University of Hamburg
Department of Informatics

Bachelor Thesis

Typosquatting in Programming Language Package Managers

presented by

Nikolai Philipp Tschacher
born on May 1, 1991 in Tübingen
Matriculation Number 6632193
BSc Information Systems

submitted on March 17, 2016

Supervisor: Dr. Dominik Herrmann
First Reviewer: Prof. Dr.-Ing. Hannes Federrath
Second Reviewer: Dr. Dominik Herrmann

Objective

During the rise of the web 2.0, high level programming languages like Ruby, Node.js (Javascript), Python, Perl and PHP gained a lot of popularity which ultimately let to more libraries being hosted in centralized package managers. As it is known from typosquatting (Also: cybersquatting, domain squatting), the popularity of big websites may be misused by third parties by registering domain names that resemble their target domains, but contain some intentional typos. The same principle may be applied to package managers: If a user intends to install a popular software and mistypes/misspells the package name, a malicious squatted package is downloaded instead, which can lead to code execution (potentially with administrative privileges).

This work tries to show that the same methodology of domain squatting can be applied to package mangers. The main goal of the work is to estimate the risk of such attacks. Furthermore, this thesis tries to develop simple and effective countermeasures that package managers could employ.

In the empirical part of the thesis, semi-automatic and transparent uploads of typo-squatted packages to several registries are tried out in order to measure the severity of such an attack by counting the number of successful installations. It is expected that this form of squatting can be used to install malware an other computer systems. The number of successful installations is anticipated to correlate with the popularity and the propensity to misspell a certain package name.

The empirical part is to be followed by an analytical part. The analytical part generates ideas for countermeasures that allow repository maintainers or users to detect typosquatting attacks in the future. For this purpose potential typosquatting candidates could be generated for each legitimate package name with the help of the Levenshtein distance algorithms or Bayesian networks. Another option that can be considered is the Metaphone algorithm.

Abstract

In an empirical study, the concept of typosquatting was applied to programming language package managers such as Pythons central package repository (*PyPi*). Domain typosquatting is the act of purposefully registering hostnames which are similar to other, often popular, target hostnames. The reason for such malicious acts is to redirect traffic to a third party site that can then exploit the user's unintentional misspelling by showing advertisements or infecting the user with malware.

In some cases, the installation of third party programming language packages allows the package author to execute code on the installing user's system. A typosquatting attack simulation on the package repositories *pypi.python.org (Python)*, *rubygems.org (Ruby)* and *npmjs.com (Node.js)* by finding and uploading package names that are similar to famous existing packages, or by creating typos algorithmically, was conducted. Then information was collected of each single host that installed such a typo package.

The results of the attack showed that over 17 thousand distinct hosts installed those typo packages and executed their code. All these computers could have been infected with malware if malicious agents would have been the attackers instead. Around 50 percent of these confirmed installations were conducted with administrative rights, which increases the security impact considerably. A main result of the thesis is, that very few package names account for a large part of all installations. Furthermore, the meta data which the typo packages sent to the previously setup web server, was analyzed.

In the theoretical part of the thesis, existing research on DNS typosquatting was applied on programming language typosquatting. Several ways to generate typo candidates (Levenshtein distance and related algorithms) were presented.

Finally, this thesis presents various ideas on how to effectively protect users from the dangers of programming language typosquatting in an analytical part. It is hoped that package repositories will notice this work and consider using some of the presented defense techniques.

The target audience of the obtained results is the cybersecurity community and everybody involved in programming language package management administration and design.

Acknowledgements

I want to thank Dr. Dominik Herrmann, my supervisor, for his time and helpful ideas during many meetings in these five months of writing this thesis. Many thanks go to Dr. Dominik Herrmann and Prof. Dr. Hannes Federrath for agreeing on my proposal to *prove* successful installations with HTTP requests to a university web server. I want to thank my father, Prof. Dr. Wolfgang Tschacher, for helping me with many suggestions on how to improve the language and presentation of this work. My acknowledgments belong to Donald Stufft, one of the *PyPi* administrators, who was very cooperative and allowed me to continue the typosquatting experiment. I want to thank Michal Jaworski for sending me helpful suggestions and ideas about the distributed *notification program*. Furthermore, I am grateful for Robert Kerns warning to assign descriptions to all typo packages to clarify the intentions of the empirical experiment.

Contents

1 Introduction

1.1 Background and General Idea

Domain Name System (DNS) typosquatting exploits the fact that humans misspell names when typing them in the address bar of web browsers. However, misspellings can have much worse consequences than arriving on a wrong website. At worst, the website tries to exploit the user's browser with malware, which is not a simple task considering the protective mechanisms in modern browsers and the necessary, extremely complex attack methodologies (buffer and integer overflows, format string vulnerabilities, other low level memory corruption exploits in a sand-boxed environment).

In January 2015, the thesis author wondered whether this idea of abusing typos can be transfered to other areas than the DNS. By using the programming language Python for several years, it was learned that the third-party package manager *pip* (a command line application) is used to install software libraries from Python's community repository named *PyPi*. So the natural question that the thesis author asked himself was: How many users do commit typos when issuing an installation command in the terminal by using *pip*? Because everybody can upload any package on *PyPi*, it is possible to create packages which are typo versions of popular packages that are prone to be mistyped. And if somebody unintentionally installs such a package, the next question comes intuitively: Is it possible to run arbitrary code and take over the computer during the installation process of a package?

A pretest to the empirical study of this thesis was launched back in January 2015 by uploading some typosquatted packages on *PyPi*. Those packages were named such that they resembled well known and much downloaded package names. For instance, a package named *request* instead of the famous and often downloaded package *requests*, was created. In those fake packages, a short program was included which sent some data to a remote web server, that had been setup previously. This program (called *notification program* in the remainder of this thesis) notified the server whenever a user downloaded and executed the typo package and would thus record that somebody out there had installed one of these typo packages. The data transmitted included the information which is depicted in a Python dictionary in Figure 1.1.

Figure 1.1: Data that was collected in the pretest to this thesis in January 2015. Sending the environment variables was not necessary and can be considered hostile.

```python
data = {
   'ip': installed_package, # name of the typo package
   'ia': installed_at, # timestamp of the installation
   'ho': host_os, # name of the operating system and architecture
   'ar': admin_rights, # if the code was run with administrative rights
   'env': environ, # the shell/cmd environment variables
   'ii': ipinfo # IP address information of the host
}
```

Over the course of a few days, more than 50 people actually unintentionally downloaded those typo packages and thus executed the code of the *notification program*. This confirmed the previously stated assumptions: People frequently commit typos. Arbitrary code will be executed seconds after pressing enter on a mistyped install command.

The reaction of security researchers however was very quick: Within a few days, people would post threads in Internet forums about the *malicious* packages [red15; Mat15b] (The distributed packages did not cause any harm – Yet the sending of the above listed information can be considered hostile).

With this idea of typosquatting package repositories and the proof of concept, the present bachelor thesis was born. By expanding on the pretest, this thesis replicates the early experiment in a more systematic and broader study. It is tested whether other package repositories than Python are exploitable as well. The hypothesis is the following: It is assumed that people commit misspellings when they install packages with the package manager client in their favorite programming language. It is additionally expected that these typos can be exploited to such a degree, that it is possible to infect thousands of computers. Existing typosquatting can be observed in programming language repositories (Compare to section *Observing Typosquatting in the Wild* on page 53).

Starting from November 2015 until January 2015, 214 packages for the package repositories of the programming languages Python, Node.js and Ruby were distributed. 166 of these package names were created algorithmically with edit distance algorithms which covered all possible typos of two chosen names. The rest of these package names were chosen according to (assumed) human propensity of misspelling package names.

The main objective of this thesis is twofold. First, it is tried to prove empirically that there is a huge security gap in most recent programming language package managers (on the server and client side). Second, defensive actions are suggested which effectively protect the community against typosquatting attacks.

To the knowledge of the author, there is no prior published research about typosquatting in programming language package repositories. However, great effort in researching DNS typosquatting has been made. Those insights can be applied on this very specific research niche: Programming language package repositories.

1.2 Thesis Structure

The remainder of this thesis proceeds in the following way: In the second section, previous research in DNS typosquatting is explored. It is hoped that lessons learned in DNS typosquatting can leverage the fight against typosquatting in package managers, since many of the underlying concepts are closely related. In the second part of the theoretical section, the thesis introduces the reader to various edit distance algorithms which are used in a later part of the thesis to detect and generate typo packages.

Then in the following chapter, the experimental design of the empirical phase is explained. After having discussed the architecture and methodology of the experiment, results of the empirical phase are presented in the fourth section. Then the practical implications of the findings are elaborated in the next section (mostly defensive actions). Afterwards the thesis finalizes with a discussion of the results in which the impact of the findings is analyzed and the validity of the approach is discussed.

1.3 Terminology

In the following thesis, often specific concepts and terms are used which are not intuitively understandable. Therefore, the definitions of some of the most used concepts follows. These definitions can only be understood when progressing to later chapters of the thesis.

- **Package manager (client)**: Software which downloads, extracts and installs third party libraries which were uploaded by programmers. The package manager client is normally used in a terminal. In this thesis, *programming language package managers* often means both, the client side and the server side (*package repository*). When only the server side is meant, the term *package repository* is chosen. Examples for *package manager clients*: *pip, npm, gem.*
- **Package repository (server)**: Server software which hosts third-party packages for a specific programming language. Examples: *pypi.python.org (PyPi), npmjs.com, rubygems.org*.
- **Notification program**: Software written in the same programming language as the targeted package manager which is executed upon mistyping and downloading a typo package. This program collects some host specific but not personalized data and sends it to a web server hosted in the university network.
- **Unique installations**: An installation of a typosquatted package can be confirmed when a HTTP GET request with the payload data in the parameters is received. Often such installation requests occur more than one single time. Therefore, to count each successful installation only once, requests are considered unique only, if the IP address and the data in the parameters are distinct. For example, a user with a static IP address, who downloads first a *PyPi* package and then a week later a *rubygems* package, would be counted as having issued two unique requests.
- **Typo victims**: Everyone who installed one of the distributed typo packages. Identified by the IP address and the timestamp of the installation.
- **DNS typosquatting**: The act of intentionally registering a domain name to profit from traffic of users misspelling domain names. Also known as *typosquatting* or *domain squatting* or *cyber squatting*.
- **Interactions**: The number of unique *(IP address, administrative rights, host operating systems)* tuples from the installation database. There are more *interactions* than distinct IP addresses, because a single user (one IP address) may install packages with alternating administrative rights or different operating system versions.
- **Infection**: The process of downloading a typo package and executing the *notification program*. The term *infection* was chosen to emphasize that a malicious attack could have happened instead.

2 Theoretical Background of Typosquatting

2.1 Differences to Domain Name System Typosquatting

Although the simple idea behind typosquatting seems to be well known in the scientific community, no academic research efforts have been made to investigate typosquatting in other areas than the domain name system (DNS). There are virtually no prior empirical experiments in typosquatting programming language package managers, that the thesis author is aware of. This however does not mean, that package repository administrators are not aware of the dangers of typosquatting [Bal15]. The insights gained by academic DNS typosquatting research were of paramount importance for problems encountered during this thesis. Therefore, an overview to the state of the art research in typosquatting is given.

It must be understood that most existing research about DNS typosquatting is divided in two parts: Finding typosquatting domains and analyzing those domains with automatic crawlers. Often papers also propose defensive mechanisms against typosquatting. The focus in previous research was mostly on techniques involved in locating typosquatting domains. The analytical part of those papers is substantially different from the one of this thesis, because DNS typosquatting deals with advertisements (and not so much with malware [SKCS14]), whereas the consequences of typosquatting in programming languages often lead to malware being installed on computers, as this thesis will show in its subsequent chapters.

2.2 Previous Research

Moore and Edelman make use of the *Damerau-Levenshtein (DL)* distance and a self defined *fat-finger* distance to identify potential typosquatting websites in their work [ME10]. They define the *DL* distance as the *"minimum number of insertions, deletions, substitutions or transpositions required to transform one string into another"*. They also make use of the so-called *fat-finger* distance, which includes neighboring keyboard characters: Only characters are used that are approximate to each other on a QWERTY keyboard. In their work, Moore and Edelman only considered domains of 5-15 characters in order to reduce the risk of generating false typos. Then they applied the *DL* algorithm with a maximum distance of two and the *fat-finger* distance of one and two. They concluded that *"typo domains with Levenshtein or fat-finger distance one of popular domains where overwhelmingly confirmed as true typos"*. They also observed that the *fat-finger* distance typos of popular domain names were more often confirmed as typos as typos generated with the *DL* algorithm and the same edit distance [ME10].

Moore and Edelman mention various defenses against typosquatting. One of them is the legal framework *Anti-cybersquatting Consumer Protection Act (ACPA)*, which was published as legal lawsuit framework against typosquatting in 1998. Additionally, ICANN introduced the *Uniform Domain-Name Dispute-Resolution Policy (UDRP)* that eases the process of deciding whether a domain name infringes another website or not [ME10]. Their conclusion is two faced: While it is easy to identify typosquatting sites and their mostly few large providers, the problem will not cease to exist as long as advertisement networks will not stop to support and corporate with typosquatters [ME10].

Wang et al. from Microsoft Research developed a method called *Strider Typo-Patrol* to reveal *“large-scale systematic typosquatters”* [WBWV06]. Their system consists of three major parts: A typo-neighborhood generator, typo-neighborhood scanner and a domain parking analyzer. As previously stated, this thesis is mainly interested in the typo generation methods, since they might help to identify typo packages for the empirical part of the thesis. Wang et al. make use of five different *typo-generation models*:

1. Missing-dot typos: The dot following the *www* subdomain is removed. Example: *wwwexample.org* instead of *www.example.org*
2. Character-omission typos: Characters are deleted in the typo variant. Example: *gogle.com* instead of *google.com*
3. Character permutation typos: Following characters are interchanged: Example: *example.org* becomes *exmaple.org*
4. Character-replacement typos: Characters are replaced according to a proximity table on a standard keyboard. Example: *example.org* becomes *wxample.org*
5. Character insertion typos: Characters are inserted from a proximity table. Example: *example.org* becomes *exasmple.org*

Their *typo-generation model* resembles the *Damerau-Levenshtein* distance metric. Wang et al. should have at least detected this close relation. Their study is from 2006 and thus relatively old. However, it is the first attempt at a systematic approach to reveal typo domains. They discovered that *“the top six parking services amount for 30% of all algorithmically generated typo domains and 40%-70% of active ones”*. Another critique point is, that they focus only on a small part of the *Alexa* top domain list [Ale] to generate typos [WBWV06].

A relatively recent empirical study called *The Long "Taile" of Typosquatting Domain Names* (2014) [sic!] from Szurdi et al. examined *“the whole popularity distribution”* of typos instead of focusing just one the most popular hostnames [SKCS14]. The study revealed that 95% of typo domains focus on less popular domains and that the advertisement monetization infrastructure also exists for these domains. They estimate that around 20% of all *.com* domains are true typo domains (21.2 million hosts). Szurdi et al. make use of *Damerau-Levenshtein* algorithm with distance one. They also include typos that arise through the missing of the dot between the *www* subdomain and the hostname. They are also aware of the *fat-finger* distance. Their data source includes the whole *.com* zone file with 106 million domain names and the *Alexa* list of top one million sites [SKCS14].

Szurdi et al. generated the candidate typo list with the following edit operations: Addition, deletion, substitution of one character, transposition of neighboring characters. They found out that this generated set includes 4.7 million existing candidate typos. Then they proceeded to crawl this set of typos to further classify the results with content related techniques. Their work is very good in illustrating that the majority of typo domains do not target the most popular domains. Szurdi et al. finish their work by publishing their *YATT (Yet Another Typosquatting Tool)* framework, which essentially allows DNS servers to blacklist and filter out typo domains. They also created a frontend for the framework (as a *Firefox* plug-in) which contacts their typo protection DNS server. In this bachelor thesis, mostly popular package names have been used as a basis for typo generation. Future work could proceed similarly as Szurdi et al. and exploit typos in the *Long "Taile" of Typosquatting Names* [sic!] [SKCS14].

Kahn et al. developed an *intent reference technique* to find typosquatting domains by passively observing web traffic and detecting typosquatting typical behavior like: High bounce rates after visiting the typo website, direct entering of URL's and visits to other, more popular sites after mistakingly having visited the typo site[KHLK15]. The passive data source consists of HTTP and DNS requests recorded in a large U.S. university network. The HTTP data was anonymized (hashing of IP addresses) before the analysis was conducted. Another dataset consists of HTTP/HTTPS traffic logs from a large technology service provider [KHLK15].

A common disadvantage of lexical techniques (such as edit distance algorithms) to determine typo candidates is the potentially high false positive rate. Kahn et al. argue that it is easy to make mistakes: *Nhl.com* could be considered a typo of *nfl.com*, although both domains are legitimate websites. Disadvantages are that lexical models *"might not capture all user typos, and it cannot capture typos which are not yet registered"*. Kahn et al. are convinced that to determine user harm, the analysis must compare between the lost time of seeing a *website not unavailable error* in a browser, and the time wasted through activity of users on a typosquatters website. The typo happens either way – They argue that it might even be the case that the presence of an annoying typo leads to a quicker correction of the users mistake [KHLK15].

Kahn et al. introduce a *Conditional Probability Model* to determine the probability whether a specific action can be considered the direct cause of typosquatting. The difference between a typosquatting site and two sites which are unrelated but still possess a close *Damerau-Levenshtein* distance are, that in the former, the original intent to visit another site still exists [KHLK15]. This *"aggregate evidence that visits to a given site are almost always followed by visits to a lexically similar site, without the converse being true"* is the fundamental property of their *Conditional Probability Model*. This alternative metric to find potential typo candidates identifies typo sites based on a cunning mix of behavioral heuristics and lexical similarity and is definitively an improvement over typo identification with string similarity metrics only. Their *Conditional Probability Model* model could also be used by package repositories as a defense mechanism, because all user interactions are logged in real time on the web server's log files [KHLK15].

Agten et al.'s study is the first longitudinal research about typosquatting in which they observe the top 500 domains of the Internet over the course of seven months [AJPN15]. By collecting over 900 GB of typosquatting data, they are *"able to investigate the changes of typosquatting over time"*. Their main result is, that most top sites do not make use of defensive registrations. They were also able to find out that 50% of all typo domains belong to only four typosquatting providers and that 75% of short popular domain names are already registered. Thus typosquatters increasingly proceed to register longer typo names and change their monetization strategy over time. Their typosquatting model uses the *Damerau-Levenshtein* distance of one and *fat-finger* distance of one. Their model is closely related to the model of Wang et al. [WBWV06; AJPN15].

Agten et al. observed the number of typo category changes over time: They plotted an average of 2.84 typo category transitions during the seven months (sites changing from malicious to legitimate and vice versa) [AJPN15]. Future work that builds on top of this thesis may make use

Figure 2.1: Implementation of the *Hamming* distance in Python.

```python
def hamming_distance(a, b):
    """
    Computes the Hamming distances between strings a and b.

    requires:
        len(a) == len(b)

    ensures:
        len(retval) <= len(a)
    """
    if not len(a) == len(b):
        raise Exception('Strings a and b must have the same length.')
    n = 0
    for i, j in zip(a, b):
        if i != j:
            n += 1
    return n

if __name__ == '__main__':
    print(hamming_distance('alpha', 'aupna'))
```

of the longitudinal aspect of Agten et al.'s study: Observing typosquatting in package managers over time could reveal hidden trends and patterns [AJPN15].

2.3 Edit Distance Algorithms

Edit distance algorithms *"allow to delete, insert and replace simple characters in both strings"* and each of these operations can have different costs [Nav01]. There are a variety of different string edit distance metrics [Nav01]. In this section, the most important ones are presented and then they are introduced with some illustrative code in Python. For our purposes, the *Levenshtein* distance and the *Damerau-Levenshtein* distance metrics are important algorithms, but there are also other approximate string matching algorithms. This thesis makes use of the edit operations of the *Levenshtein* distance in order to generate typo names for *npmjs.com* algorithmically (The process of creating these names and the algorithm used can be looked up in section *Generation of Typosquatting Targets* on page 16).

2.3.1 Hamming Distance

The *Hamming* distance is given by the number of different symbols in two equally lengthy strings, by comparing characters with the same index only [Ham50]. The *Hamming* distance of the two strings *abcdq* and *qbdde* is three, because the first, third and fifth character are different. Therefore, the *Hamming* distance only allows replacements (substitutions) with cost one as simple edit operation [Nav01; Ham50]. This simple metric was implemented using Python and can be reviewed in Figure 2.1.

2.3.2 Levenshtein Distance

The *Levenshtein* distance is defined as the minimal amount of insertions, replacements and deletions to transform a source string *a* into the target string *b*, whereby each of these three basic operations have the same cost [Nav01]. The *Levenshtein* distance is named after the Russian mathematician Wladimir Levenshtein, who introduced this string similarity metric in 1965 [Lev66]. The *Levenshtein* distance was developed to improve signal processing by creating new electronic error correction codes [Nav01]. Physical transmission of signals is very error prone and there was a need to find the original message after a failed delivery. Therefore, such error correction codes have been developed and the *Levenshtein* distance was invented [Nav01]. The *Levenshtein* distance is defined recursively on two strings *a* with length *i* and *b* with length *j* for which $i, j > 0$ as follows [RY98]:

$$D_{a,b}(i,j) = min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} \tag{2.1}$$

whereby $1_{(a_i \neq b_j)}$ is the indicator function, which is 1 whenever $a_i \neq b_j$ and 0 otherwise. An alternative form of the recurrence relation for the *Levenshtein* distance $D(i, j)$ for strings *a* with length *i* and a string *b* with length *j* can be taken from [Nav01]:

$$\begin{aligned} D(i,0) &= i && (2.2) \\ D(0,j) &= j && (2.3) \\ D(i,j) &= \text{if } (x_i = y_j) \text{ then } D_{i-1,j-1} && (2.4) \\ \text{else } & 1 + min(D_{i-1,j}, D_{i,j-1}, D_{i-1,j-1}) && (2.5) \end{aligned}$$

The *Levenshtein* algorithm is mostly implemented using dynamic programming in a bottom up fashion [RY98]. A matrix is populated with edit distance values between all substrings of *a* and *b*. The running time of the algorithm is $O(|a||b|)$ in the worst and average case, whereas only $O(min(|a|,|b|))$ space is needed [Nav01]. The *Levenshtein* distance algorithm is implemented in Figure 2.2. When the code in Figure 2.2 is executed, the Script calculates the *Levenshtein* distance for the strings *kitten* and *sitting* and prints the dynamic programming matrix generated during the execution of the algorithm.

2.3.3 Damerau-Levenshtein Distance

Whereas the *Levenshtein* algorithm consists of deletion, insertion and substitution operations, the *Damerau-Levenshtein* algorithm adds transpositions of two neighboring characters to the set of allowed operations [Dam64; Lev66]. Damerau lists the following edit operations examples in his paper: ALPHIBET instead of ALPHABET (substitution), ALHPABET instead of ALPHABET (transposition of neighboring characters), ALLPHABET instead of ALPHABET (insertion) and ALPABET instead of ALPHABET (deletion) [Dam64].

The American mathematician Fred J. Damerau worked similarly as Levenshtein on error correction codes and published his paper on spelling errors in 1964 [Dam64]. In this thesis, the *Damerau-Levenshtein* string distance metric is not further used, because the edit operations of the classical *Levenshtein* algorithm are sufficient.

Figure 2.2: Implementation of the *Levenshtein* distance in Python.

```python
import pprint

def levenshtein(a, b, debug=True):
    """
    Computes the Levenshtein edit distance between
    the string a and string b.

    requires:
        (len(a) > 0 && len(b) > 0)

    ensures:
        (return >= 0 && return <= max(a, b))
    """
    if not (len(a) > 0 and len(b) > 0):
        msg = 'Strings a and b cannot be empty.'
        raise ValueError(msg)

    D = []
    for i in range(len(a)):
        D.append([])
        for j in range(len(b)):
            D[i].append(0)

    for i in range(len(a)):
        D[i][0] = i

    for j in range(len(b)):
        D[0][j] = j

    for i in range(len(a)):
        for j in range(len(b)):
            cost = [1, 0][a[i] == b[j]]
            D[i][j] = min(D[i-1][j] + 1,
                D[i][j-1] + 1, D[i-1][j-1] + cost)

    if debug:
        pprint.pprint(D)

    return D[-1][-1]


if __name__ == '__main__':
    print(levenshtein('kitten', 'sitting'))
```

2.3.4 Longest Common Substring Distance

The longest common substring distance of two strings is the minimal number of characters to be deleted and inserted in order for them to become equal [Nav01; NW70]. The longest common substring distance between the words *alpha* and *axxha* is 4, because in each string two characters need to be removed and inserted in order to get two identical strings. Both edit operations do cost one. The name of the distance originates from the fact that the algorithm considers the longest common substring, before it starts to delete or insert characters to approximate both strings [Nav01]. Furthermore, the distance is symmetric and the property $0 \leq d(x,y) \leq |x| + |y|$ does hold [Nav01].

2.3.5 Q-Gram Distance

When two strings a and b are split up in q-grams (whereby $q \leq min(a,b)$) and count the number of times these q-grams do not appear in the other string, the q-gram distance of two strings is obtained [Ull77].

2.3.6 Metaphone Algorithm

The *Metaphone* algorithm is the progression of the *Soundex* algorithm and was first developed to identify different surnames by Lawrence Philips in 1990 [LKDR07; Phi00]. The *Metaphone* algorithm transforms a string to an encoding, which has a related phonetic pronunciation. When two strings output an identical phonetic hash value, they are regarded to have the same pronunciation [LKDR07]. This makes it possible to apply the *Metaphone* algorithm on two strings, a and b, whereby b is a typo version of the string a, and yield the same phonetic representation for the two inputs: $Metaphone(a) = Metaphone(b)$ (This concept is called *similarity key* [PN11]). In comparison to the *Levenshtein* distance, the *Metaphone* algorithm only works with English words [Phi00]. Furthermore, phonetic information is processed in the algorithm, whereby the *Levenshtein* algorithm cannot include such linguistic meta information.

Using phonetic algorithms to create typo candidates that have a higher possibility to be misspelled is certainly the way to go in a systematic approach, but it remains unclear how well a phonetic algorithm would work with package names, that are not necessarily related to the English language. Another constraint of the *Metaphone* algorithm is its proprietary license (Compare with `http://www.amorphics.com/buy_metaphone3.html`, accessed on 15th March 2016) and the difficulty in obtaining a recent implementation for free. After all, the proposed defenses should be based on openly accessible algorithms, such that third party package repositories, which are often community driven, can make use of them.

3 Methods

Before the empirical study could begin, the main intention with this thesis needed to be declared: That some of the biggest and most frequently used package managers for much used programming platforms, can be attacked with typosquatting methods, which were exhaustively researched in combination with the domain name system (DNS). In other words: It will be shown that the state of the art security of package managers is so inadequate, that it allows averagely technical gifted attackers to launch large distributed attacks and even create botnets within just a few weeks.

The experimental setup which was established before the study could begin, was rather simple and essentially the same as with the pretest back in January 2015. First and foremost, a simple web application with the *Django* web framework and *nginx* as a web server was developed (1). The web application basically allows to store parametrized HTTP requests in a *sqlite3* database for further evaluation. The web application also included *CRUD* functionality and a basic statistical overview over all received installations. The University of Hamburg provided the thesis author with a virtual private server (running Debian) in their datacenter to host the application. The server was thoroughly tested in a local environment, and after setting it up using *Uwsgi* on the university web server, also remotely.

Then in a second step, the *notification program* that would inform the web server about a successful package installation, was implemented for the three chosen target programming languages (Python, Node.js and Ruby) (2). The logic in the implementation of these three *notification programs* was essentially identical. Their task is to run as stable as possible (on all major versions of the language still in use) and send a HTTP request with the collected data to the web application. It was not sufficient to write only one version in one programming language, because the targeted systems would only guarantee, that the language of the installing package manager was available.

In a further step, a template package for all three package managers was created and tested (3). Then the *notification program* was included in those three template packages. The advantage of this approach is, that only meta information like package names and package descriptions needed to be changed, whenever a new typo package was uploaded.

Then the typos were created. Typo names have been chosen by examining the most downloaded packages of the targeted package repositories and by finding lucrative typo names manually in a creative, mental process (4). For two popular package names (*request* and *async*) of *npmjs.com*, all possible typos with edit distance one were generated algorithmically.

The final step encompassed the creation and upload of all packages for all used packages names to the package repositories (5). After all packages were unregistered, the *sqlite3* database with the saved installation data was downloaded from the server for the empirical analysis of the results.

Table 3.1: Total number of downloads for some well known package repositories [DeB]. Names in cursive font are the repositories targeted in this thesis. Source: *http://www.modulecounts.com/*, accessed on 24th February 2016.

Package repository	Total downloads	Average daily growth
Npm (node.js)	244180	414/day
Maven Central (Java)	133901	95/day
Rubygems.org	114792	47/day
GoDoc (Go)	113664	267/day
Packagist (PHP)	87044	120/day
PyPI (Python)	75269	74/day
Nuget (.NET)	50679	35/day
Bower (JS)	49298	46/day
CPAN	33435	6/day
CPAN (search)	33435	6/day
Drupal (php)	33288	12/day
Clojars (Clojure)	15175	11/day
Hackage (Haskell)	9391	2/day
CRAN (R)	7972	6/day
Crates.io (Rust)	4156	9/day
MELPA (Emacs)	2956	2/day
Hex.pm (Elixir/Erlang)	1597	9/day
LuaRocks (Lua)	976	1/day
Pear (PHP)	602	0/day
Perl 6 Ecosystem (perl 6)	569	1/day

3.1 Targeted Programming Languages

As mentioned before, the programming languages Python, Node.js and Ruby were targeted during this thesis. All of these computer languages are high level scripting languages and are much used in scientific research (Python with its *numpy, scipy and matplotlib* frameworks for scientific computing), web work (all of them, but mostly Node.js and Ruby with its *Ruby on Rails framework*) and system administration tasks. Table 3.1 shows the total number of downloads of some well known package repositories [DeB]. The chosen programming language repositories have a large community of contributors and many packages with high download numbers (even though the download count and number of uploaded packages does not tell much about the real industry prevalence of said languages). Their rank by total downloads is 1. (*Node.js*), 3. (*Rubygems.org*) and 6. (*PyPi*) as Table 3.1 shows. Other reasons for choosing these languages besides the popularity, are going to be discussed in the next section.

3.2 Prerequisites for Typosquatting Attacks

Not every package manager is vulnerable to typosquatting attacks. There are some requirements which need to be taken into account, before attacking a package repository (Listed in decreasing importance in the ordered list below). The first two points in the following list need to be fulfilled, otherwise the package repository is not attackable by typo squatting attacks.

1. The possibility of registering any package name and uploading code without supervision.
2. The feasibility to achieve code execution upon package installation on the host system.
3. Accessibility and presence of good documentation for uploading and distributing packages on the package repositories.
4. Difficulty in quickly learning the target programming language.

This kind of attack works best, if code is directly executed upon a module download, triggered by the package manager client program. In Python, each package that is publicly registered, needs to have a *setup.py* file that contains package meta data such as names, description and fixtures belonging to the package. Whenever a user installs a package from the *PyPi* package repository, this *setup.py* is executed by a local Python interpreter. This means, that it is possible to hide the import statement for the *notification program* in the *setup.py* file and thus execute code there.

Node.js and its package manager, *npm*, provide various hooks on specific events to execute code. There is also a *preinstall* option that can be set in the *package.json* file, that provides options and metadata for a published Node.js package. It is favorable to write this preinstall script also in Javascript and execute it with the *node* binary, because *node* is guaranteed to be installed on the target system, when *npm* is used to install third party packages.

Achieving code execution with Ruby was slightly trickier. There is no official way (like in Node.js) or easy method (like in Python's *setup.py* file) to execute code upon installing packages with the Ruby package manager named *gem*. However, code execution was achieved by creating an empty native Ruby extension and placing the notification code in a Ruby extension configuration file named *extconf.rb*, which is interpreted during the pseudo build process. It is assumed that not all users who were tricked to install Ruby typo packages, would execute the notification code, because the hosts need a native build stack and compiler suite in order to execute the *extconf.rb* file and build the native source code. This method to gain code execution during the installation process was inspired by Victor Costan, who blogged about this as early as 2008 [Cos08].

Several other programming languages were also evaluated and tested for their attacking propensity. The next criteria, the quality of package repository documentation, was also satisfied by Python, Ruby and Node.js. During this thesis, more than three package managers were intended to attack, but some of them had such a bad documentation quality (*CPAN*, the Perl package repository, for example) that they were ignored altogether. The following section further elaborates why some programming languages were rejected.

3.3 Rejected Programming Languages

It has been tried to add more programming platforms to the typosquatting attack. *Bower (Javascript), CPAN (Perl), Nuget (.NET), Packagist (PHP)* and *R (CRAN)* were examined closer, and for each of them, reasons were found, to not include them in the attack.

There is no way to execute code upon package installation in *Bower* – The authors stated rightfully in a open discussion, that allowing code to be executed, would be a huge security risk and thus was made impossible [Sc13]. The *CPAN* ecosystem was simply too complex and cumbersome to try to attack. The declining popularity of *CPAN* and Perl in the past years was another reason to exclude it from research. *Nuget* from .NET seems to be a good candidate for a typosquatting attack. It was not included in the research because of time problems. There are nevertheless many indications that *Nuget* is exploitable by typosquatting attacks as well: Jeff Handely, core developer of the *Nuget* team, wrote in a blog post from October 2014 that *"Nuget is broken by design"* and that *"Many are still shocked that NuGet allows packages to run arbitrary PowerShell scripts during package installation"* [Han14].

Packagist (PHP) is not vulnerable to direct code execution upon package installation, because all installed packages are stored as dependency in sub folders, which are never directly touched. *CRAN (R)* was not investigated closer because of time pressure. However, it would make sense to examine it closer in future work.

3.4 Generation of Typosquatting Targets

After having chosen the programming languages to be attacked, the typo packages names needed to be generated. Each programming language package repository has a statistical site with a list of top installed packages [rub; Tai; npm]. This information was used to determine possible typo candidates. Another information source for packages that exhibited a propensity to be misspelled, was the experience of the author with the language Python: Over the course of years, many misspellings were made and various pitfalls in relation to package names and their installation were learned.

Once the set of names to attack were found, typo variations were generated: These names were either found in a creative mental process or they were generated with the *Levenshtein* algorithm. All possible typos with *Levenshtein* distance one for the two package names *request* and *async* were created, to find out more about the popularity distribution of those two names. Those two package names were chosen, because they are both popular packages in Node.js (*npmjs.com*): *Request* had 570715 daily downloads and 13308774 monthly downloads. The download count for *async* with 1096407 daily downloads and 26292179 installations in the last month is even higher (Data accessed on the 19th February 2016).

All the edit distance algorithms discussed in the theoretic section incorporate different use cases. In this thesis, a very simple typo generator for *request* and *async* was used. The algorithm made use of three edit operations: Insert operation, replace operation and delete operation. The implementation only processes the characters of the source string, without introducing characters outside of the source string (like the *fat-finger* distance does for instance). The algorithm generates all possible typos with edit distance one.

To compensate for the lack of characters that are close to the source characters on a QWERTY keyboard, typos obtained from three typo generation tools, which are available online, were considered. The list of all used typos for *request* and *async* can be found in the Appendix in section *Data for Algorithmically Generated Typos* on page 59. The algorithm used to generate the typos can be inspected in the code listing below. The algorithm is based on the *Levenshtein* distance.

```python
def generate_typos(s):
    """
    Generates typos with edit distance one for s.

    requires:
        A non empty string s.

    ensures:
        To return a set of all possible
        typos with edit distance one generated
        by the delete, replace, insert operation.
    """
    results = set()

    for i, char in enumerate(s):
        results.add(delete_op(s, i))
        for j, _ in enumerate(s):
            results.add(insert_op(s, char, j))
            results.add(replace_op(s, i, j))

    return results


def insert_op(s, c, i):
    """
    Inserts the char c at index i and
    returns the new string.

    requires:
        (len(s) > 0 && len(c) == 1
            && i in range(0, len(s)+1))
    ensures:
        (len(retval) + 1 == len(s))
    """
    assert len(s) > 0 and len(c) == 1\
        and i in range(0, len(s)+1)

    return s[:i] + c + s[i:]


def replace_op(s, i, j):
    """
    Replaces/substitutes the char at position i with the char
    at position j.

    requires:
        (len(s) > 0 && j in range(0, len(s))
            && i in range(0, len(s)))
    ensures:
        (len(retval) == len(s))
    """
    assert (len(s) > 0 and j in range(0, len(s)) \
                and i in range(0, len(s)))

    l = list(s)
    temp = l[i]
    l[i] = l[j]
    l[j] = temp
    return ''.join(l)

def delete_op(s, i):
    """
    Deletes the char at position i.

    requires:
        (len(s) > 0 && i in range(0, len(s)))

    ensures:
        (len(retval) == len(s))
    """
    assert (len(s) > 0 and i in range(0, len(s)))

    return s[:i] + s[i+1:]


if __name__ == '__main__':
    typo_names = ['async', 'request']

    for typo in typo_names:
        print(generate_typos(typo))
```

3.5 Collected Information

The logic implemented in the *notification program* essentially collects the following non personal information and sends it to the university web server in the parameters of a HTTP request:

- The typosquatted package name and the (assumed) correct name of the package. This information was hard-coded in the *notification program* before the package was distributed. Example: *coffe-script* and *coffee-script* (correct name).
- The package manager name and version that triggered the operation. The package manager name was also hard-coded, before the package was uploaded. The package manager version was retrieved dynamically. Example: *pip* and the outputs of the command `pip --version`
- The operating system and architecture of the host. Example: *Linux-3.14.48*
- Boolean flag, that indicates whether the code was run with administrative rights. Getting this information on Windows systems is not trivial and possibly error prone.
- The past command history of the current user that contains the package manager name as a substring. This information could only be retrieved from unixoid systems, because Windows systems do not store shell command history data. Example: Output of the shell command `grep "pip[23]? install" ~/.bash_history`
- A list of installed packages that were installed with the package manager.
- Hardware information of the host. Example: Outputs of `lspci` for linux. On OS X, the outputs of `system_profiler -detailLevel mini` were taken.

The complete database schema used on the web server, can be found in the Appendix in section *Database Layout* on page 64.

4 Results

4.1 Summary of Results

The empirical phase started on the 4th November 2015 and ended on the 21th December 2015. After some weeks, a second part of the empirical phase was launched to expand on potentially high download typo candidates that have been extrapolated from the command history data of previous installations (Compare to section *Analysis of Command History* on page 33). This second stage empirical phase was conducted between the 18th and 26th January 2016. In those two empirical phases, exactly 45334 HTTP requests by 17289 unique hosts (distinct IP addresses) were gathered. Packages for three different package managers, *PyPi (Python), rubygems.org (Ruby)* and *npmjs.com (Node.js – Javascript)* were uploaded and distributed. Most installations were received from *PyPi* with 15221 unique installations measured by distinct IP addresses. Then *rubygems.org* follows with 1631 distinct installations. *Npmjs.com* with 525 total unique IP addresses counted, had the smallest number of installations.

At least 43.6% of the 17289 unique IP addresses executed the *notification program* with administrative rights. From the 19603 distinct *interactions* (See definition in *Terminology* on page 5), 8614 machines used Linux as an operation system, 6174 used Windows and 4758 computers were running OS X. Only 57 hosts (or 0.29%) could not be mapped to one of these three major operating systems. These were mostly FreeBSD and Java operating systems (Or in rare instances, junk data that was submitted manually and thus not possible to parse). The number of *interactions* is higher than the number of unique IP addresses, because *interactions* are defined as the number of unique *(remote address, administrative rights, host os)* tuples. Therefore, if one single user installs typo packages with three different operating systems, this will count as three distinct *interactions*.

To get a first impression of the results, a plot that visualizes unique installation requests over time was created in Figure 4.1. The figure is divided into two subplots. The first plot displays the daily installation count of all packages over the timespan of the whole empirical phase. The second plot splits the installations in two disjoint sets: The number of installation caused by the top five packages as one plot (circles as markers) and all unique installations from the rest of all remaining 209 packages as another plot (data points with squares as markers). Furthermore, the graph also shows the number of unique installations that were conducted with administrative rights in the light version of the plotted lines. Figure 4.1 displays Saturdays (CET time) as vertical dotted lines. It is recognizable that installation numbers drop significantly on weekends.

The lack of installations during the second half of December 2015 and the first half of January 2016 can be explained with a pause in the empirical phase. No packages were distributed during this time and therefore no installations are recorded. The very low installation count during this phase (Visible in Figure 4.1) originates from a small number of installations that was observed, although no typo packages were uploaded. A possible explanation for this phenomenon is that existing packages were reinstalled or distributed over different channels than the primary source (repository).

4.2 Distribution of Installations per Package

If the number of unique installations for each uploaded package is accumulated and plotted (Figure 4.2), one can immediately see that the plot resembles a function with logarithmic distribution. Very few packages amount for almost all installations. 214 distinct packages amount to total 19721 unique installations. The first five packages only have already 12869 or 65.25% of all installations. The top ten packages together amount to 83.1% of all installations. Here the number of unique installations for all packages is larger than the number of unique IP addresses encountered in the empirical phase (19721 versus 17289). Some users installed more than one package (1.14 packages per user on average), therefore the aggregated sum of installations for all packages is higher than all encountered IP addresses. How can this extreme installation distribution visualized in Figure 4.2, with few very popular typo packages, be explained?

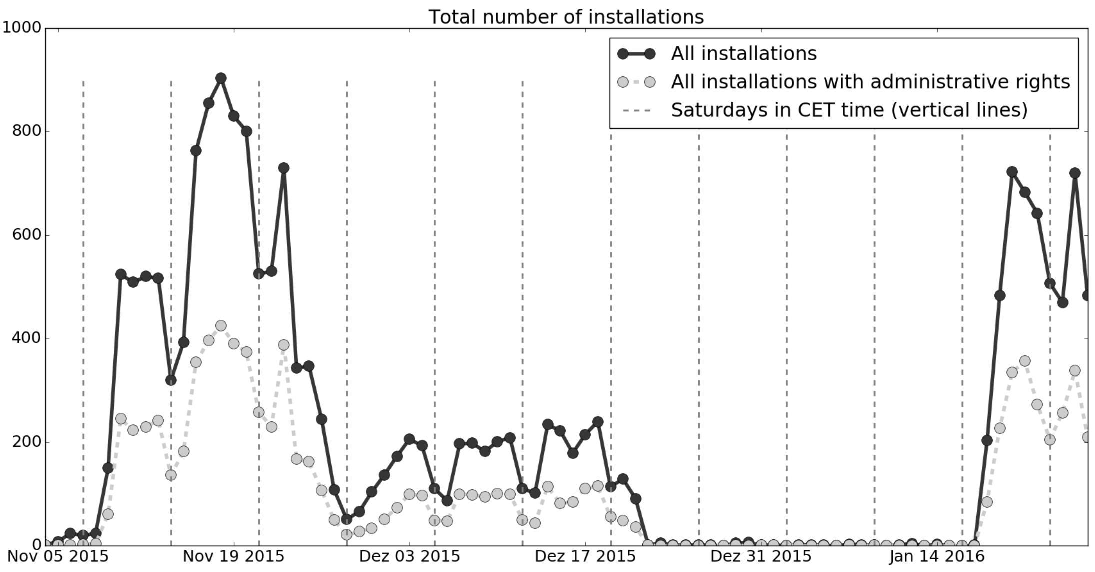
A possible answer for this phenomenon lies in the different package quality to cause installations. Some typo packages used in the empirical phase were originally Python2 standard library names (For example the package names *urllib2, urllib, cpickle, md5* and others) that did not continue to exist in the language successor Python3. Therefore, a lot of people tried to install the missing packages via the Python package manager, although the *PyPi* package repository does not host Python core packages. This immediately leads to the dangerous insight that it is possible in *PyPi* (and also in *rubygems.org*) to register packages that have the same name as standard library packages and thus to exploit the confusion people have with such package names.

Some statistical metrics of the installations received during the empirical phase are the following:

- 214 total different uploaded typo packages on three different package repositories
- 92 average installations per package
- The standard derivation of installations per package is 433 and thus relatively high
- The most installed package (*urllib2*) received 3929 unique installations in almost 2 weeks (284 average installations per day)
- The most installed package per day was *bs4* with 366 unique daily installations on average
- The least installed package had only one installation (Probably by a mirror or crawler)

The observation that different package names cause drastically different installation numbers motivates the clustering of uploaded package names in three different categories:

1. **Creative typo names** like *coffe-script* instead of *coffee-script*
2. **Stdlib typos or core package names** like *urllib2*
3. **Algorithmically determined typo names** like *req7est* instead of *request*



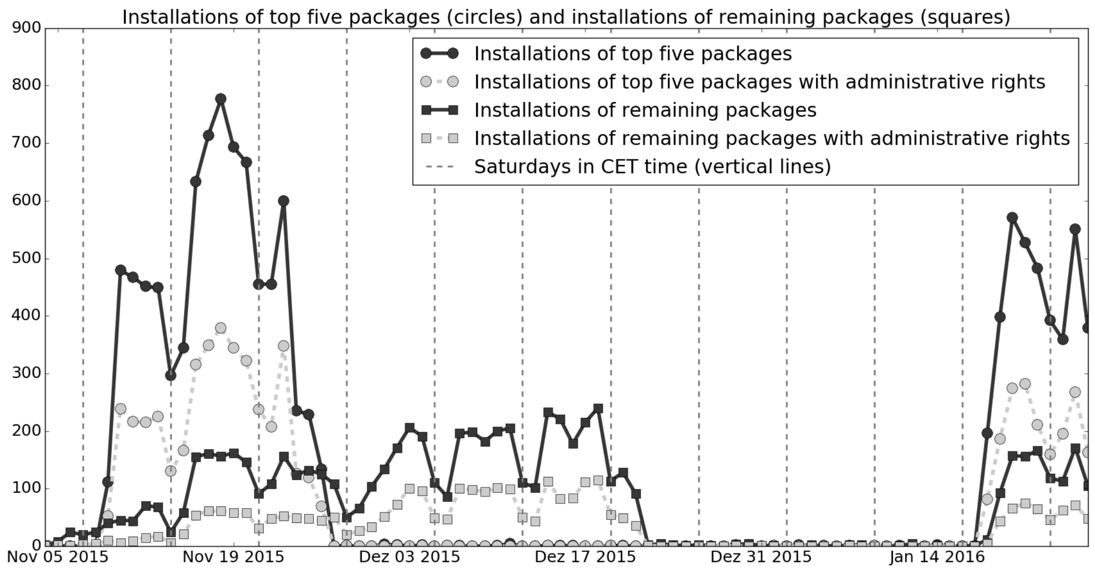


Figure 4.1: Visualization of installations in the empirical phase. Each point shows the installations on a certain day. The upper plot shows the total number of unique installations on each single day. The light dashed line are the installations with administrative rights. The bottom plot splits up installations in two sets: From the top five installed packages (circles as markers) and the rest of all packages (squares as markers). Light sub-graphs show the administrative ratio.

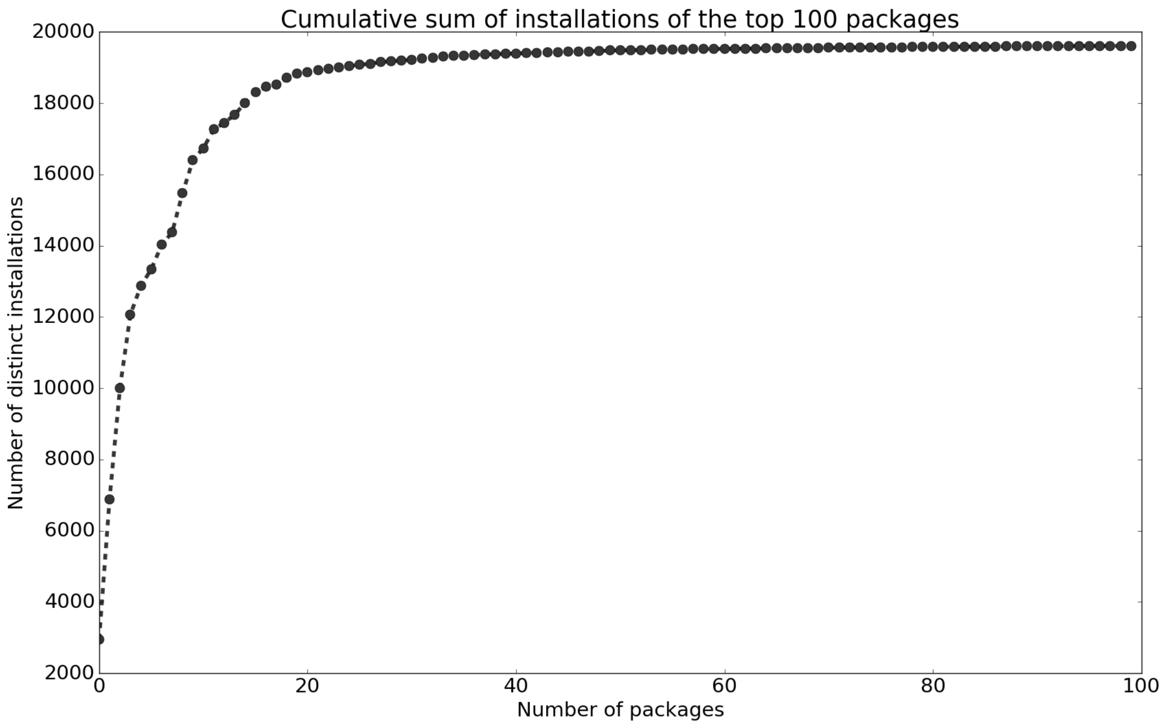


Figure 4.2: Cumulative sum of installations for the top 100 packages by number of installations. The cumulative sum converges rapidly to the sum of all installations (19721). Packages are sorted descendingly by average installations per day.

4.2.1 Creative Typo Names

The origin of *creative* typo names is the lack of peoples ability to distinguish the correct from the false version of the name. Alternatively, the cause for confusion might be a grammatically/semantically slightly different alteration of the name. Often only humans can create these kinds of typos, because its creation process requires an intuitive understanding of *What grammatical mistake is easy to make with the origin name*. The package name *coffee-script* perfectly illustrates this concept: A software (using the *Levenshtein* distance algorithm) would not be able to *understand* why *coffe-script (1)* is a better typo than *cofee-script (2)*, because both typos have an edit distance of one and both typo versions were obtained through a delete-operation in a repeated letter. For humans however, *(1)* intuitively resembles more to the real name than *(2)*. It could be argued that the 2-gram *ee* is much more rare than the 2-gram *ff* and thus people will commit the typo (1) more often. However, there are many such heuristics and no algorithm can consider them all. *Creative* typos cannot easily be created by algorithms, because only humans have a good understanding of *human misspellings*.

4.2.2 Stdlib Typos and Core Packages

Stdlib typos are no typos in the original sense, because there does not exist a correctly spelled counterpart in the package repository. Standard library names should either be reserved by the core language designers or impossible to register for normal users. Programming language users often get confused when using different major versions of languages (such as Python2 and Python3 or Ruby 1.9 and Ruby 2.1) by their inconsistencies in naming core packages. For instance, Python2 has a package called *urllib2*, but Python3 does not. To make things even more complex, there also exists a package named *urllib3* on *PyPi* only, but not in the core of Python2 or Python3. All of these naming similarities will entice people to reinstall missing functionality with package managers, because they are confused and they think installing it with the third party package manager will resolve their problems.

It is not easy to clearly differentiate between *creative* typos and *stdlib* typos. As a rule of thumb, all names that do already exist in the typo spelling and have not a typo in the original sense, are regarded as *stdlib* typos. All names that have a real typo in themselves and cannot easily be algorithmically generated are classified as *creative* typos.

4.2.3 Algorithmically Determined Typo Names

Algorithmically determined typo names are typos generated programmatically using various algorithms such as the *Levenshtein* edit distance. These algorithms have limited capabilities in generating high quality typo names, because it is an AI task to find package names that are prone to misspellings for humans. These algorithms cover the whole popularity distribution of typo names. They might be modified by heuristics or trained by artificial intelligence techniques. One example heuristic that would make generated typos more potent, is to prefer words with two identical letters next to each other. For example, the candidate word *coffee-script* would be more attractive than the word *coca-cola*.

4.2.4 Implications

The most important result in this section is the discovery of different classes of typo names. With this insight, one can see that *stdlib* typos had by far the most impact in terms of raw installation numbers. Then *creative* typo names follow by a wide margin. In fact, there is no package belonging to the *creative* or *algorithmic* category in the top ten packages (Compare to Table 4.2). Algorithmically determined typo names tail at the end of the ranking. The top packages sorted by number of average daily installations are illustrated in Table 4.2. As Table 4.2 shows, there are two different metrics to measure the installation count of the submitted typo packages: The public package statistics from the package repositories (ni_s) and the installation count based on the empirical data the *notification program* submitted (ni_e). In this thesis, whenever unique installations numbers are mentioned, the metric based on empirical data is meant (ni_e), except it is clearly stated that the download count from repository statistics are used (ni_s).

It is immediately recognizable from Table 4.2 that *stdlib* typos rank at the top and that *PyPi (pip)* packages were installed most. It should be noted that the installations of the top 20 packages together amount to 95.5% of all installations. In Table 4.1 and Figure 4.3 the installation count for all packages has been plotted such that the typo category distribution is visualized. *Stdlib* typos with 95.6% of all installations are installed much more than *creative* typos (2.3%) and *algorithmically* created typos (2.1%). Figure 4.3 shows clearly that *stdlib* packages are at the top of the installation count distribution, although there are much less *stdlib* packages than *algorithmically* created ones (26 *stdlib* packages and 167 *algorithmically* created packages).

Table 4.1: Installation distribution of the three different typo categories.

Typo Category	Number of Installations
All installations	19721 (100%)
Stdlib	18855 (95.6%)
Creative	459 (2.3%)
Algorithmical	407 (2.1%)

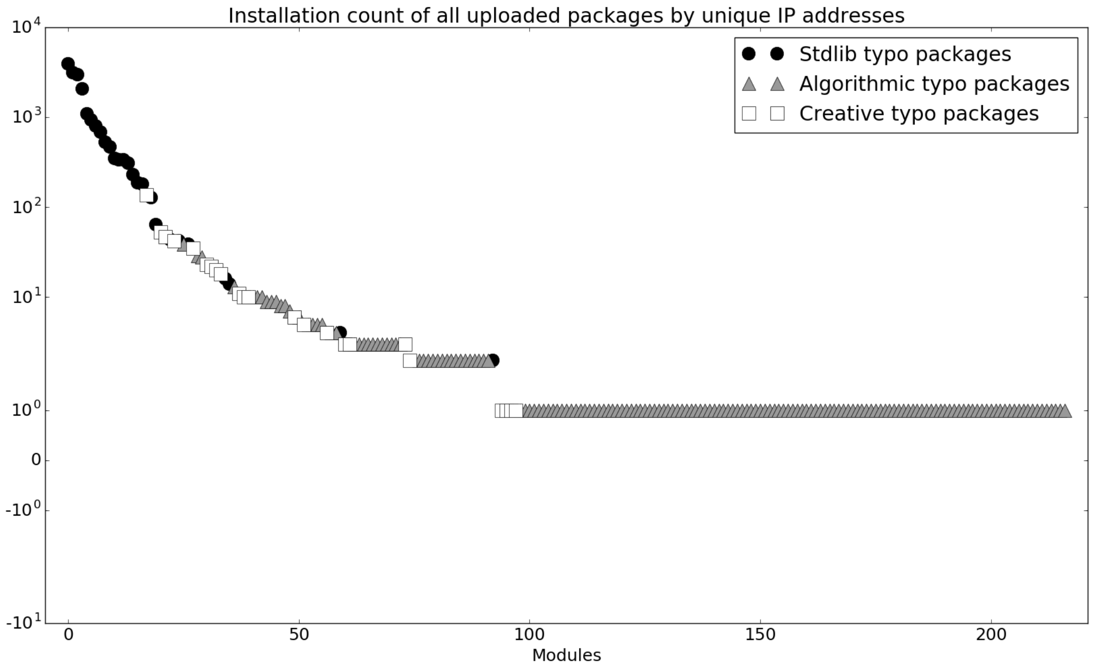


Figure 4.3: Plot of installations (ni_e) of all packages classified into the three different typo categories. The top packages belong to the *stdlib* typo category, *creative* typos follow and *algorithmically* created typos trail at the end of the popularity distribution. Note that the Y axis has logarithmic scale.

Other major factors of typo packages that affect the installation count, are the popularity of the correctly spelled version of the typo package and the quality of the typo (given by the propensity of humans to misspell the name). Due to the low sample size of uploaded packages, there is no systematic survey about how those two variables affect the installation rate of typo packages. However, it is strongly assumed that those two variables have a positive correlation with the installation count. *Stdlib* packages do not have a correctly spelled version of themselves, because they are already correctly spelled. These names should not be allowed to exist in the package repository. Therefore, it is unfair to compare *stdlib* packages to *creative* and *algorithmically* created typos, because *stdlib* names are no typos in the original sense.

Table 4.2: Table of installation information of the most downloaded packages. *pn* = Package name, *pm* = Package manager, *tc* = Typo Category, ni_e = Installation count based on empirical data, ni_s = Number of installations based on repository statistics, *aid* = Average installations per day, *mdi* = Maximum installations and minimum number of installations per day, *ut* = Upload time of package in days. Table is sorted descending by *aid*.

pn	*pm*	*tc*	ni_e	ni_s	*aid*	*mdi*	*ut*
bs4	pip	stdlib	2950	3959	366.26	442/290	8.05
urllib2	pip	stdlib	3929	4922	283.93	369/183	13.84
urllib	pip	stdlib	3132	4027	226.28	302/157	13.84
json	pip	stdlib	2058	3203	185.24	259/128	11.11
git	pip	stdlib	800	1211	99.12	146/60	8.07
python3	pip	stdlib	470	858	62.35	81/54	7.54
re	pip	stdlib	683	1330	61.38	75/52	11.13
scikit	pip	stdlib	348	728	46.56	60/42	7.47
cpickle	pip	stdlib	1098	1509	44.10	67/16	24.90
httplib	pip	stdlib	932	1421	37.44	59/20	24.90
math	pip	stdlib	336	770	30.22	43/19	11.12
yaml	gem	stdlib	526	1179	27.05	50/9	19.45
docker	pip	stdlib	182	540	22.57	31/9	8.06
csv	gem	stdlib	229	441	21.03	42/7	10.89
uri	gem	stdlib	334	1154	17.17	48/5	19.45
mkmf	gem	stdlib	310	534	15.93	31/6	19.46
coffe-script	npm	creative	138	177	12.66	6/1	10.90
hg	pip	stdlib	64	374	8.49	15/2	7.54
md5	pip	stdlib	187	537	7.53	15/1	24.84
base64	gem	stdlib	128	390	6.58	14/2	19.46
aysnc	npm	algorithmic	39	59	5.17	3/1	7.54
gruntcli	npm	creative	53	94	4.81	10/2	11.01
requst	npm	algorithmic	29	42	4.20	2/1	6.90
time	gem	stdlib	44	618	3.97	7/1	11.10
cherio	npm	creative	42	69	3.81	8/2	11.02
cofee-script	npm	creative	35	66	3.79	9/3	9.23
aync	npm	algorithmic	28	57	3.71	3/1	7.54
argparser	pip	creative	47	4031	3.40	7/2	13.83
setuptols	pip	creative	23	208	2.43	5/1	9.47
body_parser	npm	creative	22	47	2.06	5/1	10.70
docutil	pip	creative	20	203	2.01	3/1	9.94
find	gem	stdlib	39	269	2.00	4/1	19.45
asnyc	npm	algorithmic	13	37	1.72	6/1	7.54
sha	pip	stdlib	42	277	1.69	5/1	24.83
coffe-rails	gem	creative	18	189	1.65	3/1	10.89
reques	npm	algorithmic	10	48	1.45	2/1	6.90
ipaddr	gem	stdlib	16	183	1.44	4/1	11.09
reqest	npm	algorithmic	10	42	1.39	2/1	7.22
csselect	pip	creative	11	186	1.38	3/1	7.96
asyc	npm	algorithmic	10	38	1.33	2/1	7.54

4.3 Creating Typo Names Algorithmically

There were four different methods to create typo names for the two package names *request* and *async*. The first method was an own implementation of a distance algorithm as shown in section *Generation of Typosquatting Targets* on page 16. The other three sources were typo generation tools found in the Internet. The results of the empirical phase clearly show that there is no real additional value in using third party online tools (which advertise that they make use of statistical data from misspellings in order to create typo names).

If the success of each method is measured by the ratio of total distinct installations and the number of typos (Installation statistics for the different methodologies can be found in section *Data for Algorithmically Generated Typos* in the Appendix on page 59), the own implementation ranks first, closely followed by the tool from `www.digitalcoding.com/tools/typo-generator.html` (Accessed on 16th March 2016). It must be understood that this ranking has no real statistical value, because the sample size is too small: Only two typo names were used and only an average of three installations per generated typo package name was recorded.

4.4 Verification of Installation Count

An unique installation was defined as all unique 2-tuples (IP address, module name) of the data received during the empirical phase. By comparing these two metrics, it becomes clear that the installation count of the package repositories (ni_s) is higher than the number obtained by counting unique notification requests in the empirical data (ni_e) (Compare with Table 4.2).

As the number of installations grows, the gap between the two metrics tends to become smaller. This effect can be seen in Table 4.2: The two most installed packages, *urllib2* and *urllib*, both with relatively high absolute installation numbers and high average installation numbers, have a relatively small gap between the package repository count (ni_s) and empirical data count (ni_e) for installations. Packages from the same package manager with significantly fewer installations have much larger ni_s numbers than ni_e numbers.

This observation can be explained in the following way: Every new package is downloaded automatically by robots and crawlers. Other agents also download the raw files directly instead of using the package manager. The number ni_e however cannot count all the possible kinds of installations that ni_s includes. Other reasons for the smaller ni_e numbers are: Some firewalls block outgoing traffic and thus the *notification program* is unable to send its data back home. A further reason is the usage of NAT (Network Address Translation): Installations that originated from the same IP address cannot be distinguished clearly. In doubt, notification requests are counted only once. Furthermore, the package repositories count each single installation, even when several users download packages repeatedly and account for numerous installations in a short time.

If the *Pearson correlation coefficient* between ni_s and ni_e is computed for each single package manager, the following results in Table 4.3 are obtained. The correlation coefficients show that the download count on the package manager website (ni_s) and the installation count based on the obtained empirical data (ni_e) correspond closely in *npm* and *pip*: $Corr_{npm}(ni_s, ni_e) = 0.927$

Table 4.3: Pearson correlation coefficient for the installations counts of ni_s and ni_e and the ratio of the sum of all ni_s and ni_e installations (α).

Package manager	$Corr(ni_s, ni_e)$	$\alpha = \sum ni_e / \sum ni_s$
pip	0.859	0.555
gem	0.434	0.231
npm	0.927	0.421

and $Corr_{pip}(ni_s, ni_e) = 0.859$), but there seems to be no correlation between these numbers in *gem*, $Corr_{gem}(ni_s, ni_e) = 0.434$.

This observation can be explained by the installation numbers (See also Table 4.4) of some *gem* packages: *Time (44, 618), multijson (14, 1675), sprocketsrails (1, 277) and multi-xml (1, 283)*. The first number (ni_e) is much smaller than the second number (ni_s). This huge discrepancy between ni_s and ni_e of those packages is causing the small $Corr_{gem}$. As Table 4.2 shows, there are *gem* packages with much more balanced ni_s and ni_e ratios (For example the packages *yaml* and *csv*). Table 4.3 furthermore shows the fraction between the sum of all ni_e installations and ni_s installations (α). The metric α states: What percentage of the installation count as given by the package repository statistics (ni_s) can be backed up with obtained installations via the *notification program* (ni_e). In other words, α asks: How easy is it to achieve code execution in the package repositories? Python has the biggest number, 55.5% of all Python installations depicted in *PyPi* statistics were actually counted in the empirical data. Ruby installations have only a 23.1% coverage. A possible explanation for this phenomenon was already discussed in section *Prerequisites for Typosquatting Attacks*: In order for the *notification program* to run in Ruby packages, a native extension needs to be compiled on installation. Not every host has the appropriate compiler suite pre-installed. And some packages do not need any build stack, so there is no compiler suite and thus no code execution is achieved.

The exact reasons why some *rubygems.org* packages have a much smaller ni_e/ni_s ratio than Node.js or Python packages are not clear. However, it is strongly assumed that the cumbersome code execution method in Ruby is the culprit.

4.5 Anomalies in Measuring Installations

There were 11 typo packages which had a very low empirical installation count (ni_e), although their download count as published on the repositories (ni_s) indicate a much larger download activity. All packages that have a coverage of empirical installations less than 10% ($Cov = ni_e/ni_s < 0.1$) are included in Table 4.4. Especially the packages *argparser, multijson* and *time* have relatively large ni_s numbers. For those packages, it is not possible that mirrors and robots amount for the ni_s downloads, because other packages (Compare Table 4.2) have similar ni_s numbers, but much higher ni_e numbers. It is also remarkable that only Python and Ruby packages are affected, but Node.js does not have packages with small $Cov < 0.1$. Without having insight into the exact download statistics and the way the ni_s number is calculated, there is no transparent way to explain those anomalies.

Table 4.4: Tables with packages that have installation counts with a coverage of less than 10% ($Cov = ni_e/ni_s$). For the meaning of the single columns, compare to legend of Table 4.2. The table is sorted after descending ni_s.

pn	*pm*	*tc*	ni_e	ni_s	*Cov*	*aid*	*mdi*	*ut*
argparser	pip	creative	47	4031	0.01	3.4	7/2	13.83
multijson	gem	stdlib	14	1675	0.01	0.78	4/1	17.91
time	gem	stdlib	44	618	0.07	3.97	7/1	11.1
markup-safe	pip	creative	4	349	0.01	0.4	1/1	9.95
multi-xml	gem	creative	1	283	0	0.1	1/0	9.96
sprockets_rails	gem	creative	1	277	0	0.1	1/0	9.98
simpljson	pip	creative	10	187	0.05	0.98	2/1	10.19
csselect	pip	creative	11	186	0.06	1.38	3/1	7.96
ipaddr	gem	stdlib	16	183	0.09	1.44	4/1	11.09
simpeljson	pip	creative	3	183	0.02	0.32	1/1	9.44
simplejosn	pip	creative	10	178	0.06	1.14	3/1	8.8

4.6 Operating System Distribution

The *notification program* also sent a string with the host operating system to the web server. In the following section, a discussion of the operating system distribution among all installations follows.

As shown in Table 4.5, 44% of all installation belong to Linux systems. 31.5% of the installations were from Windows hosts and 24.2 % belong to OS X computers. There are 19603 total installations. This number is higher than the number of unique IP addresses counted (17289), because all unique (IP address, operating system, administrative rights) tuples are considered. For example, a user who installs a package first with administrative rights and then without, is counted twice, although she only used one IP address. Furthermore, this way of counting unique installations is also different to the obtained number (19721) in section *Distribution of Installations per Package* on page 20, because there, all unique (IP address, package name) tuples were counted.

Almost two thirds of all Linux installations were conducted with administrative rights (61.6%). With OS X and Windows it is the opposite: 68.1% of Windows installations and 73.9% of OS X installations were conducted with non administrative rights. However, these numbers might be not completely accurate because of the logic used in the *notification program*: While it is pretty straightforward to test for administrative rights on Linux hosts (By comparing the result of the POSIX function *getuid()* with zero), it is not trivial to determine the exact user rights on Windows and OS X (compare code lines 155-162 in section *Notification Program in Python* in the Appendix). Reason for this are the different necessary approaches with various Windows versions to determine the privileges (XP, Vista, Windows 7 or Windows 8). Therefore, the amount of Windows and OS X hosts with administrative rights could be much higher than the data in Table 4.5 indicates.

Table 4.5: Visualization of installations and their administrative right ratio. Installations are clustered into three major operating systems (Windows, Linux, OS X) and other operating systems (Solaris, Java OS, FreeBSD and others). The percentage number of the sum-rows refers to all 19603 installations and not to the total installations for this operating system.

Operating System	Administrative Rights	Number of Installations
All	Yes	8552 (43.6%)
All	No	11051 (56.4%)
Sum	Both	19603 (100%)
Linux	Yes	5308 (61.6%)
Linux	No	3306 (38.4%)
Sum Linux	Both	8614 (44.0%)
Windows	Yes	1974 (31.9%)
Windows	No	4200 (68.1%)
Sum Windows	Both	6174 (31.5%)
OS X	Yes	1238 (26.1%)
OS X	No	3520 (73.9%)
Sum OS X	Both	4758 (24.2%)
Other	Yes	32 (56.1%)
Other	No	25 (43.9%)
Sum Other	Both	57 (0.3%)

57 installations could not be assigned to one of the three aforementioned operating systems. They originated mostly from FreeBSD or Java operating systems. However, their administrative rights ratio is 56.1%, which is quite high, but not astonishing, because the POSIX function *getuid()* is implemented on most unixoid systems. The logic to identify the operating system from the submitted host OS field was implemented as a Python script in Figure 4.4.

4.7 Installations Over Time

By saving the timestamp of each single installation confirmation, it is possible to investigate the installation behavior over time of the users. For instance, it is strongly assumed that the installation rate drops on weekends, when the work force interrupts their work. Figure 4.5 proofs this assumption: The vertical dotted lines show Saturdays (in CET time) and a quite large decline (around 50% of installations) in installation numbers are visible around those markers. Only installations whose IP addresses belonged to the European continent have been used in this graph, in order to prevent blurring due to installations of different time zones.

Figure 4.4: Python code which illustrates the classification of the three major operating systems from the input of the *notification programs*.

```python
def get_os(host_os):
    """
    Tries to detect the operating system
    from the database field "host_os".

    requires:
        A non empty string in the parameter host_os.

    ensures:
        To return one of: Windows, Apple, Linux
        or Unknown if the operating
        system cannot be determined.
    """
    import re
    if re.search(r'[Ww]indows', host_os):
        return 'Windows'
    if 'mingw32' in host_os or 'win32' in host_os or\
         'CYGWIN_NT' in host_os:
        return 'Windows'
    elif re.search(r'[Ll]inux', host_os):
        return 'Linux'
    elif re.search(r'[Dd]arwin', host_os):
        return 'Apple'

    return 'Unknown'
```

4.8 IP Address Analysis

It is interesting to conduct IP intelligence on the collected data from the empirical phase. The geographical location of IP addresses can be determined using *geoip* databases and the associated hostnames may be found with reverse lookups. Conducting a reverse lookup might reveal which organizations/institutions commit typos and are affected by the attack.

4.8.1 IP Geolocation

The process of IP geolocation allows to map an IP address to a city or country. There are commercial databases that offer these mappings. One of them, the *MaxMind geoip database*, advertises a free database named *GeoLite2* with limited geolocation resolution precision [Max]. By mapping all IP addresses to countries, a table of countries with most installations is obtained as in Table 4.6. The United States have by far the most unique installations, followed by China. Then the top economies of Europe as well as Japan and India follow with roughly 1/8 of US installations. The global domestic product (GDP) in million Dollars and the GDP rank are included in Table 4.6, to illustrate the correlation of the number of installations with the economic production power. In most cases, the rank measured by number of unique installations and the GDP rank, seem to go hand in hand. Only Ireland does not follow this assumed relationship. Ireland has more unique installations as its GDP rank implies. The relatively large banking and software service sector of Ireland causes this skew in correlation.

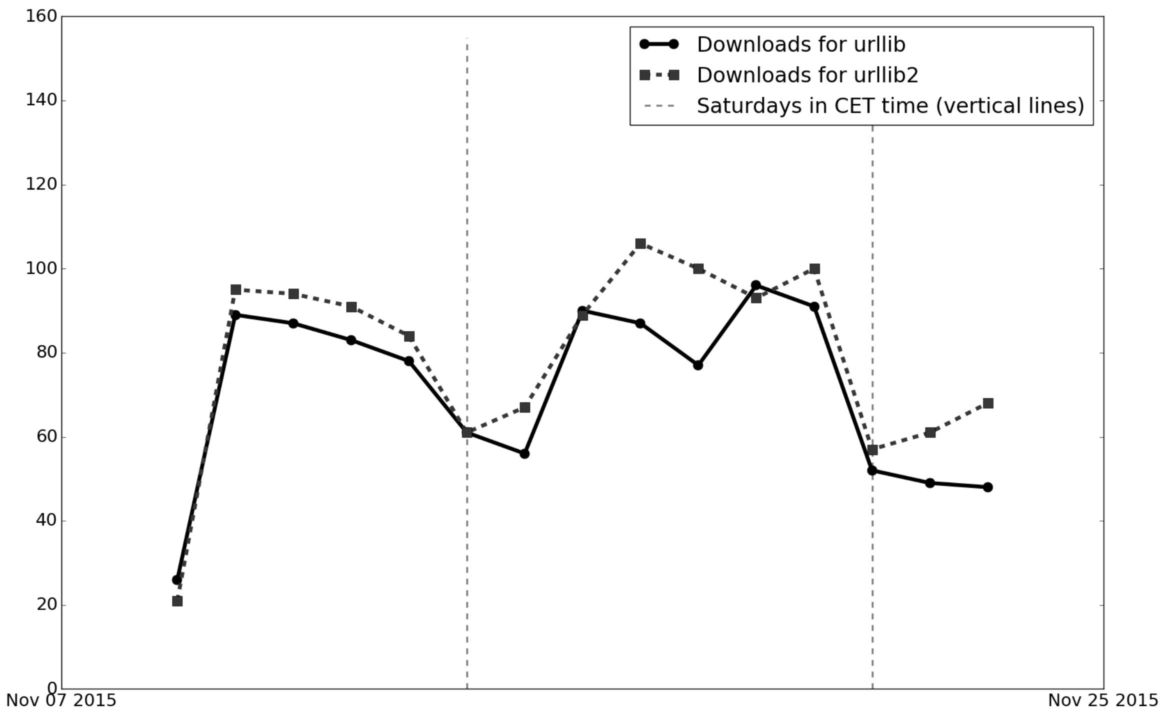


Figure 4.5: Visualization of weekly installations from European IP addresses for the package *urllib* and *urllib2*. It is recognizable that the installation rate drops on weekends (vertical dotted lines are Saturdays in CET time).

The Pearson correlation coefficient for the variable unique installations I and the GDP rank variable G is $corr(I,G) = 0.8259$ (GDP data from Worldbank, `http://data.worldbank.org/data-catalog/GDP-ranking-table`, accessed on 17th February 2016, Note: Taiwan is not considered to be a sovereign nation in the data of Worldbank). This means that the number of unique installations and the GDP rank have a high correlation. This relationship is not particularly surprising, but can serve as confirmation for the validity of the recorded data. If there was no high correlation, the validity of the gathered data during the empirical phase could be challenged.

4.8.2 Reverse Lookup

With all obtained IP addresses, it is possible to conduct a reverse lookup. A reverse lookup resolves an IP address to a host name, whereas a normal DNS lookup maps hostnames to IP addresses. Because an exhaustive search of all DNS records would need too much processing power, an extra *in-addr.arpa* domain was introduced that consists of a tree with *"reverse ordering of the numbers in the dotted-decimal notation of IP addresses"* [Tec]. The *in-addr.arpa* domain tree requires an additional resource record type named pointer (PTR) resource record. This resource record creates a mapping in the reverse lookup zone that typically corresponds to a named host (A) resource record for the DNS computer name of a host in its forward lookup zone [Tec]. The actual name resolution process is similar to the forward search, because the

Table 4.6: Number of installations per country and the GDP rank. The rank measured by the number of installations (I) and the GDP rank of these countries (G) correlate with a Pearson correlation coefficient of $corr(I,G) = 0.8259$.

	Country	Unique installations (I)	GDP rank (G)	GDP in Mio. $
1	United States	5810	1	17419000
2	China	2050	2	10354832
3	Germany	743	4	3868291
4	India	727	9	2048517
5	United Kingdom	725	5	2988893
6	Russia	605	10	1860598
7	Japan	579	3	4601461
8	France	469	6	2829192
9	Canada	432	11	1785387
10	Netherlands	314	17	879319
11	Republic of Korea	308	13	1410383
12	Brazil	283	7	2346076
13	Australia	254	12	1454675
14	Ireland	249	43	250814
15	Spain	223	14	1381342
16	Poland	213	23	544967
17	Ukraine	200	59	131805
18	Italy	192	8	2141161
19	Taiwan	177	no data	no data
20	Israel	156	37	305675
21	Sweden	147	21	571090
22	Singapore	144	36	307860
23	Switzerland	113	20	701037
24	Belgium	113	25	531547
25	Hong Kong	96	38	290896
26	Turkey	94	18	798429
27	Finland	83	41	272217
28	Austria	81	27	436888
29	Mexico	79	15	1294690
30	Norway	75	26	499817

in-addr.arpa is a plain domain tree. It needs to be understood that *"the configuration of pointer (PTR) resource records and reverse lookup zones for identifying hosts by reverse query is strictly an optional part of the DNS standard implementation"* [Tec]. Therefore, not all IP addresses gathered in the empirical phase can be mapped to a valid hostname.

The motivation behind this lookup process is that DNS names carry more information than numeric IP addresses. Often, the hostname makes it possible to find out which organization or company owns the IP address. The top level domain often carries useful information, such as whether the organization is educational (.edu), from a government institution (.gov - US only), or even has origin in military organizations (.mil - US only).

Properties of the looked up domains are illustrated in Table 4.7. The reverse lookup revealed that 5950 IP addresses did not yield a valid hostname. Some Internet service providers allocate and deallocate their IP addresses dynamically and thus do not have a reverse lookup entry. It is remarkable that 1249 IP addresses originated from mostly *ec2 amazon* web services instances. There are 394 *.edu* domains from many different universities around the world. This relatively large number of hosts can be explained by the wide usage of Python, Node.js and Ruby in the educational sector. There were also 23 *.gov* domains from governmental institutions of the United States. This number is highly alarming, because taking over hosts in U.S. research laboratories and governmental institutions may have potentially disastrous consequences for them.

From the 11339 hosts that returned a valid hostname, there were also two hosts with a *.mil* top level domain. This shows that even very security aware military institutions fall victim to typosquatting attacks.

4.9 Properties of Successfully Attacked Hosts

In this section, characteristics of hosts that transmitted data back from the *notification program* are going to be discussed. This will include information like whether a host can be considered a server or work station. The *notification program* was modified during the empirical phase to send additional meta data:

1. The bash history of all package manager commands.
2. A list of installed packages on the host.
3. System information identifying the hardware properties and settings of the host.

It is important to understand that this additional information was only raised in a later stage of the empirical phase. So only 1454 installations included command history information. And only 4205 installations sent back hardware information. This indicates that hosts were much more likely to send valid data for hardware information than for command history, because only Unix systems store command history, whereas Windows systems do not.

One interesting question that can be answered with the command history is: Are there other typos committed by the user in the past? Is it possible to create malware that generates new typos based on the past command history of infected hosts and finds new lucrative typos, which in turn are registered again, to gain even more typo installations?

4.9.1 Analysis of Command History

In the middle of the empirical phase, the Python and Node.js *notification programs* were modified to send the outputs of the past command history of Linux users. Only commands were included that had the *pip* or *npm* substring in its command. For example, the modified Python *notification program* sent the outputs of the following command to the university web server: `cat ~/.bash_history | grep -E "pip[23]? install"`.

This collected data makes it possible, to inspect some of the packages which infected users installed in the past. Because the command history file has a limited size, only the newest entries

Table 4.7: Distribution of top level domains and substrings in domains of the hosts which executed the *notification program*. The table is to be read as follows (example): 394 of total 17289 hosts (from which 11339 successfully resolved to a hostname) had a *.edu* top level domain.

Criteria	Number of hosts
All hosts	17289
Lookup failed	5950 (34.4%)
Lookup successful	11339 (65.6%)
.mil	2
.gov	23
.edu	394
.com	3122
.net	3073
.org	46
.de	505
.us	12
.ch	74
.uk	142
.ru	324
.pl	170
.it	153
.es	50
.nl	166
.fr	190
comcast.net	727
amazonaws.com	1249

are shown. The *.bash_history* file is only available on unixoid systems, so there is no command history data for Windows systems. 1454 unique installations on Linux and OS X systems with command history data were received during the empirical phase. Analyzing this amount of data is very work heavy and doing it all automatically is not intelligent, because it is easy to miss hidden patterns in the data.

Unfortunately, it cannot be verified that the package name of the last command in the command history is the name of the *notification program*, because this command will only be added to the history when the process already exited. This would have confirmed that the correct command history is processed. However, the following questions can be answered by analyzing the command history data:

1. Are there recurring common typos among independent users?

2. What are some of the most observed patterns that lead to typos?
3. Can lucrative typos (in an attacking sense) be found by analyzing the past command history?

In order to answer those questions, a way to find all real typos in the command history was needed. Luckily, the Python package repository provides a complete list of all registered and publicly available packages (`https://pypi.python.org/simple/`, Accessed on 17th March 2016). So all package names that appear somewhere in a install command in the history and which cannot be located in this list, are real typos.

The most common typos can be seen in Table 4.8. The table shows the number of times unique users committed typos. The table is to be read as follows: 90 from total 1454 unique users with a command history file, tried at least once in their command history to install a module named git, but did not succeed, because there is no such module in the *PyPi* repository.

All rows with cursive typo names in the table originate from typo packages that were used and registered during the empirical phase. The remaining rows however, have not been used in this thesis.

The analysis reveals a concerning result: By mining the command history for typos, several new high class typo candidates, which promise large numbers of installations, have been located. Especially the module names *git* (misspelled in 90 distinct hosts), *scikit* (89 unique misspellings) and *bs4* (31 hits) seem to be mistyped frequently among independent users. By registering them, lots of installations seem to be guaranteed.

By considering that alone the *urllib2* package caused an average of 284 unique daily installations (Compare with Table 4.2), even more installations are expected by the package names *git* and *scikit*. Furthermore, all cursive typo names in Table 4.8 are already highly overrepresented, because the command history data originates from installations *of these names*. Therefore, it is much more likely that these users tried the same installation already in some time in the past.

Another insight can be deduced from Table 4.8: Almost all typos are no real typos, but seem to emerge through misconceptions of users who assume that these names must exist in the *PyPi* package repository, because they exist somewhere else or are famous software names (like *git*). Other typos, like *bs4*, emerge because the software is imported as *bs4*, but installed with the identifier *BeautifulSoup*. Therefore, those names can be considered *stdlib* typos.

There was only one way to find out whether the mined typo candidates produce lots of installations: From the 18th January 2016, some of these packages that promised big popularity, were registered and uploaded on the Python package repository. The following six typos were chosen from Table 4.8: *git, scikit, bs4, python3, docker, hg*.

After one week, those packages were removed again. The upload of the packages was shortly interrupted, because the *PyPi* administrators removed those packages for approximately 12 hours. Therefore, the average installations per day (*aid*) and the upload time in days (*ut*) in Table 4.2 for these six typo names should be higher and subsequently also the absolute number of installations (ni_e and ni_s). In Table 4.9, the number of installations that were counted (successful notify request sent to our web server) can be seen and the number of installations as provided by the *PyPi* package repository statistics. The obtained data strongly indicates that the mined data was not accurate in forecasting the *typo activity* behind certain packages, but was quite good in pointing out high download candidates in general. The typo name *bs4* was much more

Table 4.8: Real typos committed in the command history of 1454 distinct hosts. Typo names in cursive have been used in the first phase of the empircal experiment. Bold names have been chosen in the second stage empirical phase. All other names are potential typo candidates.

Typo name in command history	Unique number of occurrences
git	90
scikit	89
urllib2	52
urllib	33
ipython[notebook]	32
bs4	31
docker	28
json	28
protobug	24
ipython[all]	23
python3	23
hg	17
requests[security]	15
cpickle	13
tkinter	12
pickle	11
opencv	10
pyqt	10
stringio	10
sqlite	10
libxml2	9
youtube	9
hdf5	9
tensorflow	9
httplib	9
sqlite3	9
collections	8
openssl	8
ipython3	8

requested that the mined data suggested (368 average installations a day, the most successful typo package as Table 4.2 shows). *Scikit* was half as much downloaded as *git*, although the minded data suggested nearly the same popularity.

To draw a conclusion over the predictive quality of the mined command history data: The occurrence ranking in the command history does not correlate strongly with the average (and total) installation ranking achieved in the subsequent empirical phase. The mined data clearly reveals very potent typo candidates. However, only six samples are not enough data points to make reliable statistical conclusions.

Table 4.9: Table with installation counts for packages that were chosen according to frequent misspellings in command history data. Installations counted in the empirical data (ni_e) and the installation count published in the *PyPi* repository statistics (ni_s) are listed for the six packages.

Package name	Number of unique installations (ni_e)	Repo installation count (ni_s)
bs4	2949	3834
git	800	1211
python3	470	851
scikit	348	726
docker	182	525
hg	64	64

4.10 Reactions After Infection

It is interesting to investigate the behavior of *typo victims* after they unintentionally installed the typo package. The following different reaction types were considered.

4.10.1 Post Infection Visitation Rate

The *post infection visitation rate* is defined as the number of users who visited the URL with information to the typosquatting experiment, which was printed after the *notification program* was executed (The URL printed was: `http://svs-repo.informatik.uni-hamburg.de`).

Surprisingly the number of users who visited the link outputted by the *notification program* to the terminal is very low: Only 59 IP addresses out of 17289 total encountered unique installations requested the information page about the typosquatting experiment on *index.html*. The logfiles were examined for log entries after the timestamp at which the *notification program* submitted its HTTP request. To locate potential log entries, the same IP address that submitted the notify request was used in the search. This might not be to accurate, since users could have visited the link after they changed the IP address. Those users who visited the information site with the same IP address as they committed the installation, have installed the typo package on the same computer from which they visited the URL. It is assumed that most people do not install packages to the same computer where they also browser the web (for example they install *PyPi* packages to a development computer over *SSH*).

A quick search through the logfile confirms that only 1265 of the total 61180 lines in the logfiles belong to GET requests that target the *index.html* of the web server (and thus the link printed in the warning message). And almost all of the 1265 logfile entries that target the *index.html* file belong either to web crawlers or other visitors not related to the thesis (such as scanners looking for vulnerabilities). The search needle used for the logfile was `GET / ` (with a leading whitespace to not match partial paths). Another explanation for the low *post infection visitation rate* is the defective *notification program*: As in section *Direct Mail Contact* described, the URL

Table 4.10: Time in minutes before users visited the *index.html* URL (that was printed upon typo package download) with information to the typosquatting experiment after they installed a typo package.

Reaction time in minutes	Number of visits
$x > 1$	16 (27.1%)
$1 \leq x < 5$	13 (22.0%)
$5 \leq x < 20$	7 (11.9%)
$20 \leq x < 60$	4 (6.8%)
$60 \leq x < 180$	3 (5.1%)
$180 \leq x$	16 (27.1%)
Total visitors	59 (100%)

with information to the typosquatting experiment was not outputted correctly in some Python packages with new *pip* versions.

If the time in minutes after the execution of the *notification program* until the visit to the info page of these 59 users is considered, Table 4.10 is obtained. As the table shows, 16 out of 59 visitors looked up the info page within one minute after they installed the typo package. Another 13 users needed between one to five minutes before they browsed to the info page. 16 users looked up the info page after three hours or more time had already passed. Because the sample size is relatively small, the data does not have much statistical relevance. The table shows that around half of the users who reacted looked up the info page within five minutes. This time interval has another meaning: It shows how much time passed before the typo victims actually found out about having installed a potentially malicious package. This *window of opportunity* for the attacker demonstrates how long the attack went unnoticed. A real life attack however would surely not notice the victim that they just got hacked - They would not even realize that they committed a typo and that they installed malware, because it is trivial for the malware to simulate a correct package installation, modify the terminal output on the fly (to hide the typo in the install command) and to pretend that there never has been a typo installation. Therefore, the *window of opportunity* has not much meaningfulness.

4.10.2 Counter Attacks Upon Installation

A counter attack is for example a *SQL Injection* attack or attempted admin logins into the server backend by an IP address that originated first from the *notification programs*. Such behavior emerges through the annoyance that typo packages depict to these users. The manual or automatic submission of HTTP requests to the web server in order to distort the survey results belongs also to this category.

Like all log files of publicly reachable servers, the web server log files contained lots of random attacks and HTTP requests that targeted paths which are known to be sensitive (like the *phpMyAdmin* setup file). It is considered to be a targeted attack, if the attacker uses an URI that can only be known, when the message output of the *notification program* has been inspected. There was one instance of a *SQL Injection* attack against the web application. The attacker probed for a possible vulnerability by sending typical SQL special characters in HTTP parameters (like single quotes), that would cause an error message if the application had been vulnerable.

4.10.3 Repeated Downloads of the Same Typo Package

Whenever a user downloads a typo package and executes the *notification program*, a GET request with the hosts properties is send. When a user installs the package more than once, one can see those repeated installations because a timestamp is assigned to each request. This measurement is a good indication of how well users react to unexpected behavior. If a user installs a package several times in short succession, it may be an indication that she is inattentive or does not understand that she downloaded a typo package without the expected functionality.

Another interesting observation is the repeated unsuccessful installation of the same typo packages again and again. For instance, Table 4.11 shows the hosts with the top interactions. From the time delay between single requests, it might be deduced whether the requests were fired automatically. Some of the hosts with lots of requests in a consecutive time span might be automatic build or deploy tools which were configured with a typo name instead of the correct name. Another possibility is that an attacker tried to inject incorrect data and created a script which would repeatedly submit incorrect installations.

4.10.4 Direct Mail Contact

Everyone that followed the link outputted by the warning message upon installation, would see a explanation of the thesis and its objectives and a list of the data is collected. There are also several possibilities to get in contact with the author and supervisors of the thesis (the mail address and a link to the website of the faculty was put on the *index.html* file of the university web server).

On the 26th January 2016, Michal Jaworski send a mail with suggestions on how to print the warning message which was outputted in the *notification program*. He said that the warning message output will be swallowed in newer version of *pip*, because *"pip runs package's setup.py script in separate process using Popen and captures all the output. If package installation fails then this output is never displayed"*. Michal Jaworski suggested to throw an exception with the warning message, because *pip* will print the failing logic to the standard output stream. In tests of the *notification program*, the thesis author never experienced the lack of printing of the warning message. It must be said that older versions of *pip* were used in said tests. The possibility of the package installation to fail is rather slim and therefore also the chance that an installing user will not see the warning message.

On the 19th January 2016, Robert Kern, a software developer and user of the Python package index *PyPi*, wrote a mail during the second stage of the empirical phase (when packages based on the command history of the previous empirical phase were uploaded). He asked to *"put an informative description into your PyPI packages so that people browsing PyPI or reading its feed understand that you are not a malicious agent."*. After answering him that his suggestion are going to be followed, the thesis author got in contact with the *PyPi* administrators and send an explanatory email with motivations and research goals to the *PyPi* package repository administrator and developer Donald Stufft.

Table 4.11: Top interactions of certain hosts. It is possible to guess the nature of the installing host by observing how many times and in which intervals they installed the typo package. IP addresses have been anonymized by removing the first two octets.

IP address	Number of GET requests
..10.11	805
..240.122	590
..156.118	206
..127.129	176
..158.85	140
..237.10	94
..221.5	80
..161.1	74
..115.17	73
..11.200	72
..7.20	64
..155.161	61
..64.218	59
..133.193	53
..126.98	52
..127.186	52
..129.111	50
..115.169	46
..217.58	43
..227.100	42
..7.53	41
..49.253	40
..81.23	37
..134.197	34
..227.9	34
..33.36	34

First, Mr. Stufft was not very pleased with this research and ordered to remove the functionality that captures hardware information, the filtered past command history and the list of installed packages. He reasoned that this information might include sensitive personal information. Towards the collection of past command history, he wrote: *"I don't think you should be collecting "The past command history that involved the package manager". That can contain highly sensitive information (such as URLs to custom indexes, including passwords to allow accessing said URLs)."* He similarly argued to not capture hardware information and the list of installed packages. After removing the controversial logic and promising to Mr. Stufft to send him the results of the thesis, the packages were re-uploaded and the second stage of the empirical phase was ended shorty after. This incident was also the reason for the interruption mentioned in section *Analysis of Command History* on page 33.

4.10.5 Public Reactions

Users of the repository who came in contact with the thesis often created blog posts, mailing lists replies or forums entries by reporting a possible threat or posing question about this bachelor thesis.

Back in February 2015, when the idea of *typosquatting package managers* was born, the reaction to the typo packages and its code was very quickly. The reaction to the empirical phase conducted for this thesis was much weaker and not so prompt as experienced before (Compare to section *Python – Pypi.python.org* on page 46).

A support ticket question in the repository for the Python package manager (`http://sourceforge.net/p/pypi/support-requests/571/`, accessed on 17th February 2016) was created by Emil Sauer Lynge on 18th December 2015. In this support ticket, he described the package as spyware and complained that he did not have any possibility in denying the participation in the study. He installed a *PyPi* package named *cpickle* and mentioned that *"After moving to python 3, some might not be aware that cpickle has been renamed to pickle."*. This statement confirms the main findings in this thesis: High profile typo candidates emerge through confusions between naming standards in major programming language versions.

Another incident report was posted on Pythons *PyPi* development git repository on *bitbucket.org* by Adrian Klaver on 19th January 2016. In this issue, Adrian Klaver described the uploaded typo package *docker* which was distributed during the second stage empirical phase as spyware and that the package seems benign on the surface, but not healthy to leave it in the package index (`https://bitbucket.org/pypa/pypi/issues/379/spyware-packages`, accessed on 14th March 2016). No public answers to this support tickets were observed.

5 Practical Implications

Having discussed and described the type of attack and the obtained results, it is time to think about the implications of typosquatting attacks against package managers. A question that immediately emerges, is the following: Is it possible to fend off typosquatting attacks successfully?

In the following sections, generic defenses which package repositories could employ to mitigate the risk of being attacked by typosquatters, will be discussed. First, a simple attacker model is defined which describes the kinds of typosquatting attacks, that the system should be able to withstand. In the end of this chapter, algorithms (in pseudocode) are going to be implemented in the open source version of the Python package repository, to test the feasibility of the proposed defenses.

5.1 Basic Attacker Model

In this context, it often makes sense to define an *attacker model*, which precisely declares the strongest possible attacker against which the system's defenses do still hold. It is important to understand that the proposed defenses do not prevent all installations caused by typo packages. They will merely prevent a great share of installations, such as the threat caused by lucrative typo names (Like the *stdlib* typo candidates which were mined from the command history data, compare section *Analysis of Command History* on page 33). The current package upload policy of programming language package repositories makes it nearly impossible to create defenses that protect against all typo installations (Without restricting open package registrations). Therefore, the goal of the proposed defenses is to drastically reduce the threat of typosquatting attacks, not to completely eradicate it. After all, complete security cannot be achieved, it often suffices to make the life for attackers significantly harder. Well equipped adversaries (like offensive governmental cybersecurity task forces) favor other, more direct, attack vectors in any case.

The proposed *attacker model* describes the roles (user, programmer, administrator, ...), the propagation of the attacker (in the enterprise, somewhere in the Internet, ...), behavior of the attacker (active, passive, observing, modifying) and the computing power of the attacker (limited or unlimited). If a package repository is equipped with all defensive strategies described in this chapter, then the system protects against an attacker, who must not be involved with package repositories (no insider knowledge), but has public Internet access. She furthermore has basic knowledge about computer science and can develop simple applications. She is qualified to use open source malware and create arbitrary typosquatted packages and distribute them in repositories. Therefore, her behavior can be modifying, but not as far as she is altering the internal functioning of the repository server itself (Other than simply registering typo packages). In addition, the attacker has limited financial assets and computing power.

5.2 Prohibit Core Package Names

This work has shown that most typo installations are caused by users trying to reinstall standard library names with the package manager. So it is mandatory for every package repository to completely prevent every module name to be registered by third party users, that has been part of the core language at some point in time. Users sometimes stop using a programming language for several years, only to come back and try to use deprecated and removed core libraries. The first thing they will try to do, is to restore the known behavior by reinstalling the missing functionality with a package manager.

5.3 Disallow Famous (Software) Names

Programmers do need to know many different software names and their field of applications. This can be confusing in the beginning. So it might easily happen that programmers think that there must be a Python module on *PyPi* with the name *git*, because *git* is such a widely used software component. This entices programmers to try to install the package *git*. Package management administrators should prevent normal users from registering such names. This defense is hard to implement correctly, because it is not an easy task to judge whether the registration of a popular software name was malicious or not. After all, it is still possible that the original corporation behind a name wants to develop a application for it.

5.4 Reduce the Character Set in Package Names

Obviously the most basic defense mechanism against typosquatting is to reduce the character set for packages names drastically. Ideally, there should only be lower case alphabetical characters allowed and the underscore (or the hyphen – bot not both).

For example, if you allow both the hyphen and the underscore in package names, users tend to interchange such characters, which leads to possible new typo variants. The same holds for case sensitive package names: Was the package name written in *CamelCase* or all *lowercase*?

All such special characters which link words together are prone to be remembered incorrectly. So it is extremely important to only allow one *linking character* between words.

Furthermore, the reduction to lowercase (or uppercase) letters is crucial. If package names are case sensitive, typo packages may easily be generated by just switching the case of some characters.

5.5 Introduce Additional Namespaces Into Package Names

The popular open source repository *github.com* identifies packages by two attributes [Git]: The repository name and the authors/organizations name. Identifying a package by two items makes it much harder to unintentionally misspell one identifier. Some modern package repositories like *bower.io* and *packagist.org* already make use of this additional namespace. Therefore, it is much more secure, if a package is named *ntschacher/GoogleScraper* instead of just *GoogleScraper*.

The reason is: If the package name is misspelled and not the author name, this will not have any consequences, because the typo version cannot be registered in this namespace, since this author name is already reserved. However, it could be argued that it is possible to typosquat on author names and to register accounts with typosquatted author names. This might be a valid concern. Because package names are much longer with two attributes, it is more likely that users will copy and paste the package name instead of remembering it.

5.6 Prevent Direct Code Execution on Installations

One can differentiate between consciously provided configuration logic that allows package authors to define hooks for certain events on package installation (post/pre-install hooks like in *npmjs.com* [com]) and package managers that intentionally prevent code from being executed at installation time (like the package manager *bower* from *bower.io* [Sc13]).

It is paramount to absolutely disallow any code from being executed when packages are installed. Code needs to be trusted. And in the case when everybody may register packages that are downloadable by everyone, code should not be executed automatically.

Even if there is no possibility to execute code during the installation, the malicious payload might be executed when the typo package is actually used by the programmer. There is no mean to prevent this (absolutely intended) way of code execution. In order to use the library, the installing user needs to misspell the package name a second time (at import time), which is a bigger hurdle than to grant code execution rights to the package installation process in the first time.

5.7 Generate a List of Potential Typo Candidates

The idea is to mark certain package names as suspicious based on their edit distance to existing, legitimate package names and to disallow them to be registered. Each package repository provider knows their most downloaded packages. One can generate all typo variations with the *Levenshtein* algorithm to compile a list of potential typo candidates for typosquatters. Disadvantages are, that some combinations might actually be legitimate names and that the registration process is made unnecessarily cumbersome and possibly prevents natural development of the package repository.

In the Appendix in section *Finding Existing Typo Packages* on page 63, there is a Python script which generates 14605 typos for the top packages (ordered by number of installations) of the package repositories *PyPi, npmjs.com* and *rubygems.org* and checks whether the generated candidate typos have already been registered. The exact same approach can be used to create such typo names on the server side and to check if they are used on registrations.

5.8 Use Concepts of Established Package Managers

The programming language package managers *PyPi, rubygems.org* and *npmjs.com* are relatively new package managers. It is interesting to analyze, in which way other package managers and their repositories differentiate in fending off malicious packages.

5.8.1 Debian

The *Debian* project uses *dpkg* as a package manager and *APT* as a frontend. To upload packages to the official *Debian* distribution, one must become an approved member of the *Debian* project (*Debian Developer*) [Lar]. Most programmers do not have the permission to upload packages to the *Debian* distribution, instead they contact a *Debian Developer* through a *sponsorship* process. In this process, a mentor uploads and checks the package for technical correctness [Lar]. Alternatively, one can try to become a *Debian Maintainer* (with limited upload permissions), but for this a *Debian Developer* must advocate for him and gain his trust. This makes unsupervised uploading to the official *Debian* repository impossible. During the process of becoming a *Debian Developer*, several of the *Debian* team members need to vouch for the applicant. Furthermore, she must have contributed to the *Debian* distribution for at least 6 months [Lar]. Thus a typosquatting attack on *Debian* does not seem feasible.

5.8.2 Arch Linux

Arch Linux, a more difficult to learn Linux distribution, uses *pacman* as a package manager. There are two different repositories: The *AUR (Arch User Repository)* and the *community repository* [arc]. When packages in the *AUR*, where everybody can submit packages, receive enough attention and support from a *Trusted User*, they are moved in the official *community repository* [arc]. This means that it is not possible to smuggle in typo packages into the official repositories. Packages from the *AUR* must be manually downloaded and installed, which prevents possible typo attacks. This hybrid approach is very effective against typosquatting attacks and has the advantage that anybody can submit packages without undergoing a strict qualification and selection procedure as it is the case in the *Debian* community.

Conclusion

Established package managers for major Linux distributions only grant package upload permission to trusted contributers. Anonymous third party submitters are not allowed to participate. By using this concept in programming language package managers, malicious packages will be prevented from being uploaded in most cases. When using a hybrid package manager like *Arch Linux*, the convenience of uploading packages will be kept, while the official repository is secured against typosquatting attacks.

5.9 Defense Mechanisms in Existing Package Repositories

Some package managers already prevent the most notorious attack vectors, whereas others neglect even basic security measures. The following section will discuss, whether the surveyed package repositories regard the defense mechanisms introduced in the previous chapter.

5.9.1 Node – Npmjs.com

The package repository on *npmjs.com* explicitly prevents standard library package names from being used at registration. When names identical to core packages (like *os*) were tried to upload, the submission was rejected with the error message: `npm ERR! os is a core module name:os`

The security team on *npmjs.com* actively monitors new package uploads for malicious code [Bal15]. The company *liftsecurity* is responsible for security threats on *npmjs.com*, since the *npmjs.com* team encountered a severe bug [Vos14]. It seems that they are very well aware about the threats that pre/post-install hooks introduce [Bal15]. In a blog post publication from *liftsecurity*, they analyzed a malicious module with a single command (`rm -rf`) that would delete all files on the targeted system. Even though the module was removed within two hours, *liftsecurity* states that *"You're responsible for what you require"* and that users should inspect the source before they *"npm install"* it [Bal15]. They also mention that pre/post-install hooks can be disabled by setting the flag *–ignore-scripts* in *npm* commands [Bal15]. Although each advice is generally correct, not many users will abide by these instructions, since normal users in general value convenience more than security.

Furthermore, *liftsecurity* analyzed the 404 log files of the *npmjs.com* package repository server and concluded that typo errors happen at a high scale on *npmjs.com* [Bal15]. With this analysis of the web server logfiles, they did exactly what is proposed in the next section of this thesis, to be an optimal defense against typosquatting package managers.

As a result of the observation that almost all new submitted packages during the empirical phase were downloaded by a single host immediately after registration, it was concluded that this could be a mirror server or some kind of automatic tool that scans the code for security defects.

5.9.2 Python – Pypi.python.org

The Python package repository seems to have a much worse security infrastructure than *nodejs.com* or *rubygems.org*. There is no functionality that prevents *stdlib* packages from being submitted.

During the empirical phase, it could be observed that single, notoriously easy to commit typos are blacklisted from being used in the registration process. For example, it was tried to use the typo name *setuptool* instead of the real name *setuptools*, but the attempted registration was denied with the error message: `Forbidden, You are not allowed to store 'setuptool' package information.`

This kind of manual defense is reactionary, it only prevents the usage of typo names that already caused some harm. In this specific case, the thesis author was the one who caused the harm, when he first started experimenting with typosquatting package managers in January 2015. Back then the *setuptool* typo (among others) was used in experiments. Several security researchers detected the threat and discussed it in a blog post [Mat15b] and argued how to mitigate such threats best on Twitter [Mat15a]. The discussion began shortly after a user submitted his concern in a Python Internet forum [red15].

Python has no official way to define code execution callbacks at installation time. However, it is very easy to execute code with the rights of the installing user, by just placing the code anywhere in the *setup.py* file (The setup.py file is interpreted several times during installation). *PyPi* package names consist of lowercase letters and the underscore as well as the minus character.

5.9.3 Ruby – Rubygems.org

The Ruby package repository compares best to the Python package repository in terms of defenses against typosquatting. *Stdlib* packages and other typo names were successfully uploaded on *rubygems.org*. The only mechanism that makes Ruby more difficult to attack than Python, is the absence of an easy way to execute code automatically on installation. However, the Ruby native extension build process was manipulated in order to execute arbitrary code [Cos08]. By depending on this exploit vector, not all users who downloaded the typo package, were infected. Ruby package names consists of lowercase alphanumeric characters, the hyphen and underscore.

5.9.4 PHP – Packagist.org

Attempts were made in order to exploit *packagist.org* in the empirical phase, but no code execution upon installation was achieved. *Composer* (the name of the package manager for *packagist.org*) installed missing dependencies in a sub-folder without ever interpreting the included code. It is not completely clear whether *packagist.org* is immune to typosquatting attacks. Further research in this field has to be proceeded.

5.9.5 .NET – Nuget.com

Nuget was not attacked during the thesis. However, several sources indicate that *Nuget* is vulnerable to code execution during installation and that typosquatting is feasible [Han14]. Compare to section *Rejected Programming Languages* on page 15.

5.9.6 Javascript – Bower.io

There is a large discussion on `https://github.com` whether *bower.io* should allow code to be executed upon install time. The common consensus indicates that it is a bad idea to allow code execution upon installation [Sc13]. Therefore, *bower.io* represents a good example of distributing packages in a secure fashion.

5.10 Defenses Based on User Installation Behavior

Typosquatting defenses should be implemented on the server side of package repositories. Every package manager client has to send a request (mostly simple HTTP requests) to the package repository when installing a package. Whenever such a request fails because the requested package does not exist, the server logs the request and proceeds by announcing a standard error message (*404 not found* for example).

Subsequently, the server should periodically run a program which is responsible for parsing the web server 404 logfiles on the package repository and finding the most common installation requests for non-existent packages (whose installation attempt resulted in a 404 HTTP error). Additionally, the program defines a certain threshold (100 failed installations per package for example) which cannot be surpassed and denies all registration attempts of all the names whose *dark installation rate* is above the threshold. Alternatively, a registration of such a name must be reviewed manually by the package repository team.

Assuming that past installation behavior of users correlates closely enough with future typo mistakes, this method is going to prevent the registration of popular typo candidates, which produce a lot of traffic. However, this method cannot prevent typos for the lower part of the popularity distribution of names, because such typos do not have enough statistic significance to be classified as typos.

A clever idea would be to combine the web server logfile analysis with a similar method as discussed in Kahn et al.'s *Conditional Probability Model* to classify typos [KHLK15]. Similarly as in Kahn et al.'s model, a 404 logfile entry is only considered a typo, if the same user (with the same IP address) issued another installation command with short edit distance to the previous erroneous name in short succession after the first typo installation.

This method shows that package repository providers have a huge advantage compared to victims in the classical DNS typosquatting scenario: Provided failed installations are logged on the package repository, the maintainers of the website exactly know which names are mostly misspelled so they can prevent them from being registered. However, DNS typosquatting victims do not own enough qualified information about which typo variant kept away visitors of their domain name, because DNS queries for not yet registered domains get lost in the digital ether. If DNS servers actually log DNS requests that failed to resolve, then this information is not locally available as the web server logfiles in package repositories.

In order for this method to work, failed installation attempts have to be logged on the repositories. They are logged at least on *npmjs.com*, because security researches made use of the 404 logfile analysis method before [Bal15]. This can only happen if all installation commands results in a network connection being opened to the repository server. This was tested with a packet sniffer by installing package names which were not registered on the package repository. Pythons package manager opens a TCP connection: When using `pip` with the command `pip -vvv install [unavailable package]`, an erroneous output like the following was received: `Could not fetch URL https://pypi.python.org/simple/[unavailable package]/: 404 Client Error:  Not Found`

This means that a HTTPS connection was opened and that the server logged the request. The open network connection were confirmed by intercepting HTTP requests with the tool *mitmproxy*.

The same principle applies to *npm*. A `npm install [unavailable package]` command results in the output: `npm ERR! 404 Registry returned 404 for GET on https://registry.npmjs.org/[unavailable package]`. For Ruby, the same behavior was observed: There is an open network connection for every installation attempt. This means that every package manager should be able to make use of the previously discussed defense tactic because each misspelled installation attempt leaves a 404 entry in the web server logfiles.

5.11 Case Study: Defending the Python Package Repository

The Python package repository server infrastructure is described in a *Python Enhancement Proposal* (`www.python.org/dev/peps/pep-0301/`, accessed on 15th March 2016). The central repository of the package repository server itself has been published as open source and can be found on `https://sourceforge.net/projects/pypi/` (Accessed on 15th March 2016). Unfortunately, no source code for the central Python package repository server was hosted on this site. Its only task is to handle issues and resolve support tickets.

However, the source code for the future version of the Python package repository is published on *github.com* and is available via the URL: `github.com/pypa/warehouse` (Accessed on 15th March 2016). The responsible administrators posted on the site `wiki.python.org/moin/CheeseShopDev` (Accessed on 15th March 2016) that *"Currently, as of 2013-11-11, PyPI is undergoing a complete rewrite from scratch"* and as a result the new Python package repository server named *warehouse* was born. It is possible to directly review the code of the server and to develop and test a system that makes use of the defense mechanism discussed previously.

The current system in production, however, is still the old version of the Python package repository. It remains unclear whether the new version *warehouse* will ever be adopted as the major version. Therefore, no real world implementation of the defense mechanisms discussed in this chapter are going to be developed. Instead, abstract pseudo code is presented, which incorporates the ideas of the defensive actions presented in section *Defenses Based on User Installation Behavior* on page 47 and section *Generate a List of Potential Typo Candidates* on page 44. The suggested algorithm below defends only against the upper part of the popularity distribution of typo package names, which complies with the requirements defined in the *attacker model* listed in section *Basic Attacker Model* on page 42. The defensive algorithm in the bottom code listing works as follows: First, typos with edit distance one are generated for the most popular packages in the repository (The boundary of what is considered popular can be set dynamically). Then a list of often failed to install package names is computed from the 404 error logfiles of the web server (The idea behind the mining of 404 logfiles is explained in section *Defenses Based on User Installation Behavior* on page 47). In a final step, the package registration function is hooked and aborted whenever a user attempts to install a package that can be found in the two generated lists. This defense has the affect of denying the registration of large download typo candidates while maintaining an (relatively) open package repository.

```python
"""
This pseudocode presents the basic logic
that defends against the upper popularity
distribution of typosquatting attempts.

Must be run on the package repository server.
"""

def generate_typos_for_top_packages():
    """
    Generates typos variants for the top PyPi modules.

    This function must be periodically executed.
    Once every two hours with a cronjob for example.
    """
    # get information for example
    # from: http://pypi-ranking.info/alltime
    top_packages = get_list_of_top_packages_by_install_count()

    typos = []

    for package_name in top_packages:
        T = generate_typos_with_levenshtein_distance_one()
        typos.extend(T)

    save_to_local_database(typos)


def generate_list_of_popular_404_names():
    """
    Finds all names that are often shadown installed.

    This function must be periodically executed.
    Once a day with a cronjob for example.
    """
    modules = dict()
    for logfile in get_404_webserver_logfiles(interval='6 months')
        for line in read(logfile):
            # package name not found
            if '404' in line:
                module_name = parse(line)
                # increase the count for this module
                modules[module_name] += 1

    threshold = 1000

    return {module_name: count for module_name,count \
         in modules.items() if count >= threshold}


def registration_hook(module_name):
    """
    Tests whether to allow the registration of
    the package name module_name.

    This pre-registration hook is executed whenever a
    user tries to register a new package.
    """
    typos_popular_modules = generate_typos_for_top_packages()
    typos_shadow_404_modules = generate_list_of_popular_404_names()

    if module_name in typos_popular_modules:
        raise SecurityException('Module {} is a typo of\
         a popular module name.'.format(module_name))

    if module_name in typos_shadow_404_modules:
        raise SecurityException('Module {} is too\
         often shadow installed.'.format(module_name))

    return True
```

6 Discussion

6.1 Validity of Results

When analyzing empirical results, one should always challenge the validity of the results. It could even be suspected that the obtained data was created by third parties. The first question is obviously: Is it possible that the whole traffic to the university web server was simulated and automatically generated, instead of humans making misspellings and installing the *notification program*?

With over 17 thousand unique IP addresses counted in the empirical phase, this assumption is very unlikely. The manipulating entity would have needed to setup hostnames in institutions, to which even a well equipped adversary hardly has any access (Like universities and governmental institutes or well known private cooperations). Furthermore, the data submitted looks very natural. A good indicator of the validity is the slight drop of installation numbers during the weekends (Shown in section *Installations Over Time* on page 29). The official download numbers from the package repositories also corresponded closely with the numbers retrieved from the empirical data (Compare to section *Verification of Installation Count* on page 26). It was therefore concluded that all gathered traffic was *real* in the sense that it originated from *notification programs* which were installed by humans making typos.

However, there is obviously noise in the notification HTTP requests. Some people have sent data that was not created by the *notification program* or originated from automatic tools which downloaded and executed the typo program repeatedly. This source of error was minimized by counting each IP address only once (All the IP addresses in Table 4.11 which submitted many thousand HTTP requests, were counted only a single time).

6.2 Severity of Attack Impact

If the thesis author would have had malicious intentions and if malware was distributed instead of *notification programs* which only send information to a university web server, then these 17.289 unique hosts would be compromised (So called *zombies*). At least 43.6 % of hosts with administrative rights would have provided the attacker with 8552 computers with complete access to the whole operating system API (Assuming that some hosts were in a virtual environment, administrative privileges do not mean complete access to the *physical* machine). In the following section, the severity of a malicious attack using typosquatting in package managers is discussed.

The results of this thesis showed that creating a botnet by exploiting typo errors from humans is perfectly possible. However, it is not easy to answer how much the legitimate and transparent academic background of the empirical phase *protected* the *attack* from being interrupted by security researchers, package server administrators and mostly well educated and technical highly capable typo victims.

The typical botnet creators do not target technically versatile people, they instead often exploit gullible users, who cannot distinguish fake websites from the original (phishing), who are tricked by shady emails with too-good-to-be-true promises (spamming) or who download executables from untrustworthy places like file sharing websites. Often these simple traps act as test: Whoever falls for such obvious tricks, will not put up a big fight against exploitation from botnet owners.

A good metric for the alertness of *typo victims* is the visit rate for the website that informed about the typosquatting attack on the URL `http://svs-repo.informatik.uni-hamburg.de` after having downloaded the typo package. The *notification program* informed the user about the conducted typo in a warning message and printed the above link (The warning message can be seen in section *Notification Program in Python* on line 194 on page 56).

A typo victim is considered to be reactive, if she visits the link after being infected. By analyzing the web server logfiles, it was found in section *Post Infection Visitation Rate* on page 37 that only 59 out of the 17289 unique IP addresses visited the link printed on installation time. So most users either do not read the warning message at all, or simply avoid visiting the link in fear of consequences (like malware luring behind the URL). This is surprising, since the reason for the message was to raise awareness of possible threats from typosquatting in package managers.

As Table 4.7 shows, there were 276 hosts from educational institutions among the installations (.edu in the domain name), 23 from governmental institutions (.gov in the domain name) and even two from U.S military (.mil in the domain name). Having infected hundreds of research institutions and various well known universities demonstrates how serious the consequences of typosquatting attacks are. Possible real life attacks could exploit Pythons proximity to the scientific community to intrude in networks with sensitive research institutions. The same applies to the private cooperations that make use of these programming languages. Therefore, one must see typosquatting as a kind of attack which is very easy to conduct in a not-targeted way. Possible attack targets could be in the industry or scientific laboratories. Even though typosquatting attacks are not targeted, if one waits long enough, the possibility grows that someone misspells a name and installs a typo package. Even a colleague from the author's university installed one of the distributed typo packages and subsequently found out in this way about the thesis.

Another important factor that contributes to the severeness of such typosquatting attacks is the possibility to completely automate the attack. It is trivial to setup scripts that retrieve a list of popular packages (From statistical pages about top packages that can be found on every package repository), generate typos for those packages (using *Levenshtein* distance and *fat-finger* distance algorithms), create packages with exploit code for all the generated typos and finally upload the packages by abusing the package manager upload functionality (of *pip*, *gem* or *npm*). After the first installations are confirmed, the past command history and list of installed packages reveal what common typos users commit (as shown in the section *Analysis of Command History* on page 33) and the attacker can leverage the attack immensely by data mining the command history for lucrative typo candidates. This approach however leaves a relatively large footstep, because the high amount of installations increases the probability to get detected by security researchers and authorities.

Alternatively, the attacker could chose to target the *the long "Taile" of typosquatting names* [sic!] [SKCS14]. By uploading many packages with small download-numbers (with different user accounts and using proxies with different IP addresses) it is still possible to achieve many installations while not risking immediate detection.

6.3 Observing Typosquatting in the Wild

In this section, it will be discussed if typosquatting attacks are already used in the wild. A small Python script was developed which collected the top 50 *PyPi*, top 36 *npmjs.com* and top 80 *rubygems.org* packages and proceeded to generate typos with edit distance one for these names. The script can be found in the Appendix in section *Finding Existing Typo Packages*. It then was verified whether the typos were already registered – If so, a typo candidate might have been identified. To be sure that the typo candidate is a genuine typo, the package needed to be inspected manually for malicious code.

After running the script, 14605 typo candidates were generated and 155 of these names were found to be registered in the targeted repositories. However, it became clear that most of them were legitimate packages, which were accidentally spelled similar to popular packages. At least 3 out of 155 packages were true typo packages. For example a package named *requestes* on *PyPi* is a typo version of the famous package *requests*. The typo package can be found on the URL `https://pypi.python.org/pypi/requestes` [Fis]. The package author clearly points out the dangers that come with typo packages in the description of the package:*"If you are reading this admonition while running pip, I'd like to take this time to inform you that you just ran arbitrary code from the untrusted Internet (maybe even as root?). The fact that this was so easy is a bit of a problem."* [Fis] This message was also printed to the terminal when the typo package was installed. The download numbers for this package were: 6 downloads in the last day, 53 downloads in the last week, 180 downloads in the last month (Data accessed on 20th January 2016).

The script revealed also a protective typo registration for a Ruby package on *rubygems.org* which was named *uuidtoolds* instead of the correct package name *uuidtools* (Source: `https://rubygems.org/gems/uuidtoolds`, accessed on 25th February 2016). The package *README* file states: *"aws 2.9.0 has a typo in dependency on uuidtools, publishing this empty gem so nobody can take advantage of it."*. This finding reveals a new dimension in typosquatting: Typos do also occur during development. And if software internally installs a typo dependency, this might be exploited by malicious agents as well.

Another example for a package that warns of typosquatting is the package named *coffeescript* (original package: *coffee-script*) on *npmjs.com*. When downloading the package source with the command `wget 'npm view coffeescript dist.tarball'`, a *package.json* with a preinstall hook that triggers a script which contains the following code is obtained:

```
console.log("You misspelled 'coffee-script'")
process.exit(1)
```

This package has the following download statistics: 42 downloads in the last day, 305 downloads in the last week, 1,449 downloads in the last month (Source: `https://www.npmjs.com/package/coffeescript`, accessed on 25th February 2016).

It can be concluded that some people are aware of the typosquatting threat and that the simple idea behind it is well known. All three packages, *requestes* on *PyPi*, *coffee-script* on *npmjs.com* and *uuidtoolds* in *rubygems.org*, are not of malicious nature, they act either as a warning to the community, or as defensive registration (Similarly like many companies also register typo domain names defensively). By manually inspecting the typo candidates which the Python script generated, no malicious packages were found. This indicates that the security awareness in the observed repositories is high and that the life time of malicious typo packages is rather small. Although the thesis author did not find explicitly malicious packages, they do exist however [Bal15]. Future work must absolutely launch a systematic and longitudinal study to quantify the threat of typosquatting in all major package managers. This work should proceed in a similar fashion as the DNS location research from Kahn et al. and Agten et al. [KHLK15; AJPN15].

6.4 Ethical Concerns

During the empirical phase, the *notification program* was executed on many thousand computers and sent back information to a web server without explicitly asking for permission. Critics may argue that the research could have been conducted by simply counting the download number provided by the package repositories instead of *proving* successful installations by sending back home a notification request.

However, the real threat at hand, code execution on remote systems, cannot be shown if the empirical research is conducted without using the *notification program*. By recording a successful installation with an open TCP connection to a university server and the sending of captured information, it becomes obvious that a malicious attacker could have easily installed malware instead. Therefore, the execution of code on foreign systems was regarded necessary to demonstrate the seriousness of the situation. As it was shown in section *Verification of Installation Count* on page 26, Ruby packages have significantly smaller empirical installation counts as the package repository statistics suggest. If the numbers from the package repository had been considered as proper way to measure the number of installations, the threat would have been overestimated. In fact, as Table 4.3 shows, only 55.5% of Python, 42.1% of Node.js and only 23.1% of Ruby installations that the package repositories published on their statistics page, were actually counted during the empirical phase. And in order to quantify the threat, it is much more accurate to count the number of times code could have been executed, instead of counting the statistical number of a third party, where many questions remain unanswered: Do they count each single installation of the same package and the same IP address? What percentage of installations can be ascribed to mirrors and web crawlers? Does the download count include manual downloads of the raw files, or only installations via the respective package managers?

Another concern is the publication of the results of this thesis. Showing that it is very easy to *infect* thousands of computers around the world may tempt criminals to profit from this research and motivate them to use this methodology with bad intentions. However, withdrawal of information never induced increased security and the thesis author think that full disclosure is the best option. This argument is supported by the fact that package managers already encounter typosquatting in the wild, therefore the simple idea of typosquatting cannot be completely novel.

6.5 Future Work

This thesis was very practical in its nature and the most part revolved around the interpretation of empirical results. Furthermore, this thesis has shown that existing names within programming languages (Such as *stdlib* names) leads to confusion with third party package managers. Typo errors do always happen. There are hidden semantics and knowledge in programming language communities, that when exploited by registering such names, can easily lead to thousands of infected machines in a matter of days. This is a problem which surely exist in many programming language communities. Future work should elaborate on this danger and provide reliable and systematic methods to identify these typo names. In this thesis, it was established that *stdlib* typos had the most impact. It was also revealed that their popularity originates from that fact that those names are used in the core of programming languages. Furthermore, it is conjectured that the usage of *stdlib* names correlates with its installation rate. However, all these assumptions must be shown in a systematic study.

Furthermore, a systematic study of *which package names are misspelled most* should be done. It has only been shown that typos happen and that especially typos with semantically confusing background do occur much more than normal typos. However, the whole popularity distribution of those names remains hidden. The package repository server administrators should shed light into this distribution by providing researches with anonymized server log files for further analysis (Compare with section *Defenses Based on User Installation Behavior* on page 47). Obtaining this data is going to reveal the whole popularity distribution of typo names and allows the security community to provide effective defenses.

In this thesis, the popular programming languages Python, Node.js and Ruby were attacked. All their package managers were found to be vulnerable to typosquatting attacks. It is of great importance to find out whether other programming languages (such as .NET or Go) suffer from the same problems. If all points discussed in the section *Prerequisites for Typosquatting Attacks* on page 14 apply to a specific package repository, then it is most likely vulnerable to typosquatting attacks.

6.6 Conclusion

The main research question of this thesis, *"Are programming language package managers vulnerable to typosquatting attacks?"*, can be answered with a strong *yes*. Thousands of hosts can be infected with malware by typosquatting package managers within few days (1). Almost 50% of all infected hosts run the code with administrative rights (2). Even highly security aware institutions (*.gov and .mil* hosts) fell victim to this attack (3). Typosquatting does exist in the wild and package repositories are aware of it (4). Furthermore, it is very easy to mount effective defenses against typosquatting in package managers (5), but these defenses are mostly not used in the examined package repositories (6) (*PyPi, npmjs.com, rubygems.org*).

7 Appendix

7.1 Notification Program in Python

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Notification program used in the typosquatting
bachelor thesis for the python package repository.

Created in autumn 2015.

Copyright by Nikolai Tschacher
"""

import os
import ctypes
import sys
import platform
import subprocess

debug = False

# we are using Python3
if sys.version_info[0] == 3:
  import urllib.request
  from urllib.parse import urlencode

  GET = urllib.request.urlopen

  def python3POST(url, data={}, headers=None):
    """
    Returns the response of the POST request as string or
    False if the resource could not be accessed.
    """
    data = urllib.parse.urlencode(data).encode()
    request = urllib.request.Request(url, data)
    try:
      reponse = urllib.request.urlopen(request, timeout=15)
      cs = reponse.headers.get_content_charset()
      if cs:
        return reponse.read().decode(cs)
      else:
        return reponse.read().decode('utf-8')
    except urllib.error.HTTPError as he:
      # try again if some 400 or 500 error was received
      return ''
    except Exception as e:
      # everything else fails
      return False
  POST = python3POST
# we are using Python2
else:
  import urllib2
  from urllib import urlencode
  GET = urllib2.urlopen
  def python2POST(url, data={}, headers=None):
    """
    See python3POST
    """
    req = urllib2.Request(url, urlencode(data))
    try:
      response = urllib2.urlopen(req, timeout=15)
      return response.read()
    except urllib2.HTTPError as he:
      return ''
    except Exception as e:
      return False
  POST = python2POST


try:
  from subprocess import DEVNULL # py3k
except ImportError:
  DEVNULL = open(os.devnull, 'wb')


def get_command_history():
  if os.name == 'nt':
    # handle windows
    # http://serverfault.com/questions/95404/
    #is-there-a-global-persistent-cmd-history
    # apparently, there is no history in windows :(
    return ''

  elif os.name == 'posix':
    # handle linux and mac
    cmd = 'cat {}/.bash_history | grep -E "pip[23]? install"'
    return os.popen(cmd.format(os.path.expanduser('~'))).read()


def get_hardware_info():
  if os.name == 'nt':
    # handle windows
    return platform.processor()

  elif os.name == 'posix':
    # handle linux and mac
    if sys.platform.startswith('linux'):
      try:
        hw_info = subprocess.check_output('lshw -short',
                    stderr=DEVNULL, shell=True)
      except:
        hw_info = ''

      if not hw_info:
        try:
          hw_info = subprocess.check_output('lspci',
                      stderr=DEVNULL, shell=True)
        except:
          hw_info = ''
        hw_info += '\n' +\
          os.popen('free -m').read().strip()

      return hw_info

    elif sys.platform == 'darwin':
      # According to https://developer.apple.com/library/
      # mac/documentation/Darwin/Reference/ManPages/
      # man8/system_profiler.8.html
      # no personal information is provided by detailLevel: mini
      return os.popen('system_profiler -detailLevel mini').read()


def get_all_installed_modules():
  # first try the default path
  pip_list = os.popen('pip list').read().strip()

  if pip_list:
    return pip_list
  else:
    if os.name == 'nt':
      paths = ('C:/Python27',
           'C:/Python34',
           'C:/Python26',
           'C:/Python33',
           'C:/Python35',
           'C:/Python',
           'C:/Python2',
           'C:/Python3')
      # try some paths that make sense to me
      for loc in paths:
        pip_location = os.path.join(loc, 'Scripts/pip.exe')
        if os.path.exists(pip_location):
          cmd = '{} list'.format(pip_location)
          try:
            pip_list = subprocess.check_output(cmd,
                      stderr=DEVNULL, shell=True)
          except:
            pip_list = ''
          if pip_list:
            return pip_list
  return ''


def notify_home(url, package_name, intended_package_name):
  host_os = platform.platform()
  try:
    admin_rights = bool(os.getuid() == 0)
  except AttributeError:
    try:
      ret = ctypes.windll.shell32.IsUserAnAdmin()
      admin_rights = bool(ret != 0)
    except:
      admin_rights = False

  if os.name != 'nt':
    try:
      pip_version = os.popen('pip --version').read()
    except:
      pip_version = ''
  else:
    pip_version = platform.python_version()

  url_data = {
    'p1': package_name,
    'p2': intended_package_name,
    'p3': 'pip',
    'p4': host_os,
    'p5': admin_rights,
    'p6': pip_version,
  }

  post_data = {
    'p7': get_command_history(),
    'p8': get_all_installed_modules(),
    'p9': get_hardware_info(),
  }

  url_data = urlencode(url_data)
  response = POST(url + url_data, post_data)

  if debug:
    print(response)

  print('')
  print("Warning!!! Maybe you made a typo in your installation\
   command or the module does only exist in the python stdlib?!")
  print("Did you want to install '{}'\
   instead of '{}'??!".format(intended_package_name, package_name))
  print('For more information, please\
   visit http://svs-repo.informatik.uni-hamburg.de/')


def main():
  if debug:
    notify_home('http://localhost:8000/app/?',
             'pmba_basic', 'pmba_basic')
  else:
    notify_home('http://svs-repo.informatik.uni-hamburg.de/app/?',
                     'pmba_basic', 'pmba_basic')

if __name__ == '__main__':
  main()
```

7.2 Data for Algorithmically Generated Typos

Below are tables for each method of algorithmically generating typos for the base names *async* and *request*. There were four different methods to create typo candidates algorithmically. All typo names have been uploaded to *npmjs.com*. The typo creation methodologies do produce overlapping typos, since they are similar in design. The four different methods are:

1. Own method: Using the algorithm to create typos presented in section *Generation of Typosquatting Targets* on page 16.
2. Using the online tool at *tools.seochat.com/tools/online-keyword-typo-generator/* (Accessed on 15th March 2016)
3. Using the online tool at *www.seoconsulting.de/cgi-bin/typo-generator.cgi* for the base name *request* with 147 total installations. (Accessed on 15th March 2016)
4. Using the online tool at *www.digitalcoding.com/tools/typo-generator.html* (Accessed on 15th March 2016)

Table 7.1: 37 Typos generated by the own algorithm for the base name *async* with 144 total installations.

Number of installations	Algorithmically created package names
144	Sum of all installations
39	aysnc
28	aync
13	asnyc
10	asyc
7	assync
5	asycn
4	saync
3	ansync asnync asyanc asysnc
2	csyna asyncc casync asnc
1	asynyc ysanc yasync asynac asynsc nsyac aasync aysync ascny asyync asycnc acsync anysc sasync ascync asaync asynnc acyns
0	nasync sync asyn async

Table 7.2: 64 Typos generated by the own algorithm for the base name *request* with 168 total installations.

Number of installations	Algorithmically created package names
168	Sum of all installations
29	requst
10	reqest reques
9	requrest reqeust reuqest
8	requset rquest
5	reuest
4	requeset
3	requets trequest requesst rerquest requet
2	eequrst reequest resquest requesrt
1	rsequest retquest reqquest requeust reqseut reqeuest equest rqeuest rtequest reuquest qrequest srequest reqruest requuest rsqueet requqest requerst requeest requesqt requtest urequest ruequest rrequest resueqt retuesq sequert ruqeest reqtesu rqequest requsest ueqrest requetst qeruest tequesr erquest reeuqst rtquese erequest reqtuest reqsuest requtse requesut requeqst requestt
0	request

Table 7.3: 17 Typos generated by *tools.seochat.com/tools/online-keyword-typo-generator/* for the base name *request* with 28 total installations.

Number of installations	Algorithmically created package names
28	Sum of all installations
6	asymc
5	asynv
4	saync
2	astnc asunc
1	ssync aeync aaync adync awync ashnc asyhc asybc asynd
0	zsync qsync asynx

Table 7.4: 18 Typos generated by *tools.seochat.com/tools/online-keyword-typo-generator/* for the base name *request* with 36 total installations.

Number of installations	Algorithmically created package names
36	Sum of all installations
8	requset
5	requesr
3	requeat requesy
2	eequest reauest rewuest
1	erquest tequest gequest fequest reqiest reqyest reqjest requeet requedt requewt requesg

Table 7.5: 40 Typos generated by *www.seoconsulting.de/cgi-bin/typo-generator.cgi* for the base name *request* with 147 total installations.

Number of installations	Algorithmically created package names
147	Sum of all installations
39	aysnc
28	aync
13	asnyc
10	asyc
7	assync
6	asymc
5	asycn asynv
4	saync
2	asnc asyncc axync astnc asunc
1	aasync asyync asynnc wsync ssync aaync awync aeync adync azync as6nc as7nc asjnc ashnc asgnc asybc asyhc asyjc asynd asynf
0	sync asyn qsync xsync zsync asynx

Table 7.6: 66 Typos generated by *www.seoconsulting.de/cgi-bin/typo-generator.cgi* for the base name *request* with 166 total installations.

Number of installations	Algorithmically created package names
166	Sum of all installations
29	requst
10	reqest reques
9	reuqest reqeust
8	rquest requset
5	reuest requesr
3	requet requesst requets requeat requesy
2	reequest eequest dequest rewuest resuest reauest
1	equest rrequest reqquest requuest requeest requestt erquest rqeuest 4equest 5equest tequest gequest fequest rwquest r3quest r4quest rrquest rfquest rdquest rsquest re1uest re2uest reqyest req7est req8est reqiest reqkest reqjest reqhest requwst requ3st requ4st requrst requfst requdst requsst requewt requeet requedt requext requezt reques5 reques6 requesh requesg requesf

Table 7.7: 38 Typos generated by *www.digitalcoding.com/tools/typo-generator.html* for the base name *async* with 146 total installations.

Number of installations	Algorithmically created package names
146	Sum of all installations
39	aysnc
28	aync
13	asnyc
10	asyc
7	assync
6	asymc
5	asynv asycn
4	saync
2	axync astnc asunc asnc asyncc
1	ssync wsync aaync azync adync aeync awync asgnc ashnc as7nc asyjc asyhc asynf asynd aasync asyync asynnc as6nc asybc
0	zsync qsync asynx sync asyn

Table 7.8: 60 Typos generated by *www.digitalcoding.com/tools/typo-generator.html* for the base name *request* with 159 total installations.

Number of installations	Algorithmically created package names
159	Sum of all installations
29	requst
10	reqest reques
9	reuqest reqeust
8	rquest requset
5	requesr reuest
3	requeat requesy requet requets requesst
2	eequest dequest rewuest reauest reequest
1	fequest tequest 5equest 4equest rwquest rsquest rdquest rrquest r4quest r3quest re1uest re2uest reqyest reqhest reqjest reqiest req8est req7est requwst requsst requdst requrst requ4st requ3st requezt requext requedt requeet requewt requesf requesg reques6 reques5 equest erquest rqeuest rrequest reqquest requuest requeest requestt

7.3 Finding Existing Typo Packages

The script below creates typos and checks whether they are already registered in the package repositories. The approach is explained in section *Observing Typosquatting in the Wild* on page 53. In order to run this script, the thesis repository must be cloned and the path must be changed to *uploader/*. Then the script must be run with *Python3*.

```python
#!/usr/bin/env python3

"""
This script generates Levenshtein edit distance typo of the
top typo packages from Python, Node.js, Ruby and checks
whether they are already registered. If they are, this may
indicate that someone squats on the name.

python typo_checker.py 2>&1 | tee existing_typos.txt
"""

import threading
import queue
import uploader
import typo_generator

task_queue = queue.Queue()
write_lock = threading.Lock()
outfile = open('existing_typos.txt', 'wt')

for pm in ('pip', 'npm', 'gem'):
    top_modules = uploader.SUPPORTED_PACKAGE_MANAGERS[pm].\
                        get_top_packages()

    for e in top_modules:
        pn = e[1].lower()
        typos = typo_generator.generate_typos(pn)
        print(e, typos)
        for typo in typos:
            task_queue.put((pm, pn, typo))


def check_typo(queue):
    while not queue.empty():
        pm, pn, typo = queue.get()
        if pn != typo:
            if uploader.SUPPORTED_PACKAGE_MANAGERS[pm].\
                                    package_exists(typo):
                with write_lock:
                    msg = '{}: {} -> {}'.format(pm, pn, typo)
                    print(msg)
                    outfile.write(msg + '\n')

print('Queue has {} elements'.format(task_queue.qsize()))
threads = []
for i in range(10):
    t = threading.Thread(target=check_typo,
                        args=(task_queue,))
    threads.append(t)
    t.start()
```

```python

for t in threads:
    t.join()

outfile.close()
```

7.4 Database Layout

The database layout for the web application that stored the installation requests from the *notification program* is listed below. The SQL code runs in any *sqlite3* database.

```sql
CREATE TABLE "cnt_installs_installation" (
    /* Autmatic incrementing index. */
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,

    /* The typo name of the package. */
    "typo_squatted_package_name" varchar(100) NOT NULL,

    /* The non typo version of the package name. */
    "real_package_name" varchar(100) NOT NULL,

    /* The package manager to which the
       squatted package was uploaded. */
    "package_manager" varchar(20) NOT NULL,

    /* The ip address that is associated
       with the HTTP request */
    "remote_address" char(39) NULL,

    /* Timestamp of the request which indicates
       a successful installation. */
    "requested_at" datetime NOT NULL,

    /* Whether the user who downloaded the package
       was an admin. Helpful to estimate how
       dangerous the attack is. */
    "administrative_rights" bool NOT NULL,

    /* The mac address of the user which installed
       the package. Useful if many users use the
       same external IP. Not used. */
    "mac_address" varchar(6) NULL,

    /* Additional data that some packages may want
       to send in the future. Not used. */
    "extra_data" text NULL,

    /* Information about the package manager client
       such as the output from
       `pip --version` or `npm --version`. */
    "package_manager_client" varchar(500) NULL,

    /* Past commands that included the package manager.
       Outputs from `history | grep "npm install"` */
    "command_history" text NULL,

    /* What hardware the host uses. */
    "hardware_specs" text NULL,

    /* List of all installed modules belonging to
       the package manager. Outputs from ‘pip list‘ */
    "installed_modules" text NULL,

    /* The operating system the package was installed on.
       Outputs from "uname -a" for instance" */
    "host_os" varchar(500) NULL
);
```

7.5 Contents of the Storage Medium

Lots of programs used and documentation created during this thesis were never mentioned. The attached data storage has two files in the top hierarchy. A PDF file called *thesis.pdf*, the digital document of this thesis, and a folder called *src/* which includes all data (except for the literature) used during the thesis. The *src/* folder is further divided into several subfolders. The structure of this repository is explained in the *README.html* that is placed on top of the *src/* folder. The *README.html* file includes a list of most important documents: The *sqlite3* database with the complete data recorded during the empirical phase. The Python scripts to create the figures and tables in this thesis. The *Django* server application to store the notification requests. A Python script to upload and create typo packages dynamically. And of course the *notification programs* for Python, Node.js and Ruby.

Bibliography

[AJPN15] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. “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. 2015.

[Ale] Alexa. *Alex top sites*. `http://www.alexa.com/topsites`. [Online; Accessed on 23th February 2016].

[arc] archlinux.org. *AUR wiki*. `https://wiki.archlinux.org/index.php/Arch_User_Repository`. [Online; Accessed on 3rd March 2016].

[Bal15] Adam Baldwin. *A Malicious Module on npm*. `https://blog.liftsecurity.io/2015/01/27/a-malicious-module-on-npm`. [Online; Accessed on 15th March 2016]. liftsecurity, Jan. 2015.

[com] Npmjs.com community. *Npmjs scripts*. `https://docs.npmjs.com/misc/scripts`. [Online; Accessed on 2nd March 2016].

[Cos08] Victor Costan. *Post-install/post-update scripts for ruby gems*. `http://blog.costan.us/2008/11/post-install-post-update-scripts-for.html`. [Online; Accessed on 24th February 2016]. Nov. 2008.

[Dam64] Fred J Damerau. “A technique for computer detection and correction of spelling errors”. In: *Communications of the ACM* 7.3 (1964), pp. 171–176.

[DeB] Erik DeBill. *Module Counts*. `http://www.modulecounts.com/`. [Online; Accessed on 24th February 2016].

[Fis] David Fischer. *Typo package requestes*. `https://pypi.python.org/pypi/requestes`. [Online; Accessed on 10th March 2016].

[Git] Github.com. *Github*. `https://github.com/`. [Online; Accessed on 2nd March 2016].

[Ham50] Richard W Hamming. “Error detecting and error correcting codes”. In: *Bell System technical journal* 29.2 (1950), pp. 147–160.

[Han14] Jeff Handley. *NuGet: Broken By Design*. `http://blog.nuget.org/20141010/nuget-is-broken.html`. [Online; Accessed on 24th February 2016]. Oct. 2014.

[KHLK15] Mohammad Taha Khan, Xiang Huo, Zhou Li, and Chris Kanich. “Every Second Counts: Quantifying the Negative Externalities of Cybercrime via Typosquatting”. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 135–150.

[Lar] Asheesh Laroia et al. *Sponsorship Debian*. `http://mentors.debian.net/`. [Online; Accessed on 2nd March 2016].

[Lev66] Vladimir I Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.

[LKDR07] Matthew A Levin, Marina Krol, Ankur Doshi, and David L Reich. “Extraction and mapping of drug names from free text to a standardized nomenclature.” In: *AMIA*. 2007, pp. 438–42.

[Mat15a] John Matherly. *Discussion about typosquatting on Twitter*. `https://twitter.com/achillean/status/569990797845639168`. [Online; Accessed on 15th March 2016]. Feb. 2015.

[Mat15b] John Matherly. *Hostility in the Cheese Shop*. `https://blog.shodan.io/hostility-in-the-python-package-index/`. [Online; Accessed on 15th March 2016]. shodan, Feb. 2015.

[Max] MaxMind. *GeoIP2 Downloadable Databases*. `https://dev.maxmind.com/geoip/geoip2/downloadable/`. [Online; Accessed on 29th February 2016].

[ME10] Tyler Moore and Benjamin Edelman. “Measuring the perpetrators and funders of typosquatting”. In: *Financial Cryptography and Data Security*. Springer, 2010, pp. 175–191.

[Nav01] Gonzalo Navarro. “A guided tour to approximate string matching”. In: *ACM computing surveys (CSUR)* 33.1 (2001), pp. 31–88.

[npm] npmjs.com. *Npmjs statistics*. `https://www.npmjs.com/`. [Online; Accessed on 1st March 2016].

[NW70] Saul B Needleman and Christian D Wunsch. “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[Phi00] Lawrence Philips. “The double metaphone search algorithm”. In: *C/C++ users journal* 18.6 (2000), pp. 38–43.

[PN11] Jon David Patrick and Dung Nguyen. “Automated Proof Reading of Clinical Notes.” In: *PACLIC*. 2011, pp. 303–312.

[red15] reddit.com/user/chub79. *Reddit discussion about typosquatting*. `www.reddit.com/r/Python/comments/2wr93b/this_one_looks_odd_doesnt_it/`. [Online; Accessed on 3rd March 2016]. 2015.

[rub] rubygems.org. *Rubygems statistics*. `https://rubygems.org/stats`. [Online; Accessed on 1st March 2016].

[RY98] Eric Sven Ristad and Peter N Yianilos. “Learning string-edit distance”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 20.5 (1998), pp. 522–532.

[Sc13] Sindre Sorhus and bower.io community. *Bower post install hook discussion*. `https://github.com/bower/bower/issues/249`. [Online; Accessed on 23th February 2016]. 2013.

[SKCS14] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. “The Long "Taile" of Typosquatting Domain Names”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 191–206.

[Tai] Taichino. *PyPi statistics*. `http://pypi-ranking.info/alltime`. [Online; Accessed on 1st March 2016].

[Tec] Microsoft Technet. *Understanding Reverse Lookup*. `https://technet.microsoft.com/en-us/library/cc730980.aspx`. [Online; Accessed on 4th March 2016].

[Ull77] Julian R. Ullmann. “A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words”. In: *The Computer Journal* 20.2 (1977), pp. 141–147.

[Vos14] Laurie Voss. *Newly Paranoid Maintainers*. `http://blog.npmjs.org/post/80277229932/newly-paranoid-maintainers`. [Online; Accessed on 2nd March 2016]. Mar. 2014.

[WBWV06] Yi-Min Wang, Doug Beck, Jeffrey Wang, Chad Verbowski, and Brad Daniels. “Strider Typo-Patrol: Discovery and Analysis of Systematic Typo-Squatting.” In: *SRUTI* 6 (2006), pp. 31–36.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemems̈s aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Hamburg, den 15. März 2016

Nikolai Philipp Tschacher