relation to the total number of packages on both npm and PyPI. Interestingly, it shows that during the average npm package installation, just over 57 total packages are downloaded. Package consumers are inherently required to trust the creators of all packages in the dependency tree.

	npm	PyPI
Total Available Packages	1,262,822	229,160
Average Dependency Tree Size	57.27	4.58

Table 2.1: A comparison between npm and PyPI repository and average dependency tree sizes.

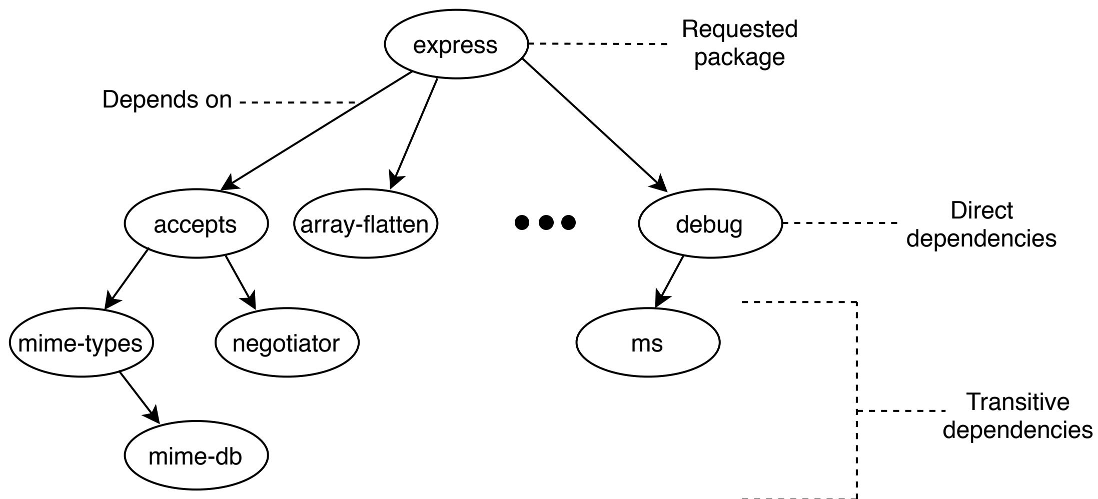

Figure 2.1: A labelled portion of the dependency tree for the `express` npm package.

2.3 How Package Managers Work

Package repository users primarily use package managers to install packages or upload packages that they have created. The following sections describe these processes in detail.

2.3.1 Installing Packages

Packages are typically requested explicitly by name through a package manager's command-line interface. Once the request is received, a query is sent to the appropriate package repository. This query first checks for the existence of the requested package and, depending on the result of the first check, also recursively finds all transitive dependencies. Packages installed without their required dependencies are of little use, so the package manager fetches these as well. The set of all dependencies, both direct and indirect, is stored in a dependency tree.

After the dependency tree has been constructed, package managers check the user's system for any required packages that are already installed. Should any of the required packages already exist on the user's system, they are removed from the eventual list of packages to be installed. Next, all missing packages are downloaded from the package repository and installed. Some packages have more elaborate installations, like those which add to a system's environment variables. These tasks are normally accomplished through the use of *installation scripts*, which allow arbitrary commands of the package creator's choosing to be executed automatically during installation. Immediately after the package has been downloaded, any installation scripts it may have are executed. Once this process has been performed for all packages in the dependency tree, the installation process comes to an end. The package consumer is then able to utilize the requested package's functionality. A high-level illustration of the entire benign package workflow, include the installation process, is shown in Figure 2.2.

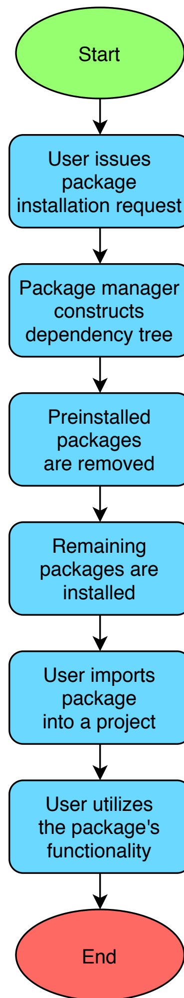

Figure 2.2: The benign package workflow from the perspective of the package consumer.

2.3.2 Creating Packages

Few restrictions are placed on package uploads. The exact rules that govern what is and is not allowed depends on the package repository. However, there are some universal similarities. Specifically, package names must follow strict naming guidelines. These guidelines are, for the most part, limited to containing only allowed characters, having a length within a certain range, and being unique. As long as the desired package name is not already in use, and follows the other naming restrictions, the package can be uploaded. Restrictions regarding the name of the package are typically the only restrictions that package creators will encounter during the package creation process. There are generally no limitations on the contents of a package. Users are able to upload

arbitrary files, including binaries. Lastly, there is no financial cost associated with uploading packages. Together, these conditions provide an ideal foundation upon which malicious entities can orchestrate typosquatting attacks.

2.4 Historical Domain Name Typosquatting

Typosquatting attacks are far from novel. Typosquatting gained notoriety as an attack against domain names [47]. Adversaries would intentionally register domains with names similar to popular domains [47, 50]. The differences between the two domain names were usually simple typos. Common typos that could be leveraged to orchestrate these attacks included transposition of consecutive characters, repeated characters, and omitted characters [50]. For example, `www.gogle.com`, `www.googel.com`, and `www.gooogle.com` could all reasonably be used to typosquat `www.google.com`. Other techniques modifications which applied specifically to domain names were also used, such as interchanging top-level domains. For instance, replacing a *.net* domain with *.org*. While these replacements do not clearly align with the aforementioned typos, they could still be effective against those who misremember the correct domain. Due to their efficacy and impact, typosquatting attacks against domain names gained significant attention. Some researchers estimate the total number of typosquatting domain names to be around 21.2 million (or 20% of all `.com` domains) [43].

Attackers would register these domains in hopes that users would make a mistake when trying to visit the targeted site. The logic behind this attack is sound. The more popular the target, the larger the chance that a careless typist would mistakenly visit the typosquatting domain. Other domain name typosquatting strategies opted for a different attack vector, in which the domain name never had to be typed at all. These attackers would register domains which were visually similar to to their targets. Replacing the letter 'm' with a consecutive 'r' and 'n', or a lowercase letter 'l' with the number '1' are excellent examples of this strategy.

The motivation for registering typosquatting domains was typically financial [23, 36, 13]. Once registered, typosquatting domains can be used in multiple ways. Domains which contain common

typos (that users would realistically make) could be used to host ads. Techniques such as this directly generate revenue for the party which registered the typosquatting domain. Other typosquatting techniques could promote business. Companies could register domains which typosquat their competition, redirecting users away from their intended site and to a competitor, effectively siphoning business [20].

Typosquatting domains which do not contain typos that users would realistically make, but rather share visual similarity with their targets, can also be leveraged for profit. An email sent from a domain which appears to be one's a financial institution at a glance could be used to deceive victims into disclosing credentials, which attackers could then use for theft. Attacks of this nature are known as *phishing* and are still a prominent security issue [17].

2.5 Historical Package Typosquatting

Historical typosquatting packages are plentiful and have affected thousands of users [29, 24]. Hundreds of typosquatting attacks have been orchestrated against package managers like npm and PyPI [37, 38]. These packages can be available for up to months before they are removed [10]. Some of the first notable attacks include those performed by an npm user named hacktask. In 2017, hacktask uploaded a total of 38 typosquatting perpetrators, targeting a wide variety of some of npm's most downloaded packages. All of these packages contained malware which uploaded the victim's environment variables to a remote server upon installation. The most effective of these typosquatting perpetrators was `crossenv`, which targeted `cross-env`. When it was discovered and removed, the malicious `crossenv` had been downloaded nearly 700 times [29].

The payloads that typosquatting packages carry are largely unconstrained, as a consequence of the freedom that platforms they leverage tend to provide. Most of the historical typosquatting attacks have had similar functionality. Malicious packages have been known to export sensitive information, like `crossenv`, though the type of information is not always limited to environment variables. Packages in the past have gone after credit card numbers and passwords [34] or cryptocurrency [33]. One of the most common payloads found in malicious packages is the reverse

shell. Attackers have used platforms like npm and PyPI to open reverse shells on the machines of their victims [35]. Once open, these reverse shells can lead to a number of opportunities through which the attacker can perform much more devastating attacks.

2.6 Past Typosquatting Defenses

The foundation of typosquatting defenses for both domain names and packages is string similarity [50, 28]. However, string similarity in and of itself is not the definitive indication of typosquatting. Levenshtein (or edit) distance is a common measure of string similarity [21, 19]. It is a measure of how many characters need to be added, removed, or replaced to transform one string into another. Most typosquatting examples have a low edit distance when compared to their target [48]. However, searching for typosquatting perpetrators by flagging strings with an edit distance of one is imperfect, often yielding high false positive and false negative rates [44]. Defenses which utilize an edit distance of one will incorrectly flag pairs like "laughter" and "daughter" (which have an edit distance of 1), while failing to detect stealthy typos which rely on human context and intuition.

2.6.1 Legal Domain Name Typosquatting Defenses

Typosquatting domain names eventually gained a significant amount of attention. Those which targeted domain names belonging to litigious companies were often met with legal ramifications. Typosquatting domains could redirect users to a competitors website, siphoning business away from the target. Because of this, legislation has been passed to help protect companies from typosquatting. The Internet Corporation for Assigned Names and Numbers (ICANN) introduced the Uniform Domain-Name Dispute-Resolution Policy (UDRP) to combat typosquatting [16]. This established the legal grounds upon which companies could protect themselves against typosquatting perpetrators who were intentionally registering confusingly similar domain names with the intention of trademark infringement. Many companies leveraged the UDRP to prevent typosquatting. For instance, the popular toy company Lego has notably been engaged in more than 300 UDRP

proceedings, accruing legal fees of nearly $500,000 [15]. Once in the possession of the target, companies could configure typosquatting domains to redirect traffic to the original site, thereby negating any potential harm that could arise from a simple spelling mistake. This solution comes with the major drawback of time. UDRP claims are not resolved instantly and in the meanwhile, have no protection against the alleged typosquatting attack.

2.6.2 Defensive Domain Name Typosquatting

More proactive domain owners might anticipate potential typosquatting opportunities. In these cases, common misspellings or visually similar domains could be preemptively registered before they can be used for harm. This behavior is known as *defensive typosquatting*. Domains which utilize defensive typosquatting, like those which can be won through legal proceedings, could then be configured to redirect to the original website or parked to prevent any misunderstanding by the end-user. A notable example of this strategy is one performed by Google. To prevent any misuse, the domain names `www.gogle.com` and `www.gooogle.com` now redirect users to the popular search engine with a similar name [12].

2.6.3 User-led Package Typosquatting Defenses

The legislation protecting domain names does not extend to package names. However, defenses against package typosquatting still exist. Package creators can employ a technique similar to the defensive domain name typosquatting described in the previous section. Because packages can be created for free, any number of misspellings of a package name can be registered. These defensive typosquatting packages are then not only out of reach for any adversaries, but they can also point users towards their desired package. Depending on the capabilities of the platform the package is created for, packages can simply be parked, throw an error during installation which alerts users to their mistake, or even redirect users to the intended package. Security specialist and PyPI user William Bengtson has taken the initiative to create over one thousand defensive typosquatting packages to protect some of the repositories most popular packages [3, 4]. When installed, his

packages alert the user that a typo was made, then suggest the user install the targeted package. A limitation of this defensive typosquatting being done by a third-party user is that it relies on trusting third parties, who could behave inconsistently with the desires of the first-party developer. In addition to these community-based defenses, package repository maintainers have also used implemented a series of defenses which aim to mitigate typosquatting.

2.6.4 Maintainer-led Package Typosquatting Defenses

Not long after the `crossenv` incident, the maintainers of npm introduced a stricter set of package naming rules [28]. The new rules prohibit new packages from containing capital letters. However, for the sake of backwards-compatibility, existing packages that differed only in capitalization were allowed to remain on the repository. For example, the distinct, and legitimate, packages `jsonstream` and `JSONStream` still exist to this day. Also, they prevent new packages which differ from popular packages only by punctuation (hyphens, underscores, and periods) from being uploaded. In the announcement revealing these new rules, the npm maintainers list examples of packages which will not be allowed on the repository due to their similarity to `react-native`. Their examples include `reactnative` and `react_native`, both of which currently exist on the repository and consistently receive a small number of weekly downloads. Other listed packages like `react.native` do not exist and cannot be created. While this solution does technically reduce the number of potential typosquatting perpetrators, it only covers a subset of possible edits that can be made to a package name. Variations on the listed examples which utilize repeated, omitted, or transposed characters are still allowed to be uploaded. Furthermore, the packages to which this protection applies is not given, and many packages which differ only by punctuation remain on the repository.

In stark contrast to npm, PyPI has a robust defense against punctuation-based typosquatting attacks built into its naming conventions. The names of all packages on PyPI are normalized [41]. This means all periods and underscores are replaced with hyphens when the package is both created and installed. This technique serves a dual purpose of preventing specific typosquatting

packages from being uploaded in the first place and simplifying the package installation process. However, this defense only prevents a portion of commonly used typosquatting methods. It does nothing to protect against one of the latest typosquatting perpetrators named `jeilyfish` (which clearly targeted the popular package `jellyfish`), discovered in December of 2019 [10].

Despite these attempts to thwart typosquatting packages from being uploaded (from both repository users and maintainers), perpetrators continue to find a way. The current restrictions imposed upon package creators do help mitigate the problem, though more can be done. More sophisticated typosquatting detection algorithms can be used to help identify perpetrators before they have a chance to permeate the repository's user base. An automated defense would also undoubtedly surpass the current manual solution which relies on reports from repository users.

Chapter 3

Review of Related Literature

3.1 Domain Name Typosquatting

A considerable amount of attention has been given to research regarding domain name typosquatting. Researchers naturally performed analysis of domain names to uncover the extent to which typosquatting exists. A submission to the 23rd USENIX Security Symposium estimated that 21.2 million `.com` domains (or about 20% of all `.com` domains) were typosquatting perpetrators [43]. Another group quantified this widespread issue by stating 95% of the Alexa Top 500 Websites were being actively targeted by typosquatting perpetrators [1]. These papers, and many others [22, 8, 2, 18], use typosquatting detection schemes similar to, or derived directly from, those proposed by Wang et al., who formalized five so-called typo-generation models that were so effective, they led to the discovery and removal of thousands of typosquatting domains [50].

3.2 Package Repository Vulnerabilities and Defenses

As typosquatting attacks began to affect package repositories, they too became the focus of research. Multiple have been written that discuss the issues of package typosquatting. However, in most cases, typosquatting is simply cited as an issue when assessing inherent vulnerabilities present in package managers/repositories and few researchers offer solutions to help mitigate the problem [52, 45, 5]. Other sources have taken much more creative approaches. In his Bachelor's thesis, Tschacher created a typosquatting package of his own [46]. This experimental package contained code which reported back to a central server with some basic information on where the

package was installed so Tschacher could track its spread. Surprisingly, his package had received more than 45,000 installation requests from more than 17,000 unique domains. When analyzed, his results showed that had even been installed on machines associated with `.gov` and `.mil` domains. Tschacher's work demonstrated the efficacy of typosquatting attacks and shed light on the possible consequences they could have, ultimately highlighting the need for stronger defenses against these attacks, which he did not implement. Past defenses which have been proposed to specifically fight typosquatting are usually limited to the utilization of features currently offered by platforms like npm and PyPI, such as name scope [27] and defensive typosquatting [39, 42, 3, 4].

Tools which attempt to detect general malware in package repositories have been implemented in both academia and industry. While some of these defenses do not specifically check for typosquatting, the packages they do find occasionally tend to employ the deceptive naming strategy. From academia, Duan et al. were able to develop a robust system which performs static and dynamic source code analysis to effectively detect malicious packages [11]. However, this approach takes place outside of the package installation process; passively protecting users until vulnerabilities can be found, investigated, and removed. In industry, repository maintainers have made it abundantly clear that they take the security of their platforms seriously. The maintainers of npm have released several blog posts describing their experimental new Security Insights API, which promises to provide security-related information on package capabilities and resource utilization [30, 31, 32].

3.3 General Software Repository Defenses

This thesis shares a common goal with a line of work done to protect general software repositories like the Google Play Store [49, 7, 51, 6]. While these works are concerned primarily with full-fledged mobile applications rather than discrete units of source code, they do share similarities to the work discussed here, despite being otherwise orthogonal. The work in this field which is most closely related to the work presented in this thesis deals with the detection of so-called *cloned* applications. A popular application is cloned when it is reuploaded under a different name. Systems

which detect cloned applications typically do so through some notion of binary [14] or functional [9] similarity. The concept of cloned software extends to package typosquatting because of the way that adversaries are allowed to reupload the source code of their target to mask malicious payloads. However, in contrast to the code similarity analysis performed on Google Play, this work has no interest in the theft of intellectual property, because of the open-source nature of package repositories. Advanced functionality/source code similarity checks in the context of package repositories would only provide evidence which may bolster typosquatting claims. Two packages with similar code but sufficiently different names fall outside the realm of typosquatting.

Chapter 4

Threat Model

4.1 Terminology

Throughout this thesis, for the purpose of consistency and clarity, the following terms are used to describe various parts of a standard typosquatting attack in the context of package repositories. Here, the term *attacker* denotes the individual who intentionally uploads the malicious package with a name that closely resembles that of a popular package. The *victim* is the individual who installs the malicious package. The terms *perpetrator* and *target* refer not to people, but to packages, specifically the malicious package and the legitimate (benign) package, respectively.

The definition of typosquatting encompasses more than lexical similarity. For a package to be typosquatting, it must intentionally attempt to deceive users into mistaking it for its target. If a package has a name very similar to a popular package, it is not automatically typosquatting. Packages exist which are intentionally given names that could reasonably be used for typosquatting, but they either have no functionality or direct the user's attention to the targeted package [3]. These packages are performing *defensive typosquatting* and are not typosquatting perpetrators since they make the conscious effort to alert the users of their mistake. Similarly, if a package coincidentally has a name similar to that of a popular package, and it has radically and blatantly different yet benign functionality, it is not considered a typosquatting perpetrator. Because this hypothetical package makes no attempt to intentionally deceive users, any alerts that flag it as a typosquatting perpetrator are false positives.

4.2 Motivation

Malicious packages are created for a variety of reasons. Like with domain name typosquatting, this motivation can be financial. Packages which aim to collect and export sensitive information from the victim's machine could very well be used for the financial gain of the attacker. Multiple historical typosquatting attacks have had financial motivations. A number of packages were created which stole cryptocurrency [33] or stole credit card numbers and passwords [34]. The motivation of many other packages is unknown. Packages whose payloads were hidden or those that opened reverse shells had unknown effects. It can be assumed that the creators of these packages likely had financial or other generally antagonistic motivations.

4.3 Capabilities

The exact capabilities of a package typosquatting attack depends on the repository being used and do not surpass the capabilities afforded to benign package developers. Past typosquatting attacks do not typically harness elaborate exploits. Historical payloads utilize the existing features present in modern programming languages. They simply exploit the inherent characteristics of package repositories and package managers, along with the blind trust that is essentially required of package consumers.

Malicious package creators are given a significant amount of freedom and power. For instance, in contrast to domain names, packages can be uploaded for free. Without a monetary restriction, package typosquatting attacks could theoretically be performed by anyone with Internet access. The fees associated with registering domain names introduce risk, essentially creating an investment. This risk, though admittedly minimal, may discourage attackers and prevent some number of attacks from taking place. Since no such risk exists for package typosquatting, very little stands in the way of an individual carrying out any number of attacks.

Not only can packages be uploaded for free, they can also have arbitrary functionality. This choice by repository maintainers allows legitimate packages to provide a greater level of utility.

However, it also permits attackers to distribute more devastating payloads. The attacker can execute arbitrary commands during either installation or runtime. Clearly, the contents of the package are executed at runtime. For a language like Python or an environment like Node.js, system calls can be executed with the privileges of the user executing the program. These platforms allow attackers to do many things. Files can be deleted, sensitive information can be stolen, and perhaps worst of all, reverse shells can be opened, giving attackers a substantial amount of access to the victim's machine, ultimately enabling a multitude of negative effects [33, 34, 35]. Other packages can deploy these payloads long before runtime, needing only a modicum of interaction on the victim's part. Installation scripts are a common feature for most package managers. Usually, they perform specialized operations required by the package which are not a part of the standard package installation process. Legitimate installation script uses are generally limited to initialization-related functions. Examples include building programs from its source code or manipulating environment variables to add to a user's PATH. Attackers can leverage install scripts to deploy payloads the instant a victim installs the typosquatting perpetrator. While this attack vector is direct and can be effective, other tactics may prove more rewarding.

Attacks could leverage the transparent nature of package repositories in their favor. As previously mentioned, these repositories are inherently open source, since source code is usually the key deliverable. Those performing typosquatting attacks could easily download the content of the package they're targeting, then reupload the code under the typosquatting name. At first, it may seem that this has no benefit to the attacker. However, this is far from the truth. Doing this gives the attacker some limited form of access to their victims, in that they have the ability to deploy code through package updates. Meanwhile, the victim installed the typosquatting package would possess the expected functionality, helping to reduce any suspicion. Then, at a time of the attacker's choosing, the typosquatting package can be updated to contain the payload. This payload can supplement the original package's functionality, making detection slightly more challenging. When any users who mistakenly installed the typosquatting package update this package, they will promptly be served the malicious payload. Attackers could opt to wait until obtains some degree of

popularity in order to affect the most victims. There are additional methods of disguising functionality that the attacker may also employ. In order to circumvent the transparent nature of package repositories, the attacker could store the payload remotely. Then, at either during installation or at runtime, the malicious package could be configured to fetch and execute the remote payload.

Typosquatting target candidates can be found quickly. Weekly download counts along with the number of dependents are official and readily available for packages on npm. PyPI, on the other hand, has official hidden download counts and dependents, though this information is still accessible through package metadata and third-party APIs. Attackers could sort lists of packages by popularity, whether that means weekly downloads or number of dependents, and target those to theoretically have the largest impact. Once a target has been identified, attackers can upload malicious packages under virtually any name. The selected typos could be simple variations, such as swapping two consecutive characters or duplicating a single character. Other more complex strategies also exist. Should a package name consist of two distinct parts, delimited by some form of punctuation, the attacker could swap these two components. For instance, the package `oauthlib-requests` could be used to typosquat the popular Python package named `requests-oauthlib`. These two packages could reasonably be mistaken for one another, which opens an attack vector for advanced typosquatting perpetrators.

To further decrease the likelihood of being detected, attackers could target dependency trees. At installation time, package managers routinely gather all direct and indirect dependencies. Typosquatters who decide to target dependency trees could affect those who never intend to explicitly install the package being targeted. Typosquatting attacks in package repositories an affect users transitively. Package creators who make a typo when installing a dependency for their package could affect all users of the package they produce, along with any users of package that depends on their package. This cycle can continue, making the chain of dependencies arbitrarily long, affecting users along any point in the chain. This style of typosquatting presents a new opportunity for attackers. Analogous to the relationship between domain name typosquatting and phishing, package typosquatting in this case could utilize visual similarity in addition to common typos.

Because the package names in the dependency tree are not explicitly requested, or even seen, by package consumers, the attack surface is much broader. Characters with visual similarity may be interchanged to deceive users. For instance, the package `1odash` could reasonably be mistaken for `lodash` when seen in a list of dependencies. Using the concept of dependencies to help aid typosquatting attacks has the added ability of affecting those who never make a typo, but install a package created by someone who did.

Chapter 5

Technical Details

This chapter outlines the series of experiments that were performed in order to quantify the efficacy and performance of the proposed solution. Throughout this chapter, low-level details pertaining to the experiments performed are also given.

5.1 Typosquatting Detection Scheme

The core goal of this work is to programmatically detect typosquatting. Given a package name and a set of packages considered to be popular, the defense must be able to determine whether or not the given package is a typosquatting perpetrator.

The foundation of any defense protecting users against typosquatting attacks is its typosquatting detection scheme. By definition, a typosquatting perpetrator's name must have lexical similarity to the name of a popular package. The selected method of string similarity which compares typosquatting targets to perpetrators directly determines the efficacy of the solution. Simple Levenshtein distance (colloquially referred to as edit distance) is commonly used to measure string similarity, though it is far too aggressive when detecting typosquatting, especially with shorter strings. This flaw stems from its indiscriminate handling of characters. When using standard edit distance, any character can be inserted, removed, or replaced with any other character. Meanwhile, realistic typos (specifically miskeys) are typically limited to a small list of candidates for each character. These candidates are typically determined by physical locality on the keyboard being used. Because most typosquatting attacks are character-discriminant, past domain name typosquatting detection schemes were essentially a restricted form of edit distance. Typosquatting

detection that relies on pure edit distance would inevitably lead to a dramatic increase in false positives. More advanced detection schemes outline a set of characteristics, or *signals*, found in the names of typosquatting perpetrators. These signals can be thought of typo categories and they include single-character omission, adjacent-character transposition, substitution based on keyboard locality, and character duplication [50]. In this thesis, these signals are adapted and extended to more effectively apply to package names. The six signals used to detect package typosquatting are listed below with accompanying definitions and examples.

1. Repeated Characters – The presence of consecutive repeated characters. The potential typo being made here is one where a key on the keyboard had been struck twice or held slightly too long. This signal is limited to one excess character. Examples of this signal include `reeact`, which targets `react`, and `expresss`, which targets `express`.

2. Omitted Characters – The omission of a single character. This signal checks for typos in which a character is simply not entered, as if the corresponding key hadn't been pressed hard enough. Examples of this signal include `event-strem`, which targets `event-stream`, and `reques`, which targets `request`.

3. Swapped Characters – The transposition of two adjacent characters. This typo could occur when typing the package name quickly, hitting keys out of order. Examples of this signal include `erquest`, which targets `request`, and `loadsh`, which targets `lodash`.

4. Swapped Words – The reordering of delimited substrings. In the context of package names, delimiters include periods, hyphens, and underscores. This signal checks for alternate permutations of substrings which are delimited by these characters. Additionally, it checks for possible delimiter substitutions and omissions. For example, `stream-event`, `event.stream`, and `eventstream` all target `event-stream`.

5. Common Typos – The substitution of characters based on physical locality and visual similarity. This signal checks for two distinct but similar forms of typosquatting, miskeys and

visual confusion. The first results from a typo made during package installation, where one character is replaced by another which is physically close to it on a standard QWERTY keyboard. For example, `cpmmander` targets `commander`. The other form of typosquatting is meant to gives packages a passing resemblance to their targets. By replacing certain characters with others which are visually similar. For example, `1odash` targets `lodash`.

6. Version Numbers – The inclusion of version numbers after the package name. This signal checks for typosquatting perpetrators which attempt to target a specific version of a package. These perpetrators could affect victims who request a specific version of a package using incorrect syntax, or those who mistake the perpetrator for the target in a set of transitive dependencies. For example, `debug-4.1.0` targets `debug` version 4.1.0.

5.2 Quantifying Popularity

The targets of typosquatting attacks are typically popular, for obvious reasons. When a package is used more frequently, more opportunities exist in which a typo can be made during the installation process. A typosquatting attack which targets a popular package can, in theory, affect many more victims than one that targets a particularly unpopular package. There is almost no incentive in typosquatting a package which receives no attention. If a package is used infrequently, the overall likelihood that a mistake is made during its installation, thus falling victim to the typosquatting attack, is even smaller.

Typosquatting perpetrators, on the other hand, are inherently much less popular than their targets. The utility they provide is generally less than or equal to the utility offered by their targets. In most cases, once a typosquatting perpetrator gains a significant amount of popularity, one of its victims will catch on and report the incident through the proper channels. While this approach ultimately results in the removal of the malicious package, it does so at the cost of infecting users. A proper solution to the problem of package typosquatting should help users identify perpetrators before the payload has a chance to propagate.

To programmatically detect typosquatting, popularity must be quantified and distinction between popular and unpopular packages must be established. Allegations of an unpopular package typosquatting an unpopular target hold little merit. Likewise, if the name of two extremely popular packages are similar, users should not expect to see typosquatting warnings when downloading one of the two. Since both packages in this example are considered to be popular, both are equally qualified to be intentionally installed. A package is most likely a typosquatting perpetrator if it is unpopular and its name differs from the name of a popular package through one of the typo categories, or signals, described in Section 5.1.

Two main candidates for quantifying package popularity exist: download count and dependent count. The former, depending on the repository, is a rough indication of the number of users who request the package either directly or indirectly. The latter is the number of other package on the repository which directly depend on the package in question. Both of these metrics are publicly available either officially or through third-party tools. On npm, for example, the page for each package displays the number of weekly downloads the package receives alongside the number of dependents. PyPI does not make this information readily available, however it can be found in package metadata and officially endorsed data sets.

Weekly downloads and dependent counts both clearly represent the popularity of a package, but weekly downloads give a more accurate approximation of true use. Dependents are a subset of downloads, since the creator of the dependent was required to download the package in order to include it in their own package. Furthermore, dependents are only packages which have also been uploaded to the repository. Some packages exist as command-line tools, which are installed with no intention of being included in other packages. The dependent count for a package of this nature could reasonably be zero, despite potentially being downloaded thousands of times per week. In any dependent-based definition of package popularity, a package like this could have the same popularity as a newly uploaded package. For this reason, weekly download counts will be used to quantify package popularity for the typosquatting defense proposed in this thesis.

Selecting a metric to quantify popularity is simply the first step towards establishing a distinc-

tion between popular and unpopular packages. A simple solution to this problem is to select a *popularity threshold*. Packages with weekly download counts greater than or equal to the popularity threshold are classified as popular while packages with weekly download counts below the threshold are considered unpopular. The only parameter required with this approach is the popularity threshold itself. Setting a popularity threshold that is too high would restrict the number of typosquatting targets, which would ultimately increase the detection scheme's false negative rate. Conversely, setting the popularity threshold too low would cause too many packages to be considered popular, granting them exemption from being considered a typosquatting perpetrator, and also increasing the detection scheme's false negative rate. In order to select an appropriate popularity threshold, analysis of package download distributions and package namespaces had to be performed.

5.3 Package Namespace Analysis

Transitive analysis of the npm and PyPI package namespaces was performed to explore the effect that a variable popularity threshold had on the degree to which typosquatting affected the repository. These repositories were specifically chosen because of their popularity and history with typosquatting attacks. The analysis began by collecting lists of all publicly available packages on each platform along with weekly download counts and full dependency trees. Using the set of all transitive dependencies was preferable to examining only direct dependencies as doing so more accurately emulated the real-world use cases of modern package repositories. By default, when installing a new package, the package manager also installs all transitive dependencies, not only direct ones. Once the package names and metadata had been collected, the analysis continued by utilizing the proposed typosquatting detection scheme.

For each of the recorded packages, the entire dependency tree was examined. Packages throughout the dependency tree were passed to functions which implemented each of the signals outlined in the typosquatting detection scheme. These functions were created to look for every other package on the repository which was similar to or confusable with a given moniker. For example,

when given the package name `loadsh`, the functions would return package names like `load_sh`, `lodash`, and `loads`. The most popular of these candidates (`lodash`, in this case) was labeled the most likely typosquatting target, and was recorded. This process is illustrated in Figure 5.1. After this operation had been performed for the entire dependency tree of all packages on both npm and PyPI, the process of determining the effect that popularity threshold would have on the degree of typosquatting could begin.

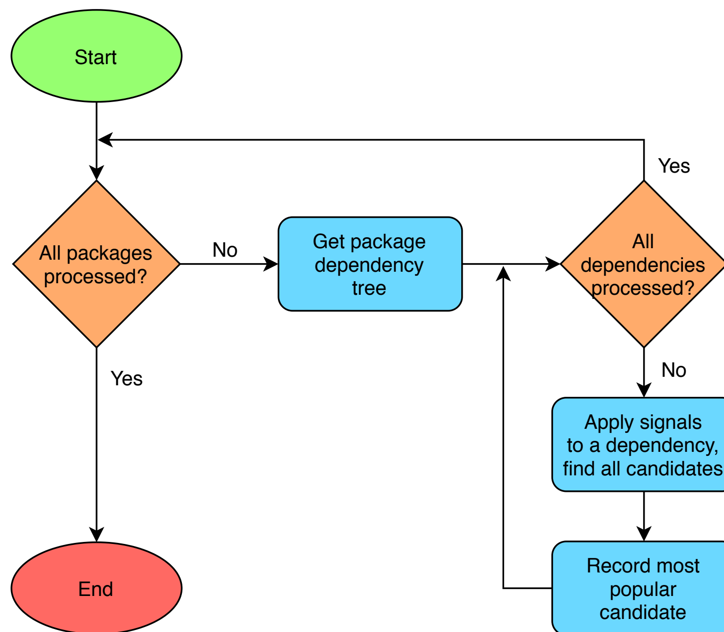

Figure 5.1: Transitive analysis of the repository namespace.

When determining appropriate popularity thresholds, one key question is worth keeping in mind. That is, how are download counts calculated? The creators of npm have provided insight into their download statistics. According to an official npm blog post, download counts are simply the number of HTTP 200 status codes returned for package archive requests [26]. This has a few important implications. First, local caches reduce package download counts. Downloading a package and utilizing it for several projects on the same machine, will result in only a single download. Second, downloads are not entirely the result of user interaction. HTTP requests can also come from repository mirrors and other scripts which download packages for large-scale analysis. In fact, npm states that a package can receive upwards of 50 downloads per day exclusively from

automated sources. In other words, a package could realistically be downloaded 350 times, all while never being requested by a single person. For this reason, 350 weekly downloads will be the absolute lower bound of the popularity threshold. Analysis will search for appropriate thresholds above this figure.

The degree of typosquatting was characterized in two separate ways: the percentage of the namespace and the percentage of all weekly downloads. Because packages are obviously not downloaded equally, it is important to examine both definitions of the degree of typosquatting. To determine these two measures, evenly spaced popularity thresholds are selected between the lower and upper bounds. At each of these thresholds, one counter records the total number of individual packages containing a potential typosquatting perpetrator anywhere in its dependency tree. Another accumulates the download counts of these packages. These values will accumulate so long as the dependency (or hypothetical perpetrator, e.g. `loadsh`) is below the popularity threshold and the potential target (e.g. `lodash`) is on or above the popularity threshold. Graphs of these values at each threshold can be used to illustrate the effect that popularity threshold has on the level of typosquatting in programming language repositories.

5.4 Package Manager Integration

To determine the effective temporal overhead that the proposed typosquatting detection scheme would impose on the standard package installation process, the official npm package installer was modified. The modified package installation workflow is illustrated in Figure 5.2. The revised installation process operates as follows:

1. The user issues an installation request, explicitly soliciting a package from the command line.
2. The package manager pulls metadata from the corresponding package repository to construct a dependency tree.

3. The package manager removes any packages from the dependency tree which are already installed on the user's system. A list of all packages that need to be installed is given to the typosquatting detection system.
4. All of the packages that remain in the dependency tree are scanned for typosquatting. Each of the signals described in Section 5.1 are applied to every package name.
5. A list of potential typosquatting targets is constructed.
6. If the list of potential typosquatting targets is not empty, then typosquatting has been detected. Details regarding each of the suspected typosquatting perpetrators are presented to the user. The user is asked if they would like to continue or abort the installation. Example prompts are shown in Figure 5.3.
7. The user's response to the prompt is captured.
8. If the list of potential typosquatting targets is empty, or the user has decided that they would like to continue the installation, then the standard installation process continues. All required packages are installed.
9. If all required packages are successfully installed, or the user has decided to abort the installation, the process gracefully comes to an end. If the user chose to abort the installation when presented with the typosquatting alert prompt, no packages are installed whatsoever.

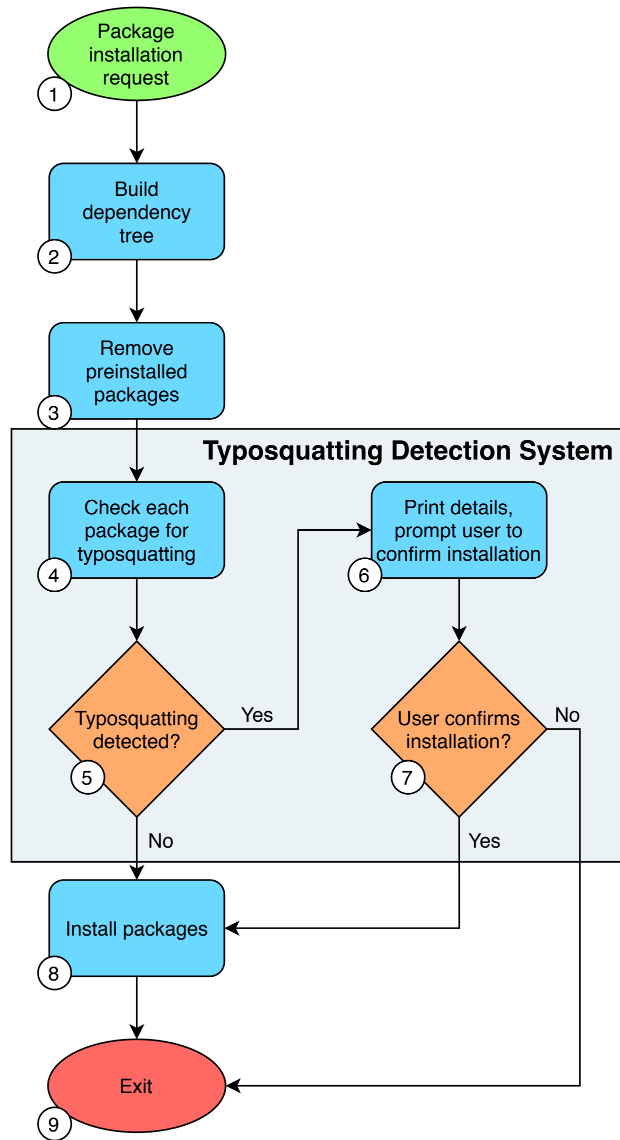

Figure 5.2: The modified package installation process.

```
C:\>npm install loadsh
WARNING! The requested package loadsh (2,031 weekly downloads) could
be typosquatting lodash (26,179,345 weekly downloads).

Are you sure you want to install loadsh? [Y/n] n

C:\>npm install seemingly_innocent_package
WARNING! The requested package seemingly_innocent_package depends on
requets (5 weekly downloads), which could be typosquatting request
(18,637,864 weekly downloads).

Are you sure you want to install seemingly_innocent_package? [Y/n] n

C:\>
```

Figure 5.3: Sample prompts warning users against both direct and indirect typosquatting.

After the installation process had been modified, one thousand packages were weighted by popularity and selected at random. Weighting packages in this context helps more accurately emulate real-world use cases. Each of these packages were first cached in order to minimize any network-based latency that could delay the package installation process. Once cached, the amount of time taken to install each of the packages via the official package manager was recorded. Then, packages were uninstalled. Finally, the amount of time taken to install each of the packages through the modified package manager was recorded, so it could be compared to the control. It is worth noting that the package manager should be configured to ignore any installation scripts during this test, to avoid executing potentially dangerous commands as a result of installing a large number of packages at random. The prompt which paused installation and requested input from the user was removed for this test so that the overhead of the typosquatting check alone could be measured.

5.5 Implementation and Infrastructure

All of the programs and modifications that were involved in the data collection and analysis described in the previous sections were written in Python 3 and Node.js. Due to the sheer size of the repository namespaces being analyzed, intensive operations routinely lasted hours, if not days. Clearly, this was undesirable. Distributed computing techniques were quickly employed to resolve this issue. The Information and Telecommunication Technology Center at the University of Kansas

comes equipped with a high-performance computing cluster. This cluster utilizes the Slurm Workload Manager which simplifies the process of managing a large number of jobs. Fortunately, the analysis performed here was extremely parallelizable, as the same general operations had to be performed to every package in the repository namespaces with no importance placed on the order in which packages are processed. Sets of package names were evenly distributed among more than one hundred nodes in the cluster, resulting in a dramatic speedup.

Chapter 6

Evaluation

6.1 Results

This section presents the direct findings of the experiments described in the previous chapter.

6.1.1 Download Distribution

When creating a defense dependent on the popularity of packages, which is based on their respective weekly downloads, it is quite important to understand the download distribution of the repositories to which they belong. These download distributions can be represented in many different ways. Figure 6.1 breaks down npm and PyPI repositories, showing what percentage of the platforms consist of packages in various download ranges. They demonstrate that a vast majority of packages on both platforms are downloaded quite infrequently. In fact, over 90% of the packages on both repositories fall below the 350 weekly download cutoff proposed by the creators of npm; below which, all downloads could theoretically come from non-human sources. These initial findings reveal a stunning imbalance in package downloads.

In other words, a relatively small subset of packages are responsible for an overwhelming majority of downloads. On a weekly basis, npm packages collectively receive around 18 billion downloads, where as PyPI packages collectively receive just over 1 billion downloads. The top one percent of all npm packages account for more than 98% of all weekly downloads. In a similar fashion, PyPI's top one percent account for nearly 96% of all weekly downloads. This means that, for all intents and purposes, that a majority users see no no need for a majority of packages on

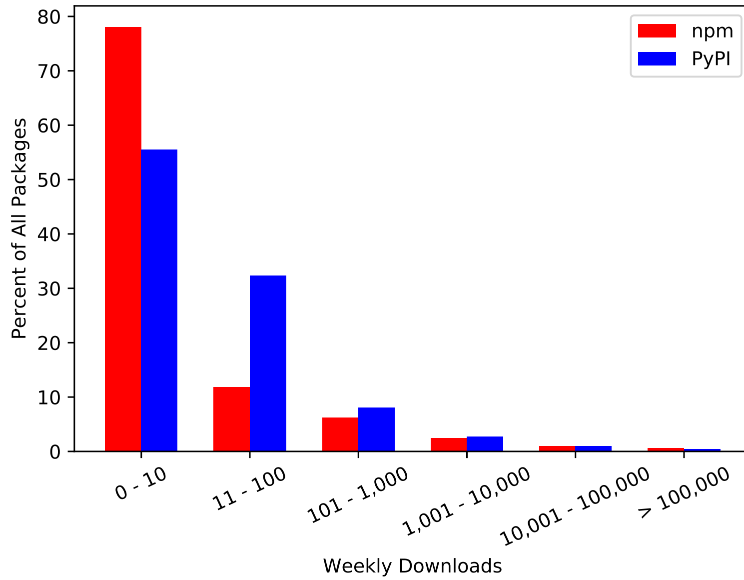

Figure 6.1: The download distributions of npm and PyPI.

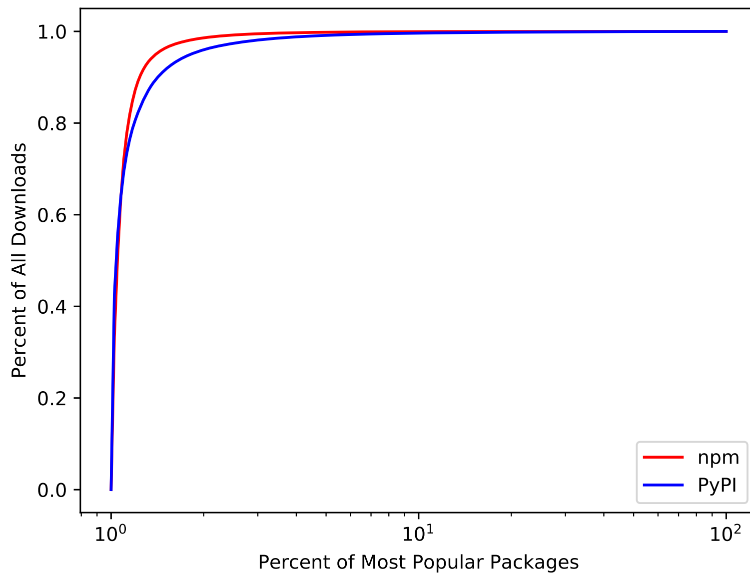

Figure 6.2: Cumulative download distribution for npm and PyPI.

these platforms. A cumulative distribution function representing what share of downloads belong to the most popular packages is plotted in Figure 6.2. Even with the log scale shown in the figure, the initial increase is staggering.

The fact that the an incredibly small percentage of packages are responsible for practically

all downloads leads to an interesting conclusion. The myriad of undesirable packages effectively pollutes the repository namespace and presents serious implications for the proposed typosquatting defense. Clearly, the popularity threshold somewhere near, if not above, the top one percent mark. Doing this ultimately forces the set of packages capable of being typosquatting perpetrators to include nearly every package in each repository.

6.1.2 Popularity Threshold

The results of the transitive namespace analysis described in the Section 5.3 are shown in Figures 6.3 and 6.4. Both of these figures show how a variable popularity threshold affects the degree of typosquatting from the previously established lower bound of 350 weekly downloads to the arbitrarily-chosen upper bound of 100,000 weekly downloads, which was considered to be the point beyond which packages were unquestionably popular. Figure 6.3 represents the degree of typosquatting as the percentage of all packages on the repository. Since, however, it was discussed in Section 6.1.1 that incredibly severe download imbalances exist on these platforms, the result was computed with downloads taken into consideration. Figure 6.4 shows the degree of typosquatting as a percentage of all weekly downloads.

The first noteworthy phenomenon displayed in these figures is the qualitative difference between the trends in Figure 6.3. The curve representing PyPI almost exclusive decreases as the popularity threshold rises, while the curve corresponding to npm steadily increases. The behavior of npm's degree of typosquatting is initially counterintuitive. As the popularity threshold rises, the number of popular packages, and therefore targets, decreases. With fewer targets, one would expect there to be fewer typosquatting perpetrators. This is the case for the PyPI curve, but not the npm curve. The unexpected growth in the npm curve is due to a rather interesting characteristic

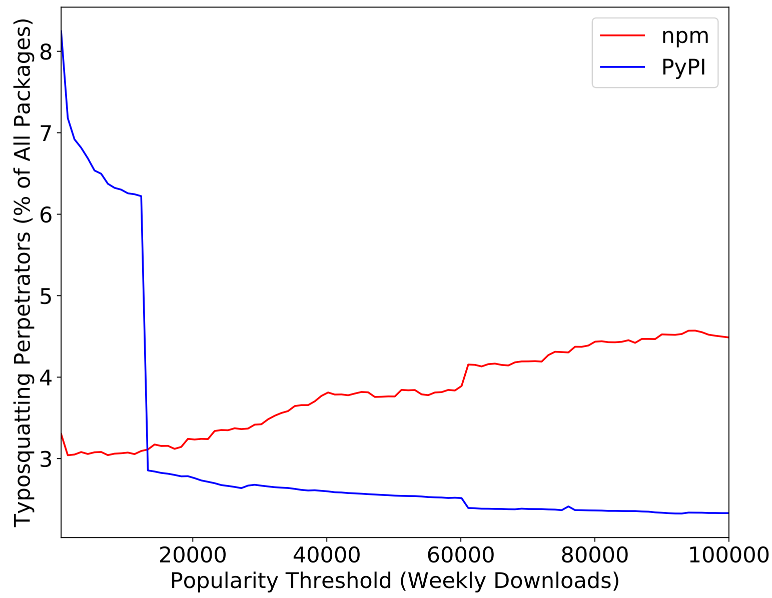

Figure 6.3: Percentage of all packages considered typosquatting perpetrators as a function of popularity threshold.

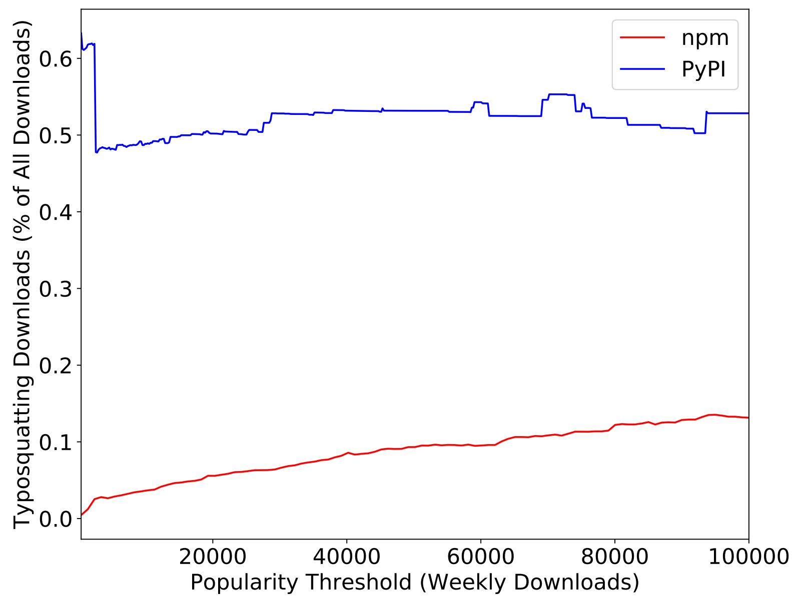

Figure 6.4: Percentage of all weekly downloads containing a potential typosquatting perpetrator as a function of popularity threshold.

of npm which is exploited by the implementation proposed in this thesis. The characteristic in question is the name similarity between reasonably popular packages. Table 6.1 shows examples of this phenomenon in which hypothetical typosquatting perpetrators are reasonably popular. The discrepancy between pairs shown in Table 6.1 is large enough that once the popularity threshold surpasses the less popular package, it loses its exemption and becomes a typosquatting perpetrator.

Package Name	Weekly Downloads
object-assign	16,759,898
object.assign	10,919,355
camelcase	36,187,784
camel-case	6,283,067
isarray	29,874,329
is-array	66,852
kind-of	51,413,827
kindof	24,273
memorystream	1,142,090
memory-stream	6,423

Table 6.1: Instances where typosquatting targets and perpetrators are both popular.

Although the packages in cases like these may not be malicious, a case could be made that they do have a negative effect on the quality of the platform. These cases could create confusion among package consumers, however this topic is out of scope for this thesis. The effect of this phenomenon, however, still must be taken into consideration when developing the proposed defense. An ideal popularity threshold should be set at a point where false positives like the cases shown above are minimized. To remove these false positives along with any other defensive typosquatting packages in either repository, a popularity threshold of 10,000 weekly downloads was selected. At this threshold, the increase in typosquatting perpetrators due to false positives in npm has yet to take place. Also, this threshold coincides with the steep drop in perpetrators observed in

PyPI, which is due to a relatively large number of targets with between 12,000 and 14,000 weekly downloads.

The selected popularity threshold has little effect on the total number of downloads affected by the proposed solution. As seen in Figure 6.4, the curves are fairly constant, for the most part, remaining around 0.5% for PyPI and 0.1% for npm. This means that for any reasonable popularity threshold, package consumers can expect to see alerts regarding the potential installation of a typosquatting risk once for every 200 packages installed on PyPI and for every 1,000 packages installed on npm. A low alert rate such as this helps to ensure that users will grow tired of seeing it and simply ignore its warning. Again, it is worth mentioning that the analysis performed here was transitive; meaning the rates discussed here are manual installs and do not count entire dependency trees, which as previously said, can be quite large.

6.1.3 Runtime Overhead Analysis

Modifications that were made to perform the timing analysis were made to version 6.12.0 of npm's command line tool. The test was performed on a machine with an Intel Core i9-9980HK processor clocked at 2.40 GHz and 32 GB of RAM. With package contents cached, the average installation time using the standard process (without the typosquatting check) was 2.604 seconds, while the average installation time using the modified process (with the typosquatting check) was 2.669 seconds. The addition of the typosquatting check added an average of 0.065 seconds to the standard installation process – an overhead of 2.5% – ultimately imposing no perceptible impact on package installation time.

6.2 Discussion

This section touches on the indirect findings of the experiments and other miscellaneous details that pertain to the work performed.

6.2.1 Undiscovered Typosquatting

After the transitive analysis of both npm and PyPI had finished, a list of the most popular packages flagged as potential typosquatting perpetrators was manually inspected. This was done with the goal of discovering novel typosquatting attacks. Interestingly, one of the first packages found was one named `loadsh` which was possibly targeting the popular utility – and most depended upon npm package – called `lodash`. Upon further inspection, `loadsh` was functionally identical to its well-renowned peer. The creator of `loadsh` had actually reuploaded a past version of `lodash` under a new name, making not a single change [24].

Despite containing no malware, this discovery was significant for a number of reasons. One could argue that this package was simply another case of defensive typosquatting. However, if this is true, it was the epitome of poor execution. In reality, the creation of `loadsh` introduced more harm than good. First, the exact version of `lodash` that was copied a since well-documented vulnerability called prototype pollution, which has been exploited to perform denial of service and remote code execution attacks. A new version of `lodash` was eventually released which patched the exploit, but `loadsh` was never updated and the vulnerability persisted. Second, `loadsh` had quite a significant user-base. At the time of its discovery, `loadsh` had been consistently earning 2,000 weekly downloads and had over 60 dependents. Due to this level of popularity, the creators of all packages that depended on `loadsh` were contacted to ask if they had intended to use `lodash` instead. All of the few developers who responded to the inquiry confirmed that their use of `loadsh` was accidental and were unaware of the mistake they had made. Eventually, `loadsh` was brought to the attention of the npm Security Team who further investigated the package, and ultimately deprecated it.

6.2.2 Limitations

The current implementation of package repository typosquatting detection scheme has room for improvement. Adjustments to the signals which described in Section 5.1 would have the most direct on the performance of the system. The current signals are relatively conservative in order

to reduce false positive. As a result, they are incapable of detecting more elaborate typos. For example, the signal which checks for swapped words strictly relies on the presence of delimiters. However, a common practice is to simply concatenate multi-token package names together, totally omitting punctuation. In these cases, swapping words cannot be performed unless word boundaries are assumed. Also, intersections of signals are not checked for as doing so would exponentially increase the execution time of the typosquatting check, though instances of historical typosquatting did contain intersections of multiple signals.

Another limitation with the proposed defense stems from its method of distinguishing between popular and unpopular packages. Separating these two types of packages using a single static threshold eliminates the possibility of detecting typosquatting pairs with less extreme differences in popularity. Regardless of where the threshold is set, there may be non-malicious packages which fall slightly below it and malicious packages which reside slightly above it. Adjusting it in either way does nothing to solve this. Essentially, this problem calls back to the Sorites Paradox because of its vague predicates. This paradox highlights the need for a more robust definition popularity. The `loadsh` example and the pairs listed in Table 6.1 show that typosquatting perpetrators could garner a non-negligible number of downloads, effectively making them indistinguishable from their targets. Any perpetrators that surpasses the static threshold would immediately become exempt from detection. In that vein, adversaries could theoretically manipulate the basic system which tracks downloads on these repositories to artificially inflate the popularity of a malicious package, granting it immunity from the proposed defense. The static differentiation between popular and unpopular packages basically creates a "gray area" of targets and perpetrators near the boundary. This "gray area" is likely responsible for most of this system's false negatives and false positives.

6.2.3 Typosquatting Signal Efficacy

Of the hundreds of previously discovered typosquatting attacks against the users of npm and PyPI, this set of signals were able to detect slightly more than 68%. The reason this figure is not higher is

primarily due to extended typosquatting campaigns against two npm packages named `js-sha3` and `buffer-xor` [44]. The typosquatting perpetrators involved in these campaigns contained stochastic character replacements. For example, these campaigns contained packages like `jsmsha3`, `js-shas`, `zs-sha3`, `js-sxa3`, `js-sha7`, `buffez-xor`, `buffgr-xor`, `buffer-xov`, `buffe2-xor`, and `bqffer-xor`. Although the typos made in these examples are unreasonable, they can still technically be considered typosquatting due to the context of the extended campaign. Modifying the signals to detect typosquatting of this nature would effectively revert back to basic edit distance. Doing this would inevitably and drastically increase the typosquatting detection system's false positive rate, as mentioned in Section 5.1. When omitting these campaigns, the aforementioned signals detect about 81% of historical typosquatting perpetrators. The remaining packages were undetected predominantly because they contained multiple signals simultaneously. For instance, `mogobd` (targeting `mongodb`) combines the omits and swaps characters while `koa-body-parse` (targeting `koa-bodyparser`) both inserts and removes characters. Combinations of signals are not checked in this implementation of the proposed defense as it would exponentially increase execution time, and therefore, the burden imposed on the end user's workflow.

Chapter 7

Conclusion

In recent years, package managers have completely transformed the software development workflow. The ability to quickly import packages allows developers to create more complex projects faster than ever. As a result, package managers have unsurprisingly gained astonishing levels of popularity. Unfortunately, adversaries began using package managers to spread malware. Due to the nature in which packages are installed, adversaries were able to employ a technique known as typosquatting, in which a malicious package is given a name incredibly similar to or confusable with a popular package. Hundreds of package typosquatting attacks have been orchestrated since the popularization of package managers.

This thesis focused specifically typosquatting attacks against package repositories. By extending the techniques used to detect typosquatting domain names, analysis was performed which uncovers the underlying degree of typosquatting in modern package repositories. The transitive namespace analysis revealed that typically 0.5% of all packages downloaded from PyPI and 0.1% of all packages downloaded from npm. Considering the sheer number of downloads each of these platforms receive each week, these percentages ultimately amount to millions of potentially compromised downloads. In an effort to protect users against these attacks, a programmatic defense against package typosquatting attacks was implemented. The proposed defense warned users of possible typosquatting during installation. With an average temporal overhead of 2.5%, this defense notified users of both direct and indirect typosquatting by checking not only the requested package, but the entire dependency tree. Users were presented with all the information regarding the suspected typosquatting attack and were asked to either continue or abort the installation. Although the presented implementation is far from perfect, it did confirm a majority of past ty-

posquatting attacks and even uncovered a novel typosquatting perpetrator that presented serious security risks.

The results presented in this thesis imply that a proactive, automated defense against package typosquatting is possible. Developers who make typos during package installation, those who install a package as a dependency, and application end users who never install packages are all better protected with the proposed defense. Repository maintainers could comfortably incorporate a typosquatting check to their official package manager releases thanks mainly to the low temporal overhead. Researchers can view this work simply as a single step towards protecting the useful package managers which have become commonplace in contemporary software development. Natural extensions of the work done in this thesis include improving the typosquatting signals and definition of popularity. The typosquatting signals constitute much of the defense's core functionality. More advanced signals will undoubtedly be able to detect more elaborate typosquatting perpetrators while simultaneously minimizing false positives.

Alternative applications of this proposed solution can be adopted to gain a variety of advantages. The implementation described in this thesis heavily relies on experienced and diligent users making the right decisions. Furthermore, it does nothing to prevent repository namespace pollution which can cause the previously discussed confusion that packages consumers may face when using a repository with the sheer scale of npm. If repository maintainers prefer to reduce clutter in their namespaces, the typosquatting check performed during package installation could instead be performed during package creation. Before a package is allowed to exist on the repository, it must be cleared of typosquatting. This solution would effectively extend the existing typosquatting defenses employed by npm discussed in Section 2.6 which check only for punctuation-based differences on an unknown set of packages. An approach such as this comes with considerable long-term performance benefits. Rather than typosquatting checks be performed on all packages for all users during installation (which would certainly test packages that have already been analyzed millions of times, if not more), the typosquatting check is performed a single time. Also, this approach could prevent typosquatting packages which contain malware from being offered

in the first place, creating a more secure development environment. False positives for a system implemented in this way, however, would restrict the naming freedom that users currently have on these platforms. The process of manually investigating and resolving false positive claims could create a large burden for repository maintainers and should be taken into account before a solution like this is deployed.

To both protect package naming freedom and eliminate repetitive testing, bulk typosquatting checks could also be performed by repository maintainers at times of their choosing. Similar to the strategy which led to the discovery of the `loadsh` package, the entire namespace could be analyzed and the most promising leads could be examined further. This, like the first alternative solution, comes with the added benefit of minimal retesting. Yet, this solution does nothing to protect users from accidentally installing malicious packages. For that reason, this solution is ideally implemented alongside the modifications to the package manager. The combination of these two techniques would both protect users from packages they unintentionally request and remove packages from the repository which may not be requested frequently.

The definition of popularity dictates what packages can and cannot be considered targets. A static threshold may be outperformed by a dynamic popularity differential system in which packages are considered typosquatting if they receive some fraction of the targets downloads, regardless of what the popularity of the two packages are. Dynamic strategies in this regard could aid in reducing the "gray area" side effect of the static threshold. Finally, improved definitions of package popularity could prevent adaptive adversaries from circumventing defenses. Taking the number of dependents into consideration may develop a more robust interpretation of popularity. These ideas are the next steps towards the everlasting goals of improving digital security and safeguarding the welfare of software developers around the world.

References

[1] Agten, P., Joosen, W., Piessens, F., & Nikiforakis, N. (2015). Seven months' worth of mistakes: A longitudinal study of typosquatting abuse. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS 2015)*: Internet Society.

[2] Alrwais, S., Yuan, K., Alowaisheq, E., Li, Z., & Wang, X. (2014). Understanding the dark side of domain parking. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (pp. 207–222).

[3] Bengtson, W. (2018). Defensive typosquatting packages created by pypi user wbengtson. `https://pypi.org/user/wbengtson/`.

[4] Bullock, M. (2017). Python package: pypi-parker. `https://pypi.org/project/pypi-parker/`.

[5] Cappos, J., Samuel, J., Baker, S., & Hartman, J. H. (2008). A look in the mirror: attacks on package managers. In *CCS*.

[6] Chakradeo, S., Reaves, B., Traynor, P., & Enck, W. (2013). Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13 (pp. 13–24). New York, NY, USA: ACM.

[7] Chatterjee, R., Doerfler, P., Orgad, H., Havron, S., Palmer, J., Freed, D., Levy, K., Dell, N., McCoy, D., & Ristenpart, T. (2018). The spyware used in intimate partner violence. In *IEEE Symposium on Security and Privacy* (pp. 441–458).: IEEE Computer Society.

[8] Chiew, K. L., Yong, K. S. C., & Tan, C. L. (2018). A survey of phishing attacks: their types, vectors and technical approaches. *Expert Systems with Applications*, 106, 1–20.

[9] Crussell, J., Gibler, C., & Chen, H. (2015). Andarwin: Scalable detection of android application clones based on semantics. *IEEE Trans. Mob. Comput.*, 14(10), 2007–2019.

[10] Denvraver, H. (2019). Malicious packages found to be typo-squatting in python package index. https://snyk.io/blog/malicious-packages-found-to-be-typo-squatting-in-pypi/.

[11] Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B., & Lee, W. (2020). Measuring and preventing supply chain attacks on package managers. *arXiv preprint arXiv:2002.01139*.

[12] Fulton, W. (2015). 14 ways to misspell google... that still take you to google. https://www.thrillist.com/tech/ways-to-misspell-google-that-redirect-to-google-domain-names-google-owns.

[13] Gagnon, Peacock, . V. (2020). What is typosquatting? https://www.gapslegal.com/articles/what-is-typosquatting/.

[14] Gonzalez, H., Stakhanova, N., & Ghorbani, A. A. (2014). Droidkin: Lightweight detection of android apps similarity. In *SecureComm (1)*, volume 152 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering* (pp. 436–453).: Springer.

[15] Hovsmith, E. (2017). Why brand monitoring is a security issue - typosquatting. https://www.anomali.com/blog/why-brand-monitoring-is-a-security-issue-typosquatting.

[16] ICANN (2020). Domain name dispute resolution policies. https://www.icann.org/resources/pages/dndr-2012-02-25-en.

[17] Imperva (2020). Phishing attacks. https://www.imperva.com/learn/application-security/phishing-attack-scam/.

[18] Kintis, P., Miramirkhani, N., Lever, C., Chen, Y., Romero-Gómez, R., Pitropakis, N., Nikiforakis, N., & Antonakakis, M. (2017). Hiding in plain sight: A longitudinal study of combosquatting abuse. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (pp. 569–586).

[19] Lhoussain, A. S., Hicham, G., & Abdellah, Y. (2015). Adaptating the levenshtein distance to contextual spelling correction. *International Journal of Computer Science and Applications*, 12(1), 127–133.

[20] McAfee (2013). What is typosquatting? `https://www.mcafee.com/blogs/consumer/what-is-typosquatting/`.

[21] Miller, F. P., Vandome, A. F., & McBrewster, J. (2009). *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press.

[22] Moore, T. & Edelman, B. (2010). Measuring the perpetrators and funders of typosquatting. In *International Conference on Financial Cryptography and Data Security* (pp. 175–191).: Springer.

[23] Norman, J. (2017). How to protect your domain from typosquatting. `https://www.techspark.co/blog/2017/12/29/protect-domain-typosquatting/`.

[24] npm (2020a). loadsh - npm. `https://www.npmjs.com/package/loadsh`.

[25] npm (2020b). npm by the numbers, official package and download counts. `https://www.npmjs.com/`.

[26] npm Maintainers (2014). The npm blog - numeric precision matters: how npm download counts work. `https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts`.

[27] npm Maintainers (2015). npm-scope | npm documentation. https://docs.npmjs.com/using-npm/scope.html.

[28] npm Maintainers (2017a). New package moniker rules. https://blog.npmjs.org/post/168978377570/new-package-moniker-rules.

[29] npm Maintainers (2017b). The npm blog - 'crossenv' malware on the npm registry. https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry.

[30] npm Maintainers (2019a). New security insights api: Sneak peek. https://blog.npmjs.org/post/188234999089/new-security-insights-api-sneak-peek.

[31] npm Maintainers (2019b). npm security insights api preview part 2: Malware. https://blog.npmjs.org/post/188385634100/npm-security-insights-api-preview-part-2-malware.

[32] npm Maintainers (2019c). npm security insights api preview part 3: Behavioral analysis. https://blog.npmjs.org/post/189189888357/npm-security-insights-api-preview-part-3.

[33] npm Security Team (2019a). npm malicious package report - cryptocurrency theft. https://www.npmjs.com/advisories/1415.

[34] npm Security Team (2019b). npm malicious package report - passwords and credit cards. https://www.npmjs.com/advisories/1106.

[35] npm Security Team (2019c). npm malicious package report - reverse shell. https://www.npmjs.com/advisories/1307.

[36] Research, D. (2018). Don't get reeled in by financial phishing scams. https://www.domaintools.com/resources/blog/dont-get-reeled-in-by-financial-phishing-scams.

[37] snyk (2020a). Vulnerability db - npm. `https://snyk.io/vuln?type=npm`.

[38] snyk (2020b). Vulnerability db - pip. `https://snyk.io/vuln?type=pip`.

[39] Spaulding, J., Nyang, D., & Mohaisen, A. (2017). Understanding the effectiveness of typosquatting techniques. In *Proceedings of the Fifth ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, HotWeb '17 New York, NY, USA: Association for Computing Machinery.

[40] Stats, P. (2020). Analytics for pypi packages. `https://pypistats.org/`.

[41] Stufft, D. (2015). Pep 503 – simple repository api. `https://www.python.org/dev/peps/pep-0503/#normalized-names`.

[42] Szurdi, J. & Christin, N. (2017). Email typosquatting. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17 (pp. 419–431). New York, NY, USA: Association for Computing Machinery.

[43] Szurdi, J., Kocso, B., Cseh, G., Spring, J., Felegyhazi, M., & Kanich, C. (2014). The long "taile" of typosquatting domain names. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (pp. 191–206).

[44] Taylor, M., Vaidya, R. K., Davidson, D., Carli, L. D., & Rastogi, V. (2020). Spellbound: Defending against package typosquatting.

[45] Tompkins, A., Bernasconi, J., Davidson, A., & Toohey, J. (2019). On security vulnerabilities stemming from the usage of open-source dependencies.

[46] Tschacher, N. P. (2016). *Typosquatting in Programming Language Package Managers*. Bachelor, University of Hamburg, Hamburg.

[47] Tunggal, A. T. (2019). What is typosquatting? `https://www.upguard.com/blog/typosquatting`.

[48] Vaidya, R. K., Carli, L. D., Davidson, D., & Rastogi, V. (2019). Security issues in language-based sofware ecosystems.

[49] Viennot, N., Garcia, E., & Nieh, J. (2014). A measurement study of google play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42 (pp. 221–233).: ACM.

[50] Wang, Y.-M., Beck, D., Wang, J., Verbowski, C., & Daniels, B. (2006). Strider typo-patrol: Discovery and analysis of systematic typo-squatting. *SRUTI*, 6(31-36), 2–2.

[51] Wermke, D., Huaman, N., Acar, Y., Reaves, B., Traynor, P., & Fahl, S. (2018). 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.

[52] Zimmermann, M., Staicu, C.-A., & Pradel, M. (2019). Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX* (pp.$\tilde{1}$7).