A LARGE-SCALE EVALUATION OF HIGH-IMPACT PASSWORD STRENGTH METERS

Xavier de Carné de Carnavalet

A thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

in Information Systems Security at

Concordia University

Montréal, Québec, Canada

April 2014

© Xavier de Carné de Carnavalet, 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Xavier de Carné de Carnavalet**

Entitled: **A Large-Scale Evaluation of High-Impact Password Strength Meters**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Jamal Bentahar ______________________ Chair

Dr. Amr Youssef ______________________ Examiner

Dr. Zhenhua Zhu ______________________ External Examiner

Dr. Mohammad Mannan ______________________ Supervisor

Approved ______________________________________

Chair of Department or Graduate Program Director

____________ 2014 ______________________________

Dr. Christopher Trueman, Interim Dean

Faculty of Engineering and Computer Science

Abstract

A Large-Scale Evaluation of High-Impact Password Strength Meters

Xavier de Carné de Carnavalet

Passwords are ubiquitous in our daily digital life. They protect various types of assets ranging from a simple account on an online newspaper website to our health information on government websites. However, due to the inherent value they protect, malicious people have developed insights into cracking them. Users are pushed to choose stronger passwords to comply with password policies, which they may not like much. Another solution is to put in place proactive password-strength meters/checkers to give feedbacks to users while they create new passwords. Millions of users are now exposed to these meters at highly popular web services that use user-chosen passwords for authentication, or more recently in password managers. Recent studies have found evidence that some meters actually guide users to choose better passwords—which is a rare bit of good news in password research. However, these meters are mostly based on ad-hoc design. At least, as we found, most vendors do not provide any explanation of their design choices, sometimes making them appear as a black-box. We analyze password meters deployed in selected popular websites and password managers. We document obfuscated open-source meters; infer the algorithm behind the closed-source ones; and measure the strength labels assigned to common passwords from several password dictionaries. From this empirical analysis with millions of passwords, we shed light on how the server-end of some web service meters functions, provide examples of highly inconsistent strength outcomes for the same password in different meters, along with examples of many weak passwords being labeled as *strong* or even *excellent*. These weaknesses and inconsistencies may confuse users in choosing a stronger password, and thus may weaken the purpose of

these meters. On the other hand, we believe these findings may help improve existing meters, and possibly make them an effective tool in the long run.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. Mohammad Mannan for his guidance in the world of research and for pushing this work up to an international conference. I wish to thank him for his kindness and availability, and for the friendly environment he provides to his students. He convinced me that there are even more interesting problems to solve through research in a Ph.D. program, in which I have been recently admitted.

I am also grateful to the anonymous NDSS2014 reviewers, Arash Shahkar and Jeremy Clark for their insightful suggestions and advice, along with my labmates Lianying Zhao (Viau) and Suryadipta Majumdar for their enthusiastic discussions.

My deepest thanks to my friends, parents and brother, for their respective support and insights they provided me in my life, which ultimately contributed to my thirst for knowledge. I must acknowledge I have not been the easiest person they know.

Finally, I dedicate this thesis to my beloved grandfather, who used to say I will become a savant. Well, I'm doing my best so far. Repose en paix.

Contents

List of Figures ix

List of Tables xi

List of Algorithms xii

1 Introduction 1

1.1 Motivation 1

1.2 Thesis Statement 2

1.3 Contributions 3

1.4 Related Publication 5

1.5 Outline 5

2 Background and Related Work 6

2.1 Background 6

2.1.1 Problems of Password 6

2.1.2 Common Solutions to Weak Passwords 10

2.2 Related Work 12

2.2.1 Password Meters and Password Security Evaluations 12

2.2.2 Evaluation of Password Policies 15

2.2.3 Strength Evaluation 16

2.2.4 Cracking Efforts 19

3 Meters Characterization **22**

3.1 Overview 22

3.2 Test Automation 24

3.2.1 Web-Based Client-Side Evaluators 24

3.2.2 Web-Based Server-Side Evaluators 26

3.2.3 Web-Based Hybrid Evaluators 27

3.2.4 Application-Based Evaluators 29

3.3 Tested Dictionaries 31

3.3.1 Cracking Tool Dictionaries 34

3.3.2 Real Password Database Leaks 35

3.3.3 Mangled Dictionaries 35

3.3.4 Leet Transformations 37

4 Empirical Evaluation **39**

4.1 Web-Based Password Meters (Client-Side) 39

4.1.1 Dropbox 39

4.1.2 Drupal 43

4.1.3 FedEx 44

4.1.4 Intel 48

4.1.5 Microsoft 50

4.1.6 Tencent QQ 54

4.1.7 Twitter 55

4.1.8 Yahoo! 57

4.1.9 12306.cn 59

4.2 Web-Based Password Meters (Server-Side) 60

4.2.1 eBay 60

4.2.2 Google 62

4.2.3 Skype 65

4.3 Web-Based Password Meters (Hybrid) 67

4.3.1 Apple 67
4.3.2 PayPal 69
4.4 Application-Based Password Meters 71
4.4.1 1Password 71
4.4.2 LastPass 81
4.4.3 KeePass 84
4.4.4 RoboForm 88

5 Results Analysis **92**
5.1 Results Summary 92
5.2 Meters Heterogeneity and Inconsistencies 95
5.3 Comparison 96
5.4 International Characters 100
5.5 Implications of Design Choices 101
5.6 Stringency Bypass 102
5.7 Google Checker Hypothesis 102
5.8 Password Policies 103

6 Discussion **104**
6.1 Implications of Online vs. Offline Attacks 104
6.2 Password Leaks 105
6.3 Passphrases 105
6.4 Relative Performance of our Dictionaries 105
6.5 Directions for Better Checkers 106

7 Conclusion **108**

Bibliography **116**

List of Figures

1 Dropbox checker password strength distribution 40
2 Drupal checker password strength distribution 43
3 FedEx checker password strength distribution 45
4 Intel checker password strength distribution 48
5 Microsoft checkers (v1, v2 and v3) password strength distribution . . 51
6 Tencent QQ checker password strength distribution 54
7 Twitter checker password strength distribution 56
8 Yahoo! checker password strength distribution 58
9 12306.cn checker password strength distribution 59
10 eBay checker password strength distribution 61
11 Google checker password strength distribution 63
12 Skype checker password strength distribution 65
13 Apple checker password strength distribution 67
14 PayPal checker password strength distribution 70
15 1Password checker (software-based) password strength distribution . . 72
16 1Password: Score versus password length (number of characters) . . . 74
17 1Password: Score versus password length (number of bytes) 75
18 Equations used to compute the coefficients of determination 76
19 1Password checker (JavaScript-based) password strength distribution 79
20 LastPass checker password strength distribution 82
21 KeePass checker password strength distribution 85
22 RoboForm checker password strength distribution 89

23 RoboForm disassembled instructions for checking identical consecutive characters 90
24 Password multi-checker output for *password$1* 97
25 Comparison between services assigning decent scores to our dictionaries 99

List of Tables

1 Password requirements and characteristics of the evaluated meters . . 25

2 Dictionaries used against password checkers 32

3 Dictionary overlaps 33

4 Estimated coefficients for 1Password output regressions 76

5 LastPass patterns rewarding rules 83

6 Summary of the types of meters and their capabilities 94

List of Algorithms

1 AutoIt high-level test automation algorithm 31
2 1Password master password's checking algorithm 78
3 1Password Firefox extension's checking algorithm 80
4 KeePass password checker pattern detection algorithm 87
5 Algorithm for RoboForm's identical consecutive characters check . . . 91

Chapter 1

Introduction

1.1 Motivation

Passwords are characterized with long-standing problems and are often not secure enough [46, 6, 53, 73, 33]; however, even with their apparent weaknesses, passwords are likely to remain in use for the foreseeable future [30]. Thus, fixing their weaknesses is an important and challenging problem to tackle. In the past few years, some research has shown that the presence of a password meter during password creation leads ordinary users towards more secure passwords [61, 23]. However, not all meters are equal, and the strengths and weaknesses of widely-deployed password meters have been scarcely studied so far. Furnell [26] analyzes password meters from 10 popular websites to understand their characteristics, by using a few test passwords and stated password rules on the sites. Furnell also reports several inconsistent behaviors of these meters during password creation and reset, and in the feedback given to users (or the lack thereof). Password checkers are generally known to be less accurate than ideal entropy measurements; see e.g., [16, 67]. One obvious reason is that measuring entropy of user-chosen passwords is problematic, especially with a rule-based metric; see e.g., the historic NIST metric [15], and its weaknesses [67]. Allegedly better password checkers have been proposed (e.g., [16, 63, 55, 31]), however we are unaware of their deployment at any public website. We therefore focus on analyzing meters

as deployed at popular websites, especially as these meters are guiding the password choice of millions of users. We also extend the evaluation to meters embedded in password managers, which help users organize their ever-growing set of passwords.

Except Dropbox and KeePass to some extent, no other meters in our test set provide any publicly-available explanation of their design choices, or the logic behind their strength assignment techniques. Often, they produce divergent outcomes, even for otherwise obvious passwords. Examples include: *Password1* (rated as very weak by Dropbox, but very strong by Yahoo!), *Paypal01* (poor by Skype, but strong by PayPal), *football#1* (very weak by Dropbox, but perfect by Twitter). In fact, such anomalies are quite common as we found in our analysis. Sometimes, very weak passwords can be made to achieve a perfect score by trivial changes (e.g., adding a special character or digit). There are also major differences between the checkers in terms of policy choices. For example, some checkers promote the use of passphrases, while others may discourage or even disallow such passwords. Some meters also do not mandate any minimum score requirement (i.e., passwords with weak scores can still be used). In fact, some meters are so weak and incoherent (e.g., Yahoo!) that one may wonder what purpose they may serve. Taking into consideration that some of these meters are deployed by highly popular websites, we anticipate inconsistencies in these meters would confuse users, and eventually make the meters a far less effective tool.

1.2 Thesis Statement

The primary objective of this thesis is to assess the relevance of currently deployed password-strength meters by systematically characterizing their assumptions and rules against a selected corpus of about 4 million passwords. As part of this goal, we explore the following research questions:

Question 1. Can we characterize password meters through empirical studies?

Question 2. Do widely-deployed password meters consistently and correctly rate passwords?

Question 3. Can we identify the ideal or expected features of a password meter by analyzing existing high-impact meters?

1.3 Contributions

1. Meter characterization. We systematically characterize the password meters of 14 prominent web service providers, ranging from financial, email, cloud storage to messaging services; we also analyze meters from four leading password management software tools. Our target meters include: Google, PayPal, eBay, Skype, Tencent QQ, Twitter, Apple, Microsoft, Dropbox, Yahoo!, 12306.cn, LastPass, KeePass, 1Password, RoboForm, Intel, Drupal and FedEx. For Microsoft, we test three versions of their checker, two of which are not used anymore. For LastPass also, we test the two versions offered by the tool. This characterization is particularly important for checkers with a server-side component or compiled software component, which appears as a black-box; no vendors in our study provide any information on their design choices. Even for client-side/open-source checkers, no analysis or justification is provided (except Dropbox).

2. Empirical evaluation of meters. For each of the 21 meters, we used nearly four million unique passwords from several password dictionaries (a total of at least 85 million test instances, approximately). Each checker is represented by its response profile to our dictionaries. This is the largest such study on password meters to the best of our knowledge. To understand these checkers, we extract and analyze JavaScript code (with some obfuscated sections) for 13 services and browser extensions involving local/in-browser processing. We further analyze the C# code for a password manager and reverse-engineer, to some extent, the five services involving server-side processing that appear as a black-box along with the two closed-source password managers.

Then, for each meter, we take the relevant parts from the source code (when available) and plug them into a custom dictionary-attack algorithm written in JavaScript and/or PHP. We then analyze how the meter behaves when presented with passwords from publicly available dictionaries that are more likely to be used by attackers and users alike. Some dictionaries come from historical real-life passwords leaks. For each meter, we test nearly four million passwords from 11 dictionaries (including a special leet dictionary we created). We also optimize our large-scale automated tests in a server-friendly way to avoid unnecessary connections, and repeated evaluation of the same password. At the end, we provide a close-approximation of each meter's scoring algorithm, weaknesses and strengths of the algorithm, and a summary of scores as received by our test dictionaries against the meter.

3. METER WEAKNESSES. Weaknesses exposed by our tests include: (a) Several meters label many common passwords as of decent quality—varying their strengths from *medium* to *very strong*; (b) Strength outcomes widely differ between meters, e.g., a password labeled as *weak* by one meter, may be labeled as *perfect* by another meter; and (c) Many passwords that are labeled as weak can be trivially modified to bypass password requirements, and even to achieve *perfect* scores. These weaknesses may cause confusion and mislead users about the true strength of their passwords. Compared to past studies, our analysis reveals the extent of these weaknesses.

4. TEST TOOLS. We have implemented a web-based tool to check results from different vendors for a given password. In addition to making the inconsistencies of different meters instantly evident, this tool can also help users choose a password that may be rated as strong or better by all the sites (from our web-based test set), and thus increasing the possibility of that password being effectively strong. Test tools and password dictionaries as used in our evaluation are also available for further studies.

1.4 Related Publication

Conference Paper. The work discussed in this dissertation has been peer-reviewed and published in the following article:

> From Very Weak to Very Strong: Analyzing Password-Strength Meters. X. de Carné de Carnavalet and M. Mannan. *Network and Distributed System Security Symposium (NDSS'14)*, Feb 23-26, 2014, San Diego, CA, USA.

1.5 Outline

The rest of this thesis is organized as follows. Chapter 2 covers some necessary background and literature related to this dissertation. In Chapter 3, we first explain some common requirements and features of the studied password checkers; then discuss issues related to our automated testing of a large number of passwords against these meters; and detail the dictionaries used in this study, including their origin, characteristics, and some common modifications we performed on them. We present the tested web services and password managers, and their respective results in Chapter 4. In Chapter 5, we further analyze our results and list some insights as gained from this study. Chapter 6 discusses more general concerns related to our analysis. Chapter 7 concludes.

Chapter 2

Background and Related Work

This chapter covers some necessary background and literature related to this dissertation.

2.1 Background

In this section, we present common problems related to passwords and focus on commonly used solutions for the particular problem of user-chosen weak passwords.

2.1.1 Problems of Password

Passwords suffer from several problems that have been studied and documented for the past decades. We discuss seven common ones below.

2.1.1.1 Strength vs. Memorability Dilemma

As pointed by Morris [46] back in 1979, users tend to choose simple passwords that they can remember. It is indeed a dilemma: passwords are either easy to remember but easily guessable, or difficult to guess but also hard to remember [28, 75, 1, 37]. The main culprit is the limitation of human memory. As reported by Johnson [35] and Miller [44], human memory is limited in time and can remember only about seven plus or minus two items that should sound already familiar to the person. Yan et al. [72]

show by a study on 400 first-year university students that random passwords—which yield better security, are more difficult to memorize. They suggest mnemonic-based passwords as being easier to memorize while presenting good strength. Adams and Sasse [1] also report that four or five passwords are the most users are expected to handle regularly. Users also forget their passwords, which tend to be a painful process to go through in organizations depending on their way of solving this issue [32].

2.1.1.2 Small Effective Search-Space

Users tend to choose the weakest[1] password they are allowed to select [24], resulting in the rise of simple common patterns. Veras et al. [63] analyze the semantic of revealed websites' passwords and find that most are related to love, sexual terms, profanity, animals, alcohol and money. In an experiment with mnemonic-based passwords, Kuo et al. [40] show that the sentences chosen by the participants are primarily based on music lyrics, movies, literature, or television shows. These studies demonstrate that a trained attacker can narrow his search to common topics and terms in order to maximize password cracking efficiency.

Furthermore, due to policy enforcements, users are known to adapt their simple passwords to make them compatible with the new policies [62]. Several past studies (e.g., [66, 38, 19]) and cracking tools (e.g., [50]) reveal the patterns commonly followed by users, including appending digits and/or special characters at the end of a password, combining two simple words, inserting digits in front of the password, uppercasing the first letter, etc. While these transformations ensure compliance with the most stringent composition policies, the resulting strength is often not increased by much.

[1] We refer to weak/bad passwords as ones that provide little security against an attacker trying to guess them by state-of-the-art techniques, typically in the order of a million trials without targeted means.

2.1.1.3 Lack of User Education

Users are generally incompetent to judge what constitutes a secure password [1, 6]. Adams et al. [1] mention that, despite users being overall security-conscious, their "knowledge of what constitutes secure password content (the character content of the password) was inadequate." More accurately, Vidyaraman et al. [64] agree on the fact that "users are never motivated to behave in a secure manner" [1] but they are "generally careless and unmotivated when it comes to system security". Bishop et al. [6] also points that users think nobody will guess the permutation they used inside their password, which often turns wrong as mentioned in the previous paragraph.

2.1.1.4 Reuse

In contrast to the number of passwords users can effectively handle, the number of passwords an active web user manages daily is reported to be from 8 (2007 [24]) to 15 (2001 [33]). This is not accounting for less frequently visited websites. In total, Florêncio et al. [24] report that an average user has about of 25 password-protected accounts. As a consequence, users reuse their passwords. Das et al. [19] study password reuse across various database leaks and phishing campaigns and show that users tend to reuse passwords directly 43% of the time, or base new ones on previous ones 19% of the time by inserting or deleting few characters, mostly towards the end of a password (88%). Their survey of university students and staff shows that users modify their password mostly to fulfill password constraints on the newer website. Their results also provide confirmation of mangling and leet transformations habits. Ives et al. [33] point to the dangers of password reuse turning the security of many services dependent on the weakest of them. Their point is further detailed by Haque et al. [27] on a laboratory experiment with 80 participants that shows the risk associated with the compromise of low-level account passwords against high-level accounts such as banking accounts.

2.1.1.5 Database Leaks

So far, several websites have been attacked one way or another and their users database including passwords has been leaked, revealing passwords of millions of users [45]. Examples include phpBB [52] (2009, 255 thousand users), RockYou [60] (2009, 32 million users), Gawker [49] (2010, 1.3 million users), YouPorn [47] (2012, 1.6 million users), LinkedIn [74] (2012, 6.5 million users), Yahoo [7] (2012, 450 thousand users), Twitter (2013, 250 thousand users), Adobe [39] (2013, between 38 to 150 million users depending on the source) and Forbes [18] (2014, 1 million users). Some databases contained passwords in cleartext, leaving accounts instantly vulnerable. On the one hand, such leaks grant attackers with more powerful understanding of how users choose passwords, which they can try to reuse at different websites due to the trend of reusing passwords explained above. These many database leaks also tell us that in reality, passwords may also be obtained from offline password cracking following a security breach rather than simple online guessing, which opens the door for attackers to try a virtually unlimited number of passwords. On the other hand, researchers also gain insights from the public leaks, which led to various insights and contributions (see Section 2.2).

2.1.1.6 Theft by Keylogging or Phishing

Another problem faced by users is that they should keep their passwords secret, and it is not always clear if they are handing them in the right hands. Shoulder surfing is the most basic issue in which another person watches the user while she is typing her password. More advanced techniques consist of malware or hardware recording keystrokes [24, 25, 42] when the user types her password, and sending it stealthily to the attacker. Users may not notice their computer is infected by such malware or is equipped with a hardware keylogger, and attackers can gain access to various kind of accounts protected by even difficult passwords with constant effort [24]. This issue is relatively new compared to others, and unveils new design failures of password schemes [25]. Another technique is phishing through which a user is tricked into

willingly providing her password to a malicious website impersonating the original one. Users do not realize the setup and think they are logging in the right place. Meanwhile, attackers benefit from real fresh passwords. A historical example which results became public is the MySpace phishing attack [56].

2.1.1.7 Internationalization issues

Bonneau et al. [12] study various problems related to password encoding and support for non-English characters on websites. They report low support for foreign passwords, including length miscalculation, misevaluation, misencoding (sometimes browser-dependent), foreign characters deletion or truncation leading to reduction of strength, and ban of foreign characters altogether. They also study English, Chinese, Hebrew and Spanish speakers behaviors and highlight that such users rarely choose their original languages for passwords, rather they use transliteration to ASCII. Also, Chinese and Hebrew users choose numbers a lot (simple ASCII-compliant alternative).

2.1.2 Common Solutions to Weak Passwords

2.1.2.1 Password Policies

Password policies have been used as a way to prevent users from choosing weak passwords. These policies often enforce a minimum number of characters, require various types of characters to be included (e.g., lowercase/uppercase letters, digits, symbols), forbid certain patterns (e.g., the use of user-related information, keyboard sequences, passwords derived from simple dictionary words) [37, 26]. Although the complexity of valid passwords appears higher, such composition policies come with several drawbacks. Studies report that it is more difficult to create a password under stringent policies [38], and get users frustrated from being unable to comply with them easily [32], along with various memorability and usability issues [38, 32]. Mazurek [41] also reveals by evaluating the guessability of passwords and surveying users reaction

of an entire university, that annoying password policies may actually be counterproductive. Also, because users tend to react in a predictable manner to these policies (adapting their old passwords with simple transformations), policies do not really solve the issue of weak passwords.

2.1.2.2 Proactive Password Checking

Another way to obtain stronger passwords from users is to prevent the creation of bad ones by evaluating the strength of a candidate password and returning feedbacks to the user in real-time during password creation. These evaluators are called proactive password checkers or meters and have been around for decades; for some earlier references, see e.g., Morris and Thompson [46], Spafford [59], and Bishop and Klein [6]. Recently, password checkers are being deployed as password-strength meters on many websites to encourage users to choose strong passwords. Password meters are generally represented as a colored bar, indicating e.g., a weak password by a short red bar or a strong password by a long green bar. They are also often accompanied by a word qualifying password strength (e.g., weak, medium, strong), or sometimes the qualifying word is found alone. We use the terms password-strength meters, checkers, and meters interchangeably in this dissertation.

Since users are reported to be security-conscious [1, 32] but fail to act appropriately because of their lack of knowledge or because of various policy inconsistencies, this solution appears as a step forward in the right direction. It indeed provides a mean for communication between the system and the user, which bridges an important gap identified by Adams et al. [1].

Two encouraging studies by Ur et al. [61] and Egelman et al. [23] (see Section 2.2) report a positive impact on the strength of user-chosen passwords while they are presented with a meter during password creation. Our work is based on these findings and serves to understand the extent to which we can hope for the usefulness of currently deployed meters.

2.2 Related Work

Below we discuss related work from early research on password security, password meters, password policies, user reactions, strength checking and password cracking techniques.

2.2.1 Password Meters and Password Security Evaluations

In a recent user study, Ur et al. [61] tested the effects of 14 visually-different password meters on user-chosen password creation. They first look at existing password meters in Alexa's 100 most visited websites and list characteristics of the graphical part of the meters along with some information about the algorithm used when readily available. Compared to this part of their work, we systematically focus on the algorithms behind several currently deployed meters, even if presented as a black-box, and identify weaknesses that may negatively impact regular users. Then, they recruit 2,931 subjects through Amazon's Mechanical Turk crowdsourcing service who are asked to imagine that they need to create a new password to access their emails because of a change in their email provider's policy. They were required to come back two days later and use the password they created. Next, after collecting subjects' passwords, the authors evaluate their strength against 3 types of attackers involving Weir's state-of-the-art cracking algorithm [68] trained on a dictionary of 14 million strings gathered from the OpenWall Mangled Wordlist,[2] the RockYou and MySpace leaked passwords along with strings coming from the Google Web Corpus.[3] Finally, participants are asked to complete a survey about their sentiment towards the task of creating the password. They found that meters indeed positively influence user behavior and lead to better password quality in general. Any meter lead to longer passwords and most lead to more digits and symbols being included. Users tend to reconsider their entire password when a stringent evaluation is given, rather than trying to bypass the checker.

[2] http://www.openwall.com/wordlists/

[3] http://googleresearch.blogspot.ca/2006/08/all-our-n-gram-are-belong-to-you.html

Passwords created under such strict evaluation were significantly more resistant to guessing attacks. More precisely, a weak adversary allowed 500 million guesses could crack 21.0% of the passwords created without meter but only 5.8% and 4.7% in the presence of meters ranking passwords with coefficients 1/2 and 1/3, respectively. A medium adversary allowed 50 billion guesses could crack 35.4% of the passwords in the first condition but only 19.5% and 16.8% in presence of the stringent meters, respectively. Finally, a strong adversary allowed 5 trillion guesses could crack 46.7% of the passwords compared to 26.3% and 27.9% in presence of the stringent meters, respectively.

However, meters with too strict policies generally annoyed users and made them put less emphasis on satisfying the meters. Authors identify two types of users: ones who aim at filling the meter's bar (or reaching a green color), and ones who content themselves of a password that is considered not to be poor (not a red bar).

In another user study, Egelman et al. [23] also reported positive influence of password meters. They also considered context-dependent variations in the effectiveness of meters, and found that passwords for unimportant accounts are not much influenced by the presence of a meter. In the first part of the experiment involving 50 participants recruited with flyers distributed around the university campus, subjects were asked to change their Single-Sign-On passwords. Results draw similar conclusions regarding the length and complexity of chosen passwords compared to the study by Ur et al. In the second part, 200 subjects were recruited from Amazon Mechanical Turk to participate in an unrelated study about beta-testing an Android application. They were contacted two weeks later to login again to see whether they qualified for the unrelated testing. Finally, they were invited one month later to complete a survey. For such unimportant account, the presence of meters led to no noticeable effects. In the study, authors also test a peer-pressure meter design, in which a user is given feedback on the strength of her password compared to all other users of a particular service. The effect of such type of meter was no different than an ordinary meter.

Bishop [5] surveys five proactive password checkers for Unix in 1992, analyzes

their design and compare them with a list of expectations. The latter include black-listing common passwords through comprehensive dictionary, per-user and per-site discrimination (because of user- and site-specific information), preventing reuse in short period of time (aging), and other implementation-related details. Weaknesses are identified in most of them, e.g., passwords failing tests are still accepted, lack of per-user or per-site customizations, addition of a character bypasses the tests, aging is not taken into account, dictionaries are insufficient, and lack of flexibility in the implementations. Two checkers in particular were recommended for their completeness.

Bonneau et al. [11] evaluate 150 websites to point out the various failures surrounding password security as currently implemented. They report that a low number of websites give advices to users while choosing passwords, despite that giving advice is recognized as a good practice [1]; some websites send an email with the user's password in cleartext; newspaper sites enforce email verification by sending a validation link whereas they appear to have little concern about such verification which leads the authors to think it is primarily intended at collecting email addresses for marketing purposes. Evaluated websites are mostly shown to enforce a minimum password length and 6 characters is the most popular threshold, however very few of them impose restrictions that involve running the password against a simple dictionary check or enforcing particular composition. 24% of the websites send the original password by email when the user forgets it and the majority (84%) allowed up to at least 100 login attempts with no restriction including Amazon, eBay and WordPress, the remaining websites allowed between 3 to 6 trials on average with few exceptions (Yahoo!'s limit of 25). 86% reveals membership of a username through the password reset function where as low as 19% provide a generic error for login with an invalid username and about a half only allows a site-specific username compared to identification by the user's email address. 27% have a broken implementation of TLS, preventing some pages such as enrollment, password update, or login forms to be covered. Overall, they evaluate that the field of password security is undermined by inconsistencies and

implementation issues. Bigger, leader or payment-related websites have better security practices and identity providers protects against weak passwords and guessing attacks, while content websites implement the poorest practices and endanger the whole system by being easily targeted by attackers who can reuse collected passwords at higher-importance websites. Too many websites actually require passwords for no real benefit which stresses even more the limited users' mental storage. Finally, authors suggest that low-security websites may require separate weak passwords while high-security website require strong passwords only in order to separate and limit password reuse. Standardization of implementations could bring additional security and prove beneficial to users, and could even be branded. The study concludes that current password security implemented in practice is flawed with technical, market and psychological failures.

Dashlane [43] examined US, French and UK e-commerce websites based on 26 criteria, including minimum password length, acceptance of the 10 most common passwords and whether they sent back users' password in plaintext by emails. Results show poor practices in general. UK e-commerce websites are slightly better in terms of password security compared to the two other countries studied. However, 63% of UK websites did not implement an explicit policy, 66% still accept trivial passwords such as *password* and *123456*, 60% do not provide any advice on choosing strong passwords, and 25% of them send passwords in plaintext via email. We did not include Dashlane's password manager in our evaluation because of difficulties in automating tests against their meter.

2.2.2 Evaluation of Password Policies

Yan et al. [73] divide a class of 400 first-year university students into 3 groups which are given different policies for creating a password. Results show that random passwords are difficult to remember, but mnemonic-based ones are as easy as usual ones. Also, mnemonic-based passwords are harder to guess for an attacker. However, the need for policy enforcement to prevent non-compliance is highlighted.

Komanduri et al. [38] focus on the effect of password-composition policies on users and report that long passwords with no other restrictions result in stronger passwords while being more user-friendly compared to shorter ones created under more restrictive policies. Evidences of common patterns chosen under restrictive policies are also established.

Kelley et al. [36] studied password composition rules by analyzing 12,000 user-chosen passwords under seven different policies, and reported that imposing only a longer length requirement yields better entropy than enforcing charset complexity on smaller passwords. Memorability and usability effects of composition policies are discussed in a separate study [38]. Studied users tend to end a password with several digits and a possible symbol, correlating with the observations from past studies (e.g., Burr et al. [15] and Weir [66]). Such known user behaviors validate the use of mangling rules in password cracking.

Furnell [26] analyzed password guidelines and policies of 10 major web services. The study primarily relied on stated guidelines/policies, and used selective passwords to test their implementation and enforcement. Several inconsistencies were found, including: differences in meters/policies between account creation and password reset pages; the vagueness of recommendations given to users for password strengthening; and the disconnect between stated password guidelines and effective password evaluation and enforcement. We provide a more comprehensive analysis, by systematically testing widely-deployed password meters against millions of passwords, and uncovering several previously unknown weaknesses.

2.2.3 Strength Evaluation

Spafford [59] presents OPUS which solves the issue of storing a large dictionary—at the time of the study (1992), on a system that performs proactive dictionary checks against candidate passwords. OPUS is based on a Bloom filter, which is a probabilistic membership checker. It provides easy addition of new blacklisted passwords and low memory consumption.

Later solutions to improve performance of dictionary checks involve decision trees by Bergadano et al. [4], reducing a 27.18MiB dictionary to a 23.6KiB decision tree classifier, and Minimum Description Length Principle-based decision trees in Hyppocrates by Blundo et al. [9].

Markov chains are leveraged by Davies et al. [20] to achieve the same goal in BApasswd. A dictionary is converted to a Markov model representation at a preprocessing stage, representing the probabilities of transition in the model. During password checking, BApasswd checks whether a given password belongs to a particular Markov chain with a log-likelihood function. The system aims at representing the original dictionary as close as possible, rather than generalizing the knowledge of the dictionary as it is the case in later work [16].

Yan [71] shows the limitations of dictionary checks in proactive password checkers and proposes an alternative based on entropy for 7-character passwords. It is easy to compute the search-space associated with each possible combinations of letters and digits in such password and take into account character repetitions. This way, small search-spaces are identified and associated with low entropy. Passwords falling into these categories should be discarded.

Bishop and Klein [6] mention in 1995 that users are not educated as to what are wise choices of passwords. They provide an extensive list of patterns to ban in passwords including dictionary checks taking into account an exact match, reversed word and various transformations (mangling, leet, misspelling); also enforcing minimum length, preventing inclusion of user-related information and various patterns such as social security numbers, license plates, keyboard sequences, and concatenation of words. The implementation proposed targets Unix systems.

Blundo et al. [8] train a perceptron to recognize whether a password is weak or strong based on a training set that considers strong passwords as containing a symbol or being random. Features provided to the perceptron are the number of charsets, number of "strong" characters, the upper-lowercase distribution, and types of diagrams.

Ciaramella et al. [17] improve on Blundo, study various neural-network constructions for password checkers, stick to a multilayer perceptron and train it to distinguish weak and strong passwords. They evaluate the performance against Hyppocrates [9] and CrackLib. The solution dramatically reduces storage space requirements, as dictionaries are no longer needed after training. The system is capable of fitting in very limited environments such as smart cards.

Jamuna et al. [34] introduce the use of SVM as a classifier for the problem of weak/strong passwords. Training their algorithm on a 10,000-word dictionary of various categories, they yield a prediction accuracy of about 99% for a 10-fold cross validation using an RBF kernel, feeding the SVM with metrics such as the number of characters in each charset and presence of various patterns.

Dell'Amico et al. [21] condemns usual naive metrics for password checking and propose a checker based on dictionaries, probabilistic context-free grammars, and Markov chains, which represented the state-of-the-art in terms of password cracking at the time of the writing. They point out that it is impossible to infer resilience against advanced password-cracking techniques from bit-strength only, and hence limit themselves to express whether a given password is known to be crackable by these known techniques only. The complementarity of the techniques is highlighted, rather than finding one prevalent over the others. They evaluate the various techniques against 3 datasets.

By analyzing 70 million passwords from Yahoo! users, Bonneau [10] develops new metrics for guessing difficulty, considering that an attacker may be interested in compromising as much accounts as possible, instead of targeting a particular user. New measurements include the expected number of guesses per account to achieve a certain success rate, given that an attacker will guess passwords in an optimal way by testing them in a decreasing order of probability. By transforming guessing difficulty into bits of entropy/security, Bonneau found that the evaluated passwords provide between 10 bits of entropy against an optimal online attacker wishing to test 10 common passwords, to 20 bits against an optimal offline attacker wishing

to crack half of the accounts. Interesting results regarding password characteristics are also reported; e.g., users who changed their account password, or forgotten and reset it five times or more in the account's lifespan, are a little more likely to have stronger passwords.

Castelluccia et al. [16] leverage the use of Markov models to create an adaptive password-strength meter (APSM) for improved strength accuracy. Markov models have been introduced for password checking by Davies [20] in 1993, however with a different perspective. Strength is estimated by computing the probability of occurrence of the n-grams that compose a given password. The APSM design also addresses situations where the n-gram database of a given service is leaked. APSMs generate site-dependent strength outcomes, instead of relying on a global metric. The n-gram database is also updated with passwords from new users. To achieve good strength accuracy, APSMs should be used at websites with a large user base (e.g., at least 10,000).

A password checker and modifier based on a context-free grammar was proposed by Houshmand [31], which suggests calculating after how many trials the candidate password will be guessed if the attacker follows an optimal plan with context-free grammars-based cracking algorithm trained on publicly accessible password database leaks. Calculation of the rank is a constant-time operation, and is then mapped to a strength. Also, the algorithm modifies the structure of the password if it is too weak, and proposes a similar password which passes the strength threshold. Modifications follow a set of predefined rules picked at random, which may allow an attacker to determine how users strengthened their weak password, which can in turn be taken into account in the cracking process.

2.2.4 Cracking Efforts

Klein [37] gathers password files from colleagues at different places in US and UK, summing up to 13,797 Unix accounts, and tries to crack the passwords with many rules. He then proposes a password checker based on dictionary check and a list of

blacklisted patterns.

In its community-enhanced version, John the Ripper [50] offers a Markov cracking mode where statistics computed over a given dictionary are used to guide a simple brute-force attack; only the *most probable* passwords are tested. This mode is based on the assumption that "people can remember their passwords because there is a hidden Markov model in the way they are generated" [50]. In fact, this mode is an implementation of a 2005 proposal from Narayanan and Shmatikov [48], which predicts the most probable character to appear at a certain position, given the previous characters of a password. The Markov mode in John the Ripper is more suitable for offline password cracking than generating a dictionary for online checkers as it produces a very large number of candidate passwords (e.g., in the range of billions). Therefore, we did not consider using such dictionaries in our tests.

Dürmuth et al. [22] extend the work of Narayanan and Shmatikov [48] to improve cracking performance. They also explore the inclusion of users' personal information (e.g., username, birthday, list of friends, education, work, siblings, first name, last name, and location) in the cracking algorithm, and found that such personal attributes further enhanced the cracker's performance.

Weir et al. [68] propose an algorithm for extracting password structures from a given training dictionary (e.g., of leaked passwords), so as to infer which ones are the most common. Base structures are in the form of a sequence of charsets with associated lengths, e.g., L_3D_1 for 3 letters followed by 1 digit. In their proposed cracking algorithm, letter-only parts are filled by searching for a word of the required size in a dictionary, and other sets are filled in a decreasing order of probability based on the derived statistics from training dictionaries.

Concurrent to our work, Veras et al. [63] leverage Natural Language Processing (NLP) algorithms to analyze semantic patterns in leaked passwords. They found that most passwords in the RockYou dataset are semantically meaningful, containing terminologies related to love, sex, profanity, animals, alcohol and money. Their semantic-aware cracking technique shows significantly better results than existing

techniques, and may also be used as a password-strength checker.

Also concurrent to our work, Das et al. [19] study password reuse across various database leaks and phishing campaigns and build a cross-site password guessing algorithm based on rules extracted from the study, including deletion/insertion, capitalization, reversing, leet transformations, substring movement and subword modification. They are able to guess successfully 30% of password pairs based on substrings by 10 guesses, and 80% by 100 guesses.

Chapter 3

Meters Characterization

3.1 Overview

Password-strength meters are usually embedded in a registration or password update page. In password managers, they are used during the creation of a master password or to give hints to every forms a user fills out. During password creation, the meters instantly evaluate changes made to the password field or wait until the user finishes typing it completely, and output the strength of the given password. Below we discuss different aspects of these meters as reported in our test websites and password managers. Some common requirements and features of different meters are also summarized in Table 1.

(a) *Charset and length requirements.* By default, some checkers classify a given password as invalid or too short, until a minimum length requirement is met; most meters also enforce a maximum length. Some checkers require certain character sets (charsets) to be included. Commonly distinguished charsets include: lowercase letters, uppercase letters, digits, and symbols (also called special characters). Although symbols are not always considered in the same way by all checkers (e.g., only selected symbols are checked), we define symbols as being any printable characters other than the first three charsets. One particular symbol, the space character, may be disallowed altogether, allowed as external characters (at the start or end of a password),

or as internal characters. Some checkers also disallow identical consecutive characters (e.g., 3 or 4 characters for Apple and FedEx respectively).

(b) *Strength scales and labels.* Strength scales and labels used by different checkers also vary. For example, both Skype and PayPal have only 3 possible qualifications for the strength of a password (Weak-Fair-Strong and Poor-Medium-Good respectively), while Twitter has 6 (Too short-Obvious-Not secure enough-Could be more secure-Okay-Perfect). At the extreme side, LastPass has only a continuous progress bar without labels.

(c) *User information.* Some checkers take into account environment parameters related to the user, such as her real/account name or email address. We let these parameters remain blank during our automated tests, but manually checked different services by completing their registration forms with user-specific information. Ideally, a password that contains such information should be regarded as weak (or at least be penalized in the score calculation). However, password checkers we studied vary significantly on how they react to user information in a given password; more detail is provided in Chapter 4.

(d) *Types.* Based on where the evaluation is performed, we distinguish three main types of password checkers for web-based meters as follows. *Client-side*: the checker is fully loaded when the website is visited and checking is done locally only (e.g., Dropbox, Drupal, FedEx, Intel, Microsoft, QQ, Twitter, Yahoo! and 12306.cn); *server-side*: the checker is fully implemented on server-side (e.g., eBay, Google and Skype); and *hybrid*: a combination of both (e.g., Apple and PayPal). Password managers can have a variety of implementations but all operate on the user-end, either as a white- or black-box. We decided to group these managers together in terms of types. This distinction of four meter types leads us to different approaches to automate our testing, as explained in Section 3.2.

(e) *Diversity.* None of the 14 web services and 4 password managers we evaluated use a common meter. Instead, each service and software tool provides their own

meter, without any explanation of how the meter works, or how the strength parameters are assigned. For client-side and open-source checkers, we can learn about their design from code review, yet we still do not know how different parameters are chosen. Dropbox is the only exception, which has developed an apparently carefully-engineered algorithm called *zxcvbn* [69]. Dropbox also provides details of this meter and open-sourced it to encourage further development.

(f) *Entropy estimation and blacklists.* Every checker's implicit goal is to determine whether a given password can be easily found by an attacker. To this end, most employ a custom "entropy" calculator, either explicitly or not, based on the perceived complexity and length of the password. As discussed in Chapter 5, the notion of entropy as used by different checkers is far from being uniform, and certainly unrelated to Shannon entropy. Thus, we employ the term entropy in an informal manner, as interpreted by different meters. Password features generally considered for entropy-/score calculation by different checkers include: length, charsets used, and known patterns. Some checkers also compare a given password with a dictionary of common passwords (as a blacklist), and severely reduce their scoring if the password is blacklisted.

3.2 Test Automation

For our evaluation, we tested nearly four million of passwords against each of the 21 checkers. In this section, we discuss how we performed such large-scale automated tests.

3.2.1 Web-Based Client-Side Evaluators

For client-side checkers, we extract the relevant JavaScript functions from the source code of a registration page, and query them to get the strength for each dictionary password with a dictionary-attack-based algorithm: a dictionary is taken as input

Type	Service	Strength scale	Length limits		Charset required	Mono-tonicity	User info	Space acceptance		Enforcement
			Min	Max				External	Internal	
Web-based client-side	Dropbox	Very weak, Weak, So-so, Good, Great	6	72	∅	No	◐	✓	✓	∅
	Drupal	Weak, Fair, Good, Strong	6	128	∅	Yes	∅[1]	×	✓	∅
	FedEx	Very weak, Weak, Medium, Strong, Very strong	8	35	1+ lower, 1+ upper, 1+ digit	Yes	∅	×	×	Medium
	Intel	Oh no!, Congratulations	1	–	∅	No	∅	✓	✓	∅
	Microsoft	Weak, Medium, Strong, Best	1	–	∅	Yes	∅	✓	✓	∅
	QQ	Weak, Moderate, Strong	6	16	∅	Yes	∅	×	×	∅
	Twitter	Invalid/Too short, Obvious, Not secure enough (NSE), Could be more secure (CMS), Okay, Perfect	6	>1000	∅	No	◐	✓	✓	CMS
	Yahoo!	Too short, Weak, Strong, Very strong	6	32	∅	Yes	◐	✓	✓	Weak
	12306.cn	Dangerous, Average, Secure	6	25	1+ charset[2,3]	Yes	∅	×	×	∅
Web-based server-side	eBay	Invalid, Weak, Medium, Strong	6	20	any 2 charsets	Yes	◐	×	✓	∅
	Google	Weak, Fair, Good, Strong	8	100	∅	No	∅	×	✓	Fair
	Skype	Poor, Medium, Good	6	20	2 charsets or upper only	Yes	∅	×	×	Medium
Web-based hybrid	Apple	Weak, Moderate, Strong	8	32	1+ lower, 1+ upper, 1+ digit	No	◐	×	×	Medium
	PayPal	Weak, Fair, Strong	8	20	any 2 charsets[2]	No	∅	×	×	Fair
Application-based	1Password	Terrible, Weak, Fair, Good, Excellent, Fantastic	1	–	∅	Yes/No[4]	∅	✓	✓	∅
	KeePass	0-128 bits	1	–	∅	No	∅	✓	✓	∅
	LastPass	0–100%	1	–	∅	No	◐	✓	✓	∅
	RoboForm	Weak, Good, Excellent	6	49	∅	No	∅	✓	✓	∅

[1] Partially covered in the latest beta version (ver 8), as of Nov. 28, 2013
[2] PayPal and 12306.cn count uppercase and lowercase letters as a single charset
[3] 12306.cn only considers the underscore as a symbol
[4] 1Password has two meters: one is monotonic, the other is not

Table 1: Password requirements and characteristics of the evaluated meters; see Section 3.1 for details. Notation used under "Monotonicity": Represents whether any additional character leads only to better scoring (not accounting for user information check). "User info": ∅ (no user information is used for strength check), and ◐ (some user information is used or all but not fully taken into account). Under "Charset required", we use 1+ to denote "one or more" characters of a given type. The "Enforcement" column represents the minimum strength required by each checker for registration completion: ∅ (no enforcement); and other labels as defined under "Strength scale".

and each password is checked against the evaluation function that returns a score (e.g., “weak”, “strong”). Outputs are then stored for later analysis. To identify the sometimes obfuscated part of the code responsible for checking the password, we use the built-in debugger in Google Chrome. In particular, we set breakpoints on DOM changes, i.e., when the password meter’s bar or qualification word changes, allowing us to look at the stack at that time of the execution and identify relevant functions and set further breakpoints on the code. Such obfuscation may be the result of code minification, which involves the removal of unnecessary data in a source code such as comments, extra spaces, optional writings, along with the shrinking of variable names in hope to make the code shorter and faster to be transmitted on the network. It has the side effect of obfuscating the original source. Fortunately, strength meters generally involve simple logic, and remain understandable with some effort after such optimization. The use of various APIs for developing the client-side code can also result in complex code. Also, as some part of the checkers are event-driven (e.g., invoked by key-press events), the use of a debugger is even more relevant as it is possible to land directly on the evaluation function without reading all the client-side code and analyzing all possible events. We tested our dictionaries using Mozilla Firefox, as it was capable of handling bigger dictionaries without crashing (unlike Google Chrome). Speed of testing varies from 7ms for a 500-word dictionary against a simple meter (FedEx), to nearly 10min for a 2-million dictionary against the most complex meter (Dropbox).

3.2.2 Web-Based Server-Side Evaluators

Server-side checkers directly send the password to a server-side script by an AJAX request without checking them locally (except for minimum length). We test server-side checkers using a PHP script with the cURL library[4] for handling HTTPS requests to the server-side checker. The chercker’s URL is obtained from the JavaScript code and/or a network capture. We use Google Chrome to set breakpoints on AJAX calls

[4] `http://curl.haxx.se`

to be pointed to the `send()`[5] call before its execution. This way, it is possible for us to inspect the stack and deduce how the call parameters are marshaled. We then prepare our password test requests as such and send them in batches.

To reduce the impact of our large volume of requests to the server-side checkers (both in terms of bandwidth and processing), we leverage keep-alive connections, where requests are pipelined through the same established connection for as long as the server supports it. Typically, we tested more than 4 million passwords for each service (the small overlap between dictionaries was not stripped), and we could request up to about 1000 password through one connection with Skype, 1500 with eBay and as much as we wanted with Google; as a result, the number of connections dropped significantly. We did not parallelize the requests to keep a low profile from the servers' perspective. However, the obvious price we paid is that our tests took longer to finish. On average, we tested our dictionaries at a speed of 5 passwords per second against Skype, 10 against eBay and 64 against Google (2.5 against PayPal and 8 against Apple for the server-side part of their checkers), generating a maximum traffic of 5KiB/s of upload and 10KiB/s of download per web service. To our surprise, we did not face any blocking mechanisms during our tests.

3.2.3 Web-Based Hybrid Evaluators

Hybrid checkers first perform a local check, and then resort to a server-side checker (i.e., a dynamic blacklist of passwords and associated rules). We combine above mentioned techniques to identify client-side and server-side parts of the checker. Our test script runs as a local webpage, invoking the extracted client-side JavaScript checker. When the checker wants to launch a request to a remote host, which inherently comes from a different origin, we face restrictions imposed by the same origin policy.[6].

To allow client-side cross-origin requests, the cross-origin resource sharing mechanism (CORS [70]) has been introduced and is currently implemented in most browsers.

[5] `http://www.w3.org/TR/XMLHttpRequest/#the-send()-method`
[6] `http://www.w3.org/Security/wiki/Same_Origin_Policy`

CORS offers new request/response headers that may be used by a server to specify additional origins that are allowed to access resources from the server. As the original servers are not supposed to be requested for their service by another origin than the original website itself, no cross-domain policies are provided. Thus, the JavaScript checker is prohibited from receiving the response when it wants to perform such a request. To allow our local script as a valid origin, we implemented a simple proxy to insert the required CORS header, `Access-Control-Allow-Origin` [70], in the server's response.

Our local proxy is problematic for keep-alive connections, as it breaks the direct connection between the JavaScript code and the remote server. Technically, the client-side code launches a request to our proxy, which then forwards the request to the remote server. Sharing a keep-alive remote connection across multiple client-side calls is more challenging. Indeed, if the proxy is implemented as a PHP script running under an independent HTTP server, we would not be able to easily reuse the server connection; in this case, each AJAX call to our script would trigger a new instance of the script, which in turns would create a new server connection. Instead, we implemented a simple HTTP server within the PHP script of the proxy that allows server connection reuse across multiple client requests. The HTTP server part waits for incoming connections and reads requests on which only basic parsing occurs. We also chose to reuse the XMLHttpRequest object to pipeline requests to our proxy from the browser. In this configuration, we use a single connection between the JavaScript code and the proxy, and we use the same pipelining mechanism as for the server-side checkers between the proxy and the remote server. Finally, because we faced browser crashing for large dictionaries tested against hybrid checkers, we needed to split these dictionaries into smaller parts and to test them again separately. To prevent duplicate blacklist checks against the server-side checker (as we restart the test after a browser crash), we implement a cache in our proxy which also speeds up the resume process.

3.2.4 Application-Based Evaluators

To cope with a multitude of passwords, password manager applications are also used widely. Some of these managers implement a meter; similar to web-service meters, these meters also vary significantly in their design. We choose four popular password managers that implement strength meters as part of a closed- or open-source software tool and/or as JavaScript browser extension.

Open-source meters. Open-source password managers in our evaluation are offered as a C# application (KeePass) or JavaScript-based browser extensions (LastPass and one of 1Password's meters). In both cases, analyzing the algorithm and automating the test of our dictionaries are straightforward. We modify KeePass to process an entire dictionary instead of a typed password and evaluate our 11 dictionaries within two minutes. Tests against the JavaScript-based browser extensions' checkers are similar to the ones for web-based client-side checkers, and yield comparable speed.

Closed-source meters. We tested two closed-sourced software-based meters (1Password and RoboForm). We evaluate the Windows versions of these checkers. Two choices are available for us to analyze them.

Simulating human typing. The first method consists in automating the evaluation of our dictionaries by simulating a user who types passwords and collecting back the results for further analysis. We leverage AutoIt[7] to automate such tests. AutoIt is a free software interpreter with a custom scripting language designed for automating the Windows GUI. The overall algorithm to automate our tests against closed-source meters is presented in Algorithm 1. 1Password meter consists of a continuous progress bar showing a score rounded to the nearest integer. As it is using a default Windows progress bar, AutoIt is able to retrieve the score directly. However, in the case of RoboForm, the meter has only five possible positions and involves a custom graphical meter, which cannot be processed directly by AutoIt to extract values. We are able to retrieve the color of selected pixels on the screen with AutoIt at five different locations

[7] `http://www.autoitscript.com/site/autoit/`

on the meter so as to collect and reconstitute the meter's output. As the meter is colored from red to green on a gray background, we know that if a pixel located at the ith position is not gray, the meter is filled at least up to this position. By checking the five points, we can identify up to which position the meter is filled. The method, although ad-hoc and slower than a direct access to the meter's value, proves to be effective. We run the tests in a virtual machine so that the password manager window can stay open in front for the whole duration of the tests without interference with our use of the computer. We also wait 5 milliseconds for RoboForm to compute and display the result before capturing pixels color on the meter bar. We verified that this duration is long enough for RoboForm to complete the evaluation, while keeping our evaluation time under a week. Tests are performed at a speed of 90 passwords per second against 1Password and 8.5 against RoboForm. Finally, the analysis of the output for 1Password is discussed in Section 4.4. RoboForm's algorithm is too complex to understand from its output only, hence we rely on the second solution for characterizing this meter.

Reverse-engineering. We can leverage a debugger attached to the application to read through the program's low-level instructions and infer its high-level behavior. We must capture a trace of the execution, or follow step-by-step the execution when the application is actively ranking a given password. Unfortunately, such a method is cumbersome as the variety of operations performed increases the length and complexity of the instructions to analyze. In our initial exploration of this solution, we faced difficulties capturing the trace of execution using IDA[8] because of software exceptions that make the debugger loose control over the application. After several unsuccessful attempts breaking the execution of RoboForm, we were successful at attaching Immunity Debugger[9] to it and setting a breakpoint on any access to the memory location that contains the user password to be ranked. Once the algorithm reads this memory location, we can follow the algorithm and understand how the

[8]https://www.hex-rays.com/products/ida/
[9]http://debugger.immunityinc.com/

Algorithm 1 AutoIt high-level test automation algorithm

Input: Window title of the meter, input dictionary
Output: Output of the meter against each word in the input dictionary
1: $hPassword \leftarrow$ handle of the input caption where to enter the password
2: $hLabel \leftarrow$ handle of the caption showing a word qualifying the strength of the password
3: $hMeter \leftarrow$ handle of the progress bar showing the strength of the password
4:
5: **for each** $password \in dictionary$ **do**
6: Insert $password$ into $hPassword$
7: Wait for processing (0 or 5ms depending on the checker tested)
8: $score \leftarrow$ progress bar status of $hMeter$
9: $strength \leftarrow$ word in $hLabel$
10: Write $password$, $score$, $strength$ as output
11: **end for**

score is calculated. Details of this analysis are given in Section 4.4.4.

3.3 Tested Dictionaries

Below, we provide details of the password dictionaries used in our evaluation.

Overview and notes. Table 2 lists the 11 dictionaries we used, including their sizes, and maximum, average and standard deviation of their password length. Dictionary sources include: password cracking tools (John the Ripper and Cain & Abel), a list of 500 most commonly used passwords (Top500), an embedded dictionary in the Conficker worm, and leaked databases of plaintext or hashed passwords (RockYou and phpBB). We mostly chose simple and well-known dictionaries (as opposed to more complex ones, see e.g., [2, 3]), to evaluate checkers against passwords that are reportedly used by many users. We expected passwords from these non-targeted dictionaries would be mostly rejected (or rated as weak) by the meters.

We trimmed passwords used from the leaked databases by removing leading and trailing spaces, as several checkers disallow such external spaces (see Table 1); however, we kept internal spaces. Four additional dictionaries are derived by using well-known password mangling rules. As we noticed that our main dictionaries did not

Dictionary name	Size (# words)	Password length		
		Max	Average	Std dev.
Top500	499	8	6.00	1.10
Cfkr	181	13	6.79	1.47
JtR	3,545	13	6.22	1.40
C&A	306,706	24	9.27	2.77
RY5	562,987	49	7.41	1.64
phpBB	184,389	32	7.54	1.75
Top500+M	22,520	12	7.18	1.47
Cfkr+M	4,696	16	7.88	1.78
JtR+M	145,820	16	7.30	1.66
RY5+M	2,173,963	39	8.23	1.98
Leet	648,116	20	9.09	1.81

Table 2: Dictionaries used against password checkers; +M represents mangled version of a dictionary; the "Leet" dictionary is custom-built by us.

specifically consider leet transformations, we built a special leet dictionary using the main dictionaries.

As a side note, many passwords in the source dictionaries are related to insults, love and sex, as reported by a semantic analysis by Veras [63]. We avoid mentioning such words as example passwords. We also noticed poor internationalization of dictionary passwords in general, where most of them originate from English. One exception is the RockYou dictionary, which contains some Spanish words (possibly due to some RockYou users being originated from Spanish-speaking countries). Finally, some leaked passwords contained UTF-8-encoded words that were generally not handled properly by the checkers (also some are malformed UTF-8 strings) [12]. Given their small number in our reduced version of RockYou dictionary, we chose to ignore them.

Dictionaries sometimes overlap, especially when considering the inclusion of trivial lists and cracking tools default dictionaries among leaked passwords, see Table 3. More comments are provided in Section 5.1.

	Top500	Cfkr	JtR	C&A	RY5	phpBB	T500+M	Cfkr+M	JtR+M	RY5+M	Leet	1Passwd	DropBox	FedEx	Intel	KeePass	MS v1	RoboForm	Twitter
Top500	-	6.61	84.37	89.18	99	95.99	0	0	2.2	0	0	54.71	99.6	16.43	99.6	99.8	27.45	91.78	75.35
Cfkr	18.23	-	45.86	46.41	93.92	86.74	2.76	0	3.87	1.66	0	25.97	77.9	14.92	70.72	72.93	18.78	25.41	17.68
JtR	11.88	2.34	-	73.34	95.99	85.08	6.21	0.59	0	0.82	0	40.25	83.92	7.22	75.68	78.96	15.12	41.64	9.68
C&A	0.15	0.03	0.85	-	8.59	4.03	0.06	~0	0.18	0.23	0	74.11	9.89	0.18	1.87	1.92	0.72	3.44	0.11
RY5	0.09	0.03	0.6	4.68	-	7.73	1.22	0.11	4.86	0	~0	3.08	5.96	0.09	1.6	1.63	0.26	2.08	0.07
phpBB	0.26	0.09	1.64	6.7	23.61	-	0.74	0.11	2.07	2.19	~0	4.33	8.09	0.23	4.05	4.12	0.57	2.63	0.21
T500+M	0	0.02	0.98	0.75	30.44	6.09	-	4.4	83.84	19.25	0	0.39	2.88	0.02	2.14	2.15	0.04	0.2	~0
Cfkr+M	0	0	0.45	0.32	12.73	4.43	21.08	-	43.19	15.82	0	0.17	1.75	0.02	0.72	0.75	0.02	0.13	0
JtR+M	0.01	~0	0	0.38	18.75	2.61	12.95	1.39	-	17.99	0.01	0.31	1.17	0.01	0.54	0.54	0.01	0.1	0.01
RY5+M	0	~0	~0	0.03	0	0.19	0.2	0.03	1.21	-	0.01	0.02	0.11	~0	~0	~0	~0	0.01	0
Leet	0	0	0	0	~0	~0	0	0	~0	0.03	-	0	0	0	0	0	0	0	0
1Passwd	0.12	0.02	0.63	99.58	7.6	3.5	0.04	~0	0.2	0.23	0	-	9.12	0.17	1.65	1.69	0.65	1.34	0.11
DropBox	0.58	0.17	3.5	35.66	39.43	17.53	0.76	0.1	2.01	2.75	0	24.47	-	0.58	11.73	11.89	2	13.97	0.46
FedEx	14.49	4.77	45.23	98.76	89.05	75.09	0.88	0.18	1.94	0.53	0	68.02	86.75	-	63.96	72.97	96.11	94.88	14.49
Intel	4.98	1.28	26.87	57.52	89.95	74.75	4.82	0.34	7.82	0.41	0	37.61	99.97	3.63	-	99.99	8.31	28.48	3.89
KeePass	4.89	1.3	27.49	57.81	89.93	74.57	4.75	0.34	7.68	0.41	0	37.86	99.37	4.06	98.06	-	8.76	28.68	3.84
MS v1	6.08	1.51	23.78	98.58	65.22	46.98	0.35	0.04	0.71	0.22	0	65.35	75.51	24.13	36.82	39.57	-	44.94	4.79
RoboForm	1.57	0.16	5.08	36.32	40.23	16.65	0.15	0.02	0.52	0.73	0	10.48	40.87	1.85	9.78	10.04	3.48	-	1.25
Twitter	93.77	7.98	85.54	87.03	98	95.01	0.25	0	2.74	0	0	60.85	97.51	20.45	96.76	97.51	26.93	90.52	-

~0 means less than 0.01%

Table 3: Dictionary overlaps shown in percentage relative to the size of the dictionary from the left-most column, e.g., 95.99% of Top500 is included in phpBB. Checkers' embedded dictionaries overlaps are also represented.

3.3.1 Cracking Tool Dictionaries

Top500. This dictionary was released in 2005 as the "Top 500 Worst Passwords of All Time" [13], and later revised as Top 10000 [14] passwords in 2011. We use the 500-word version as a very basic dictionary. Passwords such as *123456*, *password*, *qwerty* and *master* can be found in it. Actually, a "0" is duplicated in this list, making it have only 499 unique passwords. Password composition: 91% lowercase letters only, 7% digits only, and 2% lowercase letters and digits.

Cfkr. The dictionary embedded in Conficker worm was used to try to access other machines in the local network and spread the infection. Simple words and numeric sequences are mostly used; example passwords include: *computer*, *123123*, and *mypassword*.[10] Password composition: 52.5% lowercase letters only, 29.8% digits only, 15.5% lowercase letters and digits, and 2.2% mixed-case letters.

JtR. John the Ripper [50] is a very common password cracker that comes with a dictionary of 3,546 passwords, from which we removed an empty one. Simple words can be found in this dictionary too; however, they are little more complex than those in Top500, e.g., *trustno1*. Password composition: 82.79% lowercase letters only, 8.12% lowercase letters and digits, 4.4% mixed-case letters, and few other charset combinations.

C&A. Another password cracking tool, Cain & Abel [51] comes with a 306,706-word dictionary that primarily consists of long lowercase words (e.g., *constantness*, *psychotechnological*). The composition of passwords is quite unique: 99.84% lowercase letters only. The rest is shared among lowercase letters and symbols (0.09%), lowercase letters and digits (0.05%), few digits only, symbols only, and lowercase letters with digits and symbols.

[10] http://www.f-secure.com/v-descs/worm_w32_downadup_al.shtml

3.3.2 Real Password Database Leaks

RY5. RockYou.com is a gaming website that was subject to an SQL injection attack in 2009, resulting in the leak of 32.6 million cleartext user passwords. This constitutes one of the largest real user-chosen password databases as of today. There are only 14.3 million unique passwords, which is still quite large for our tests. We kept only the passwords that were used at least 5 times, removed space-only passwords (7) and duplicates arising from trimming (5). The resulting dictionary has 562,987 words, of which 39.96% are lowercase letters only, 36.39% lowercase letters and digits, 17.11% digits only, 1.93% uppercase letters only, and the rest consists of several charset combinations.

phpBB. The phpBB.com forum was compromised in 2009 due to an old vulnerable third-party application, and the database containing the hashed passwords was leaked and mostly cracked afterwards. Due to the technical background of users registered on this website, passwords tend to be a little more sophisticated than trivial dictionaries. Password composition: 41.24% lowercase letters only, 35.7% lowercase letters and digits, 11.24% digits only, 4.82% mixed-cased letters and digits, 2.68% mixed-case letters, and the rest is made of different charset combinations.

3.3.3 Mangled Dictionaries

Users tend to modify a simple word by adding a digit or symbol (often at the end), or changing a letter to uppercase (often the first one), sometimes due to policy restrictions [16, 38, 15]; for details on this wide-spread behavior, see e.g., Weir [66]. Password crackers accommodate such user behavior through the use of *mangling* rules. These rules apply different transformations such as capitalizing a word, prefixing and suffixing with digits or symbols, reversing the word, and some combinations of them. For example, *password* can be transformed into *Password*, *Password1*, *passwords* and even *Drowssap*. John the Ripper comes with several mangling rules (25 in the wordlist mode), which can produce up to about 50 passwords from a single one.

We applied John the Ripper's default ruleset (in the wordlist mode) on Top500, Cfkr, and JtR dictionaries, generating an average of 45, 26 and 41 passwords from each password in these dictionaries, respectively. Derived dictionaries are called Top500+M, Cfkr+M, JtR+M respectively. Original passwords with digits or symbols are excluded by most rules, unless otherwise specified. We chose not to test the mangled version of C&A as it consists of 14.7 million passwords (too large for our tests). Given that the original size of RY5 is already half a million passwords, mangling it with the full ruleset would be similarly impractical. For this dictionary, we applied only the 10 most common rules, as ordered in the ruleset and simplified them to avoid redundancy. For example, instead of adding all possible leading digits, we restricted this variation to adding only "1". We did the same for symbols. The resulting dictionary is called RY5+M. The rules applied for RY5 mangling are the following:

1. lowercase passwords that are not;
2. capitalize;
3. pluralize;
4. suffix with "1";
5. combine (a) and (d);
6. duplicate short words (6 characters or less);
7. reverse the word;
8. prefix with "1";
9. uppercase alphanumerical passwords; and
10. suffix with "!".

Note that although these rules are close to real users' behavior, they are compiled mostly in an ad-hoc manner (see e.g., Weir [66]). For example, reversing a word is not common in practice, based on Weir's analysis of leaked password databases. At least, John the Ripper's rules represent what an average attacker is empowered with.

3.3.4 Leet Transformations

Leet is an alphabet based on visual equivalence between letters and digits (or symbols). For example, the letter *E* is close to a reversed *3*, and *S* is close to a *5* or *$*. Such transformations allow users to continue using simple words as passwords, yet covering more charsets and easily bypass policy restrictions [55]. Leet transformations are not covered in our main dictionaries, apart from few exceptions; thus, we built our own leet transformed dictionary to test the effect of such transformations.

Our Leet dictionary is based on the passwords from Top500, Cfkr, JtR, C&A, phpBB, the full RockYou dictionary, along with the Top 10000 dictionary, and a 37,141-word version of the leaked MySpace password dictionary,[11] obtaining 1,007,749 unique passwords. For each of them, we first strip the leading and trailing digits and symbols, and then convert it to lowercase (e.g., *1PassWord$0* becomes *password*). Passwords that still contain digits or symbols are then dropped (e.g., *here4you*), so as to keep letter-only passwords. Passwords that are less than 6-character long are also dropped, while 6-character long ones are suffixed with a digit and a symbol chosen at random, and 7-character passwords are only suffixed with either a digit or a symbol. At this point, all passwords are at least 8-character long, allowing us to pass all minimum length requirements. Those longer than 20 characters are also discarded. The dictionary was reduced to 648,116 words.

We then apply leet transformations starting from the end of each password to mimic known user behaviors of selecting digits and symbols towards the end of a password (see e.g., [66, 29]). For these transformations, we also use a translation map that combines leet rules from Dropbox and Microsoft checkers. Password characters are transformed using this map (if possible), by choosing at random when multiple variations exist for the same character, and up to three transformations per password. Thus, one leet password is generated from each password, and only single character equivalents are considered (e.g., we do not consider more complex transformations such as `V` becoming double slashes: \/). The resulting Leet dictionary is composed of

[11] Collected from: `http://www.skullsecurity.org/wiki/index.php/Passwords`

77.56% 4-charset passwords, 18.66% mixed-case letters and digits, 3.72% mixed-case letters and symbols, and the rest of mixed-case letters only. Arguably, this dictionary is not exhaustive; however, our goal is to check how meters react against simple leet transformations. The near-zero overlap between this dictionary and the leaked ones (as in Table 3) can be explained by the simple password policies as used by RockYou and phpBB at the time of the leaks. RockYou required only a 5-character password and even disallowed symbol characters [29], while phpBB's 9th most popular password is *1234*, clearly indicating a lax password policy. Thus, users did not need to come up with strategies such as mangling and leet transformations.

Chapter 4

Empirical Evaluation

For each password-strength meter evaluated, we present their general behavior and response profile against our set of dictionaries, analyze the way they operate and discuss their strengths and weaknesses. Most of our tests of web-based meters have been performed between the months of June and July, 2013. The rest of web-based and application-based meters tests have been performed between February and March, 2014.

4.1 Web-Based Password Meters (Client-Side)

4.1.1 Dropbox

Dropbox has developed a client-side password strength checker called *zxcvbn* [69], and open-sourced it to encourage others to use and improve the checker. Figure 1 summarizes our results for Dropbox.

4.1.1.1 Algorithm

Zxcvbn decomposes a given password into patterns with possible overlaps, and then assigns each pattern an estimated "entropy". The final password entropy is calculated as the sum of its constituent patterns' entropy estimates. The algorithm detects

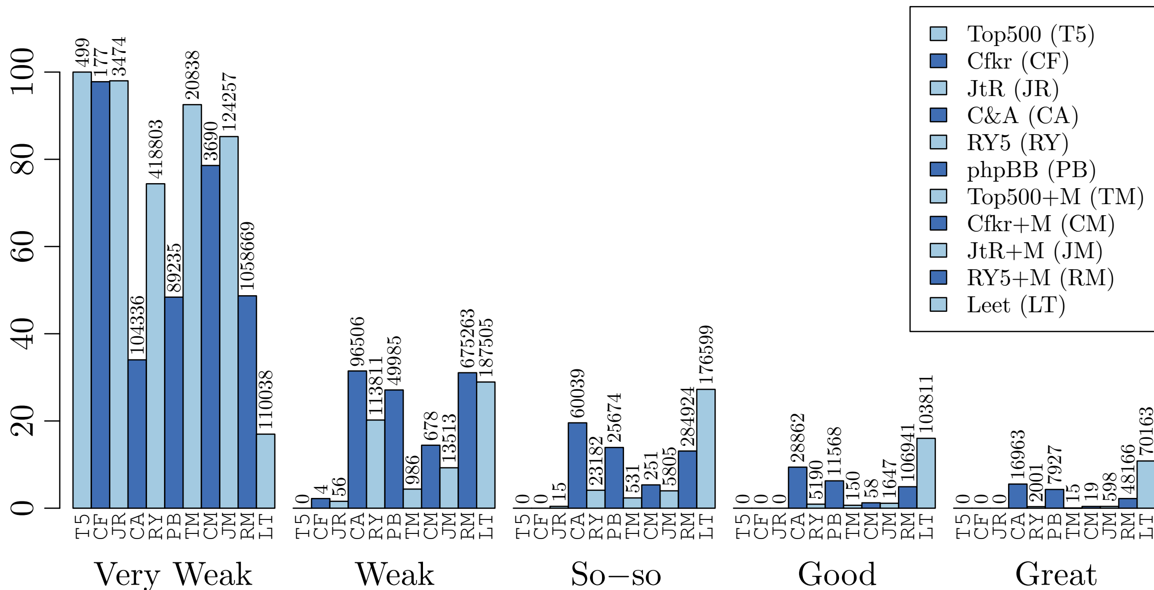


Figure 1: Dropbox checker password strength distribution

multiple ways of decomposing a password, but keeps only the lowest of all possible entropy summations as an underestimate. An interesting aspect of this algorithm is the process of assigning entropy estimates to a pattern. The following patterns are considered: spatial combinations on a keyboard (e.g., qwerty, zxcvbn, qazxsw); repeated and common semantic patterns (e.g., dates, years); and natural character sequences (e.g., 123, gfedcba). These patterns are considered weak altogether, and given a low entropy count.

To restrict common passwords, a candidate password is checked against five embedded, frequency-ordered dictionaries: 7141 passwords from the Top 10000 password dictionary [14]; 32545 English words from the Wikitionary project;[12] 1004 male, 3815 female names and 40583 surnames from the 2000 US Census. The presence of a subpart of the password in a dictionary is not forbidden or directly penalized in contrast to other password checkers involving a blacklist in our test. Such a subpart is assigned an entropy value based on the average number of trials that an attacker would have to perform, considering the rank of the subpart in its dictionary.

If no pattern is found for a given subpart of the password, it is considered a

[12] `http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists`

random string. The entropy of such string is computed based on a simple brute-force attack. For example, *MySuperP4$$w0Rd* is decomposed as *my* (5.3 bits of entropy based on a dictionary attack), *super* (9.4 bits), and a transformed *password* (about 5 bits; 1.58 for the leet transformation and 3.45 for the uppercase), which are all found quickly in the dictionaries; hence this password gets a "very weak" score (entropy of 19.8 bits). On the contrary, the partially randomly generated password *P4$$w0RdTBuK9Ye6MZkdyx* decomposes as a transformed *password* and something that does not match a dictionary word or pattern; hence, the latter part is considered to be found only by a brute-force attack (92 bits of entropy), granting the password a score labeled "great".

Finally, the entropy is matched to a score by supposing that a guess would take 0.01 second, and that an attacker can distribute the load on 100 computers. Then, the average time needed (in seconds) is computed as: $2^{entropy-1} \cdot 0.01/100$. Thresholds are applied to map the average cracking time to a strength in the following way: very weak if less than 10^2 seconds, weak if less than 10^4, so-so if less than 10^6, good if less than 10^8, and great otherwise.

4.1.1.2 User information

The first and last names and the email address of a registering user are added to a new dictionary in the algorithm to weaken the strength of a password containing these user-specific registration items. The part of the password that matches any registration item, even if transformed, is assigned a very low entropy. It is given 1 bit if it matches the first name, 1.58 for the last name, and 2 for the email address (because of the rank in the dictionary). Additional bits are assigned for transformations and uppercased letters. Overall, a password reusing a registration item will be significantly weakened. However, there is a programming error that allows most users to bypass this filter, even unknowingly. If an item contains an uppercase letter (a probable case for the names), it will not be detected as a pattern, and thus a password with such items will not be weakened accordingly. Even though this bug has been reported and

fixed one year ago on zxcvbn GitHub project,[13] the change has not been propagated to DropBox's registration page.

4.1.1.3 Strengths

Zxcvbn considers the composition of a password more thoroughly than all other checkers in our test, resulting into a more realistic evaluation of the complexity of a given password. In this regard, it is probably the best checker. Zxcvbn also assigns good scores to a password composed of multiple words, based on the assumption that having several words together, even when taken from a known dictionary, generally yields stronger passwords than (transformed) single-word passwords.

4.1.1.4 Weaknesses

The simple transformation of reversing the character order in a word (as found in John the Ripper's default mangling rules), often generates "Great!" passwords out of very simple dictionary words. For instance, *ehcsroP* (the reverse of *Porsche*) or *retupmoC* (the reverse of *Computer*) mangled from words found in the Top500 dictionary are qualified as great. In addition, zxcvbn dictionaries include only English words, and thus are unable to catch commonly used words in other languages; e.g., *Motdepasse* is the French equivalent of *Password* and is considered as great, and *contraseñas* (Spanish equivalent in lowercase) is good. As for the keyboard combinations, some design limitations fail to catch patterns such as *1a2s3d4f5g*, tagged as great.

As zxcvbn promotes the use of passwords consisting of a combination of common words such as *correcthorsebatterystaple*,[14] many words from C&A are considered as good (9.4%), or even great (5.5%), as they mostly consist of long words (often a combination of simple words). Zxcvbn, however, fails for some trivial passwords that are not listed in its internal dictionary, e.g. from RY5, *evanescence*, *SEPULTURA* (an American rock band and a Brazilian heavy metal band, respectively) and

[13] https://github.com/lowe/zxcvbn/commit/a9fa79e62adcf171f5f981d2f64801df1ebdd990
[14] https://www.xkcd.com/936/

dolce&gabana (an Italian luxury industry fashion house) are considered great. This highlights an important limitation of fixed embedded dictionaries, common to all meters we evaluated. Finally, it is interesting to note that even though Dropbox made an effort to build a better-than-average password-strength meter, they do not enforce a minimum strength during the registration process, letting users register with a possibly very weak password.

4.1.2 Drupal

Drupal is an open-source framework for building content management systems (CMS). It is the third mostly used CMS,[15] behind Joomla and WordPress [65] (as of Dec. 6, 2013). Drupal version 7.x uses a simple client-side checker based on a decreasing scoring system (Joomla and WordPress currently do not use a checker). Figure 2 summarizes our results.

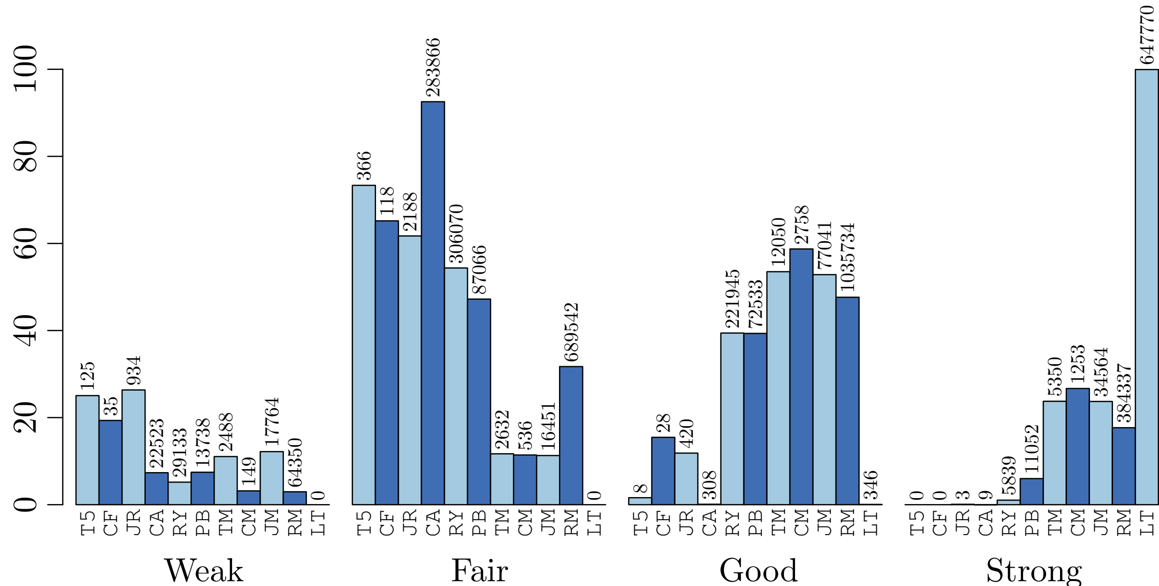


Figure 2: Drupal checker password strength distribution

[15]Example sites using Drupal include: `usenix.org`, `whitehouse.gov`.

4.1.2.1 Algorithm

First, passwords below six characters suffer a significant linear penalty with respect to the number of missing characters. A 6-character password passes the minimum length requirement and is at least considered fair. Then, the score non-linearly decreases for each charset that has no member in the given password. Starting from an initial value of 100, missing one charset reduces the score by 12.5, two by 25, three and four by 40. Thresholds are applied on the score to yield one of the four strength categories: under 60 is weak, 60–69 is fair, 70–79 is good, and above 80 is strong. The penalizing mechanism for passwords shorter than six characters prevents them from reaching a score higher than fair, even if they are composed of all charsets. Covering two charsets with a password of at least six characters ensures a good strength, while covering three charsets ensures a strong strength. Examples include: *Aa1+* is fair, *passw1* is good, *Passw1* and *AAaa1+* are strong. The beta version 8 (as of Nov. 28, 2013) introduces an additional check to prevent a user from choosing a password that is the same as the username. This restriction was actually implemented but commented out in version 7.x.

4.1.2.2 Weaknesses

No further checks are done such as searching for repetitive patterns, weak or common words, nor is any reward given to the length beyond six characters. A password such as *correcthorsebatterystaple* is thus just fair. Given the simplistic design of this checker, it is unsurprising to see that most common dictionary words are at least considered fair; C&A words, are also mostly considered as fair; and mangled dictionary words reach good and strong ratings fairly easily.

4.1.3 FedEx

FedEx allows its users to register an online account to facilitate simple services such as shipment handling and tracking. Interestingly, FedEx's password checker is quite

stringent even though FedEx hosts arguably far less sensitive information compared to other services in our evaluation; see Figure 3 for summary results.

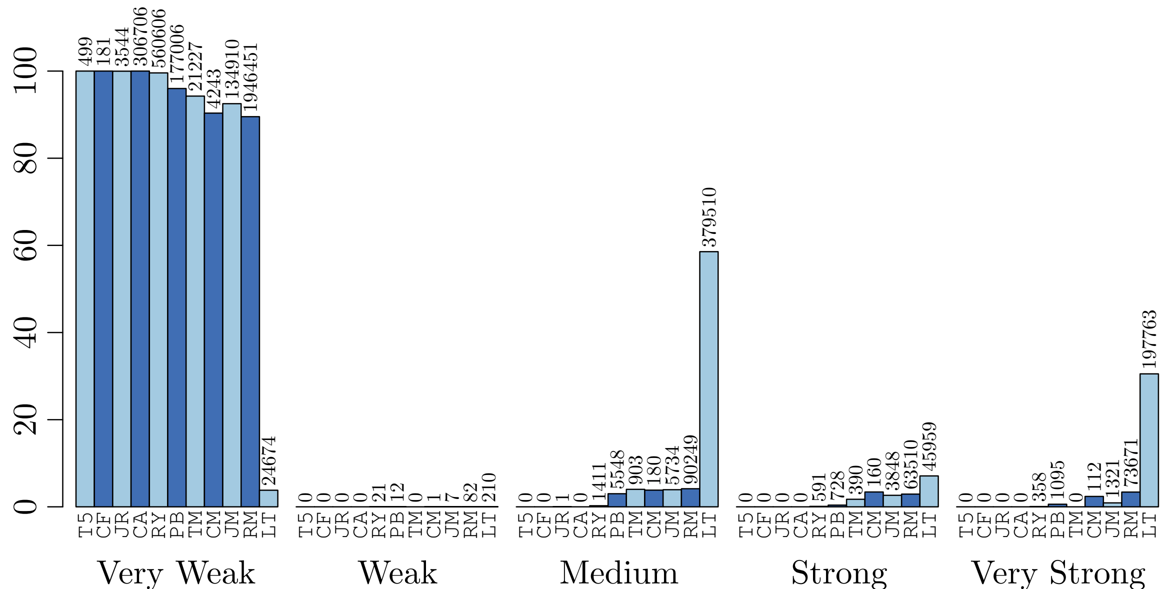


Figure 3: FedEx checker password strength distribution

4.1.3.1 Algorithm

The client-side checker is a simple 130-line JavaScript program with an embedded 566-word dictionary. Passwords matching any dictionary word are labeled as weak. The checker is aware of leet transformations, and normalizes a given password by reversing the transformation (if found). The password is also converted to lowercase before comparing it against the dictionary. These rules significantly weaken many passwords such as *P@$5W0rD*. Each strength level has specific rules to be matched. A password is very weak until it passes the basic requirements: 8 characters in length, the use of characters from three charsets (lowercase, uppercase and digit), and no three identical consecutive characters. The password can reach a medium strength if it has at least 4 unique characters and is not in the dictionary. The password becomes strong when it has 9 characters or more, in addition to at least 6 unique characters. Very strong passwords must be at least 10 characters long and have at least 6 unique

characters; or 9 characters long, 6 unique characters and have at least one symbol. Only the characters `!@#$%^&*?,~` are considered in the symbols set (remaining other characters do not influence the strength but are counted in the password length). In contrast to other checkers tested, the password strength is not shown as a text label but only as a colored meter.

4.1.3.2 Strengths

The apparently simple algorithm can actually catch the majority of dictionary passwords in our test. Only a few passwords from leaked dictionaries are rated as medium or higher (0.42% for RY5 and 3.4% for phpBB). However, this achievement is mostly due to stringent requirements such as charset diversity covering the 3 expected charsets and length of 8, which may encourage users to turn to simple yet effective leet/mangling transformations (see also Section 4.1.3.4).

4.1.3.3 Weaknesses

Currently implemented leet transformations exclude some common conversions (e.g., $a \leftrightarrow 4$). Also, adding an extra character anywhere in the password defeats the dictionary check, as the algorithm expects an exact match only (equality test on lowercased strings). Hence, *P@$5W0rD!* is very strong, but *P@$5W0rD* is weak. Most entries from the blacklist dictionary are also never used, as a blacklist check is performed only on passwords that meet the minimum length requirement (8 characters), but 383 (68%) of the dictionary words are less than 8 characters long. Among the remaining 183 blacklisted words, three of them contain digits that are leet de-transformed from the candidate password prior to checking against the dictionary; e.g., the password *1234Qwer* (found in the blacklist in lowercase) is first converted to *lze4qwer*, which in turn is not found in the dictionary and assigned a medium score (instead of weak as possibly intended). The resulting 180-word blacklist is only able to catch 210 passwords (0.03%) of our leet dictionary (rated as weak). Even the password policy is more efficient against this dictionary as 3.8% of it are rejected and tagged as very

weak (lack of digits).

Also, a password not meeting the basic requirements (e.g., including characters from multiple charsets) cannot be better than very weak, implying that long memorable passwords (single-case passphrases) are denied in favor of more complex ones. This effectively bans all non-mangled cracking dictionaries. Among the mangled ones, although there is one rule that includes a first letter uppercase and a leading digit, few passwords are long enough to be considered strong or very strong.

4.1.3.4 Targeted dictionary for FedEx

From the results in Figure 3, it may appear that FedEx's meter is nearly optimal in detecting the most common passwords. However, such results may be misleading as our dictionaries are selected to uncover only the general weaknesses of widely-used meters. A more targeted dictionary may reveal specific weaknesses of a given meter. To evaluate this hypothesis, we built a combined dictionary using words from Top500, JtR and Cfkr. We then applied slightly more refined mangling rules that are consistent with [66, 29], namely: (a) capitalize and append a digit and a symbol; (b) capitalize and append a symbol and a digit; (c) capitalize and append a symbol and two digits; and (d) capitalize, append a symbol and a digit and prefix with a digit. We then removed the passwords below 8 characters, resulting in a dictionary of 121,792 words (only 4 symbols and 4 digits are covered for simplicity). 60.9% of this dictionary is now very-strong, 9.0% is strong, 29.7% is medium, and the rest is very-weak (due to repetitions of the same character). Thus, the FedEx checker is particularly prone to qualify easy-to-crack mangled passwords as of decent strength, as it cannot detect the core word anymore. One evident weak point in the algorithm is the way a password is searched in the dictionary, which checks for equality with the entire password, rather than searching for dictionary words as a substring of the password.

4.1.4 Intel

Intel provides an independent checker entitled "Are you hackable or uncrackable?", which aims at educating people about weak passwords by providing a simple binary answer along with an estimated time it would take for an attacker to crack the evaluated password.

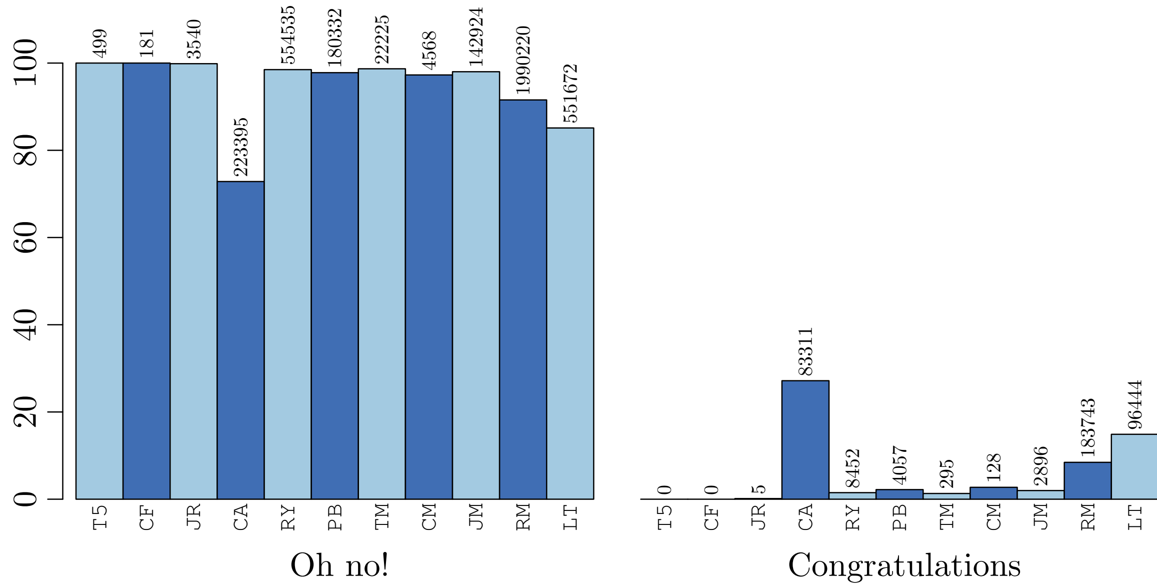


Figure 4: Intel checker password strength distribution

4.1.4.1 Algorithm

This checker is based on a simple logic. It first replaces parts of the password that are included in its 9986-word dictionary with the letter "a", considering a blacklisted word as good as a single letter. Then, it counts the number of lowercase, uppercase, digits and special characters included in the password. Special characters are considered as all non-letter and non-digit characters. An "entropy" value is then calculated as follows, although it is actually a search-space size:

$$size = \frac{lowercase^{26} \cdot uppercase^{26} \cdot digit^{10} \cdot special^{32}}{2}$$

Note that *special*^{32} means that it restricts the set of special characters to the symbols found on a US keyboard only, not including the space. Also, this calculation of the entropy assumes an attacker who already knows the type of charset used at each position.

Next, the search-space size is mapped to a time representing the effort needed to crack the password. The mapping is done by assuming an attacker can try $2 \cdot 2^{33}$ passwords per hour. It is unclear to us why this coefficient is not written as 2^{34} in the source code, nor what does the corresponding 4,772,186 passwords for second represent. Finally, labels are assigned by checking whether the time to crack a password is greater than $24 \cdot 365 \cdot 1000$ seconds, in which case the user receives congratulations for the strength of her password. Once again, it is unclear what this threshold means. It appears to aim at representing the number of seconds in a year; however this formula is certainly not the correct way to calculate the seconds in a year; the threshold is equivalent only to 101 days (3.3 months).

4.1.4.2 Strengths

For very simple words, the checker is able to assign a low score because of the dictionary check, which contains almost all Top500, 70.72% of Cfkr and 75.68% of JtR. Since words found in the dictionary are shortened to a single "a" letter, mangling rules are useless to cope with the shortage in password length. For example, *password123* is reduced to *a123*, which is included in a search-space of size 26,000, which is further covered by the modeled attack within a second. The capitalized version of this password is also caught because the word *assword* is included in the dictionary, reducing the password to *Pa123*.

4.1.4.3 Weaknesses

It is more interesting to notice what happens to our leet dictionary. *Pa$$w0rd* is estimated to resist for 3.5 hours. Although this time is certainly too short to rank this password as a good one, it is also unrealistic for an attacker to spend this much time

on such a simple password. Typically, this transformation will be checked within few minutes. Also, *ILoveFacebook* is only reduced to *ILoveFacea*, which is said to resist 5 months and is tagged as non-weak. The checker becomes highly unrealistic as password grow in length, because of the explosion of the search-space, which accounts only for simple brute-force attack. Another example is *~!@#$%^&*()_+*, the sequence of symbols on the top of a US keyboard, which is said to take 122,573 years to be cracked.

4.1.5 Microsoft

Microsoft's password checker is available as a separate webpage for users to evaluate any password, similar to Intel. The JavaScript source of the client-side checker also had a commented old version. During our study of this meter (June–July, 2013), the checker was updated to a newer algorithm. We evaluate all three versions here. All algorithms are based on predefined strict rules as explained below. Figure 5 summarizes our results.

4.1.5.1 Algorithm version 1

This version first classifies passwords as weak, and then as medium when the candidate password is at least 8 characters long, spans on a minimum of 2 charsets and is not included in the 2254-word embedded dictionary. To be classified as strong, a password must be at least 8 characters long, span on 3 charsets, and must not be a dictionary word, or some transformations of the dictionary words (e.g., leet transformations). Finally, the best score needs 14 characters covering 3 charsets, and shares the same constrains about the dictionary check and transformations as for the medium score. This version considers only the special characters of a US keyboard as part of the symbols charset (except "\", apparently due to a programming mistake).

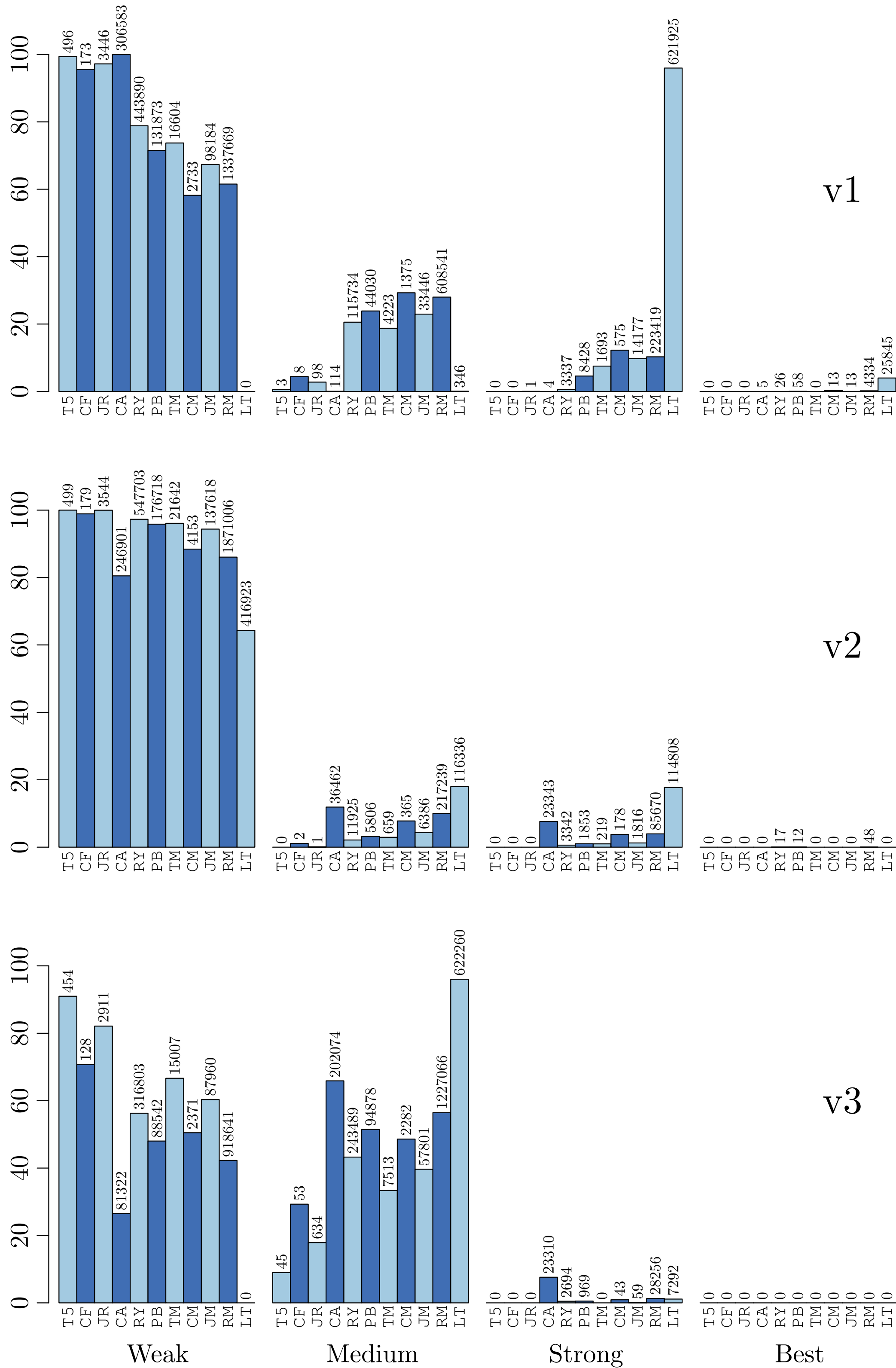


Figure 5: Microsoft checkers (v1, v2 and v3) password strength distribution

4.1.5.2 Algorithm version 2

The new algorithm, which became obsolete during our study, is based on some custom entropy calculation, and considers only mixing different charsets (lowercase, uppercase, digits, symbols) to assign better scores. The best score can be reached by either 28 lower or uppercase characters only, 23 lower-upper mixed characters only, 39 digits only, 32 symbols only, or 20 characters combined (all charsets). Due to these simplistic rules, passwords with repetitive characters, e.g., 14 and 28 "a"s are rated as strong and best, respectively. Also, this version distinguishes between symbols found on the upper part of an US keyboard and other characters. However it considers the "other" charset size to be the number of characters between ASCII code 0x20 and 0x7F (hence a size of 95), which is limited to the 7-bit ASCII table and does not account for the extended 8-bit version used to accommodate various international symbols. This leads to an overestimate of the intended entropy as all other special and international characters are considered to be found in a set of 95 elements only.

Compared to version 1, this algorithm is more stringent in terms of mandating longer and/or more complex passwords to return a strength better than weak. However, it also allows strong passwords from long lowercase words only (e.g., concatenation of common words). C&A demonstrates this behavior by being the dictionary with the most non-weak passwords (19.5% of the dictionary). The few passwords that are rated best by this version, are actually outliers in the leaked dictionaries. For example, 7 out of 12 phpBB's best rated passwords are email addresses, 2 of them are apparently MD5 hashes (32 characters in hexadecimal representation), and one simply consists of 32 stars (*). For RY5, best rated passwords are email addresses, long unique-character passwords, URLs, or what seem to be generic text messages. Finally, 48 passwords from RY5+M that are just alphabet sequences, long keyboard combinations or repetitive unique characters, are also categorized as best passwords.

4.1.5.3 Algorithm version 3

On July 11, 2013, Microsoft deployed another version of their checker, which is apparently simpler than earlier versions. A password is first rated based on its length alone: weak for under 8 characters, medium between 8–13, and strong for 14 characters or more. Strong passwords are rated "BEST" if they consist of characters from all 4 charsets. This latest change in the algorithm has a greater impact on the scoring of our dictionaries, compared to version 2. This version is the only checker that labels no passwords from our dictionaries as strong for an obvious reason: a 14-character password covering 4 charsets should be quite rare in a password dictionary. Version 3 also considers symbols as any special character found on a US keyboard.

4.1.5.4 Strengths

Although version 1 did not consider passphrases as strong, and required a minimum charset complexity, version 3 has reversed such logic by allowing better ratings for single-charset passwords. This leads to a jump from 0.04% (Version 1) to 74.5% (Version 3) of passwords in C&A ranked as medium or better. As version 3 considers the length of a password more favorably than its charset complexity, our leet dictionary is rated mostly as medium in version 3 (rather than strong in version 1).

4.1.5.5 Weaknesses

Microsoft dropped their embedded blacklist dictionary in version 2, and adopted a simpler algorithm that could be easily bypassed by repeating a character several times, e.g., padding a simple password with a unique symbol. At least, in version 3, a 32-star password is rated as strong and cannot become best unless it has also a lower and upper-case letter and a digit; however, any 14-character repeated password, e.g., *11111111111111* is rated as strong. The length requirement is also apparently too restrictive and makes randomly generated passwords such as *C7ef*1Y6#A* being only medium in versions 2 and 3, and *9Exe3Yz@SGErq* being medium in version 3; in fact, any random 13-character password is rated as medium in version 3.

4.1.6 Tencent QQ

QQ is an instant messaging software service developed by the Chinese company Tencent and mostly targets Asian countries. It is reported to have at least 800 million users as of March 2013.[16] Figure 6 summarizes our results for QQ.

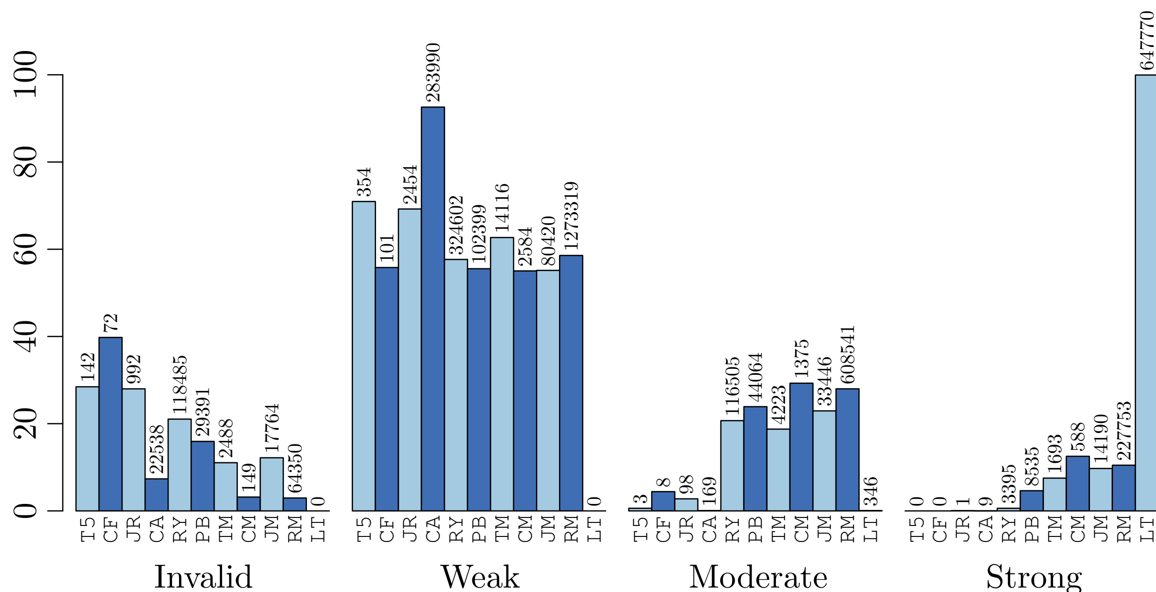


Figure 6: Tencent QQ checker password strength distribution

4.1.6.1 Algorithm

Once located, the part of the code responsible for evaluating the strength on the registration web page is only 12 simple lines of code, plus the mapping of the output numbers (1, 2, 3) to their respective label. It is also very simple in the design and proceeds as follows. No strength is returned and a warning is displayed if the password is less than 6-character long, or is composed of only 8 digits or less. We categorize such passwords as invalid in Figure 6. Passwords are not ranked better than weak unless they reach 8 characters in length. The number of charsets (from the 4 common ones) included in the password are counted and mapped as a strength as follows: the

[16]http://www.prnewswire.com/news-releases/tencent-announces-2012-fourth-quarter-and-annual-results-199130711.html

use of 1 charset is weak, 2 is moderate, 3 and 4 are strong. Note that QQ (along with Intel) does not constantly evaluate a password as it is typed by the user but rather ranks it once the user switches to another field of the form after finishing typing it.

4.1.6.2 Weaknesses

Similar to Drupal, QQ's checker is sensitive only to the number of charsets after crossing the minimum length threshold, which leads to a low ranking of passphrases and inconsistencies such as *Password1* being rated as strong.

4.1.6.3 Interesting features

During the exploration of the code and unrelated to the scope of our analysis, we noted two interesting features on the registration page. First, the total time spent creating the password in milliseconds (time focused on the password field) along with the number of keys pressed inside the field (tentative characters to create a password) are recorded and sent to the server once submitting the form. Second, the password is encrypted using a 1024-bit RSA key prior to being sent to the server, most likely because the form is not submitted through a secure connection (HTTPS) but simply sent by an HTTP request.

4.1.7 Twitter

Twitter's checker is fully client-side, and has the most diverse strength scale with 6 categories.

4.1.7.1 Algorithm

The algorithm is based on an increasing score, which considers the password length and presence of the following features: at least three digits; at least two symbols; both lower and upper-case letters; letters and digits; a mix of symbols and digits; and a mix of symbols and letters. The algorithm carries an optional check which is not

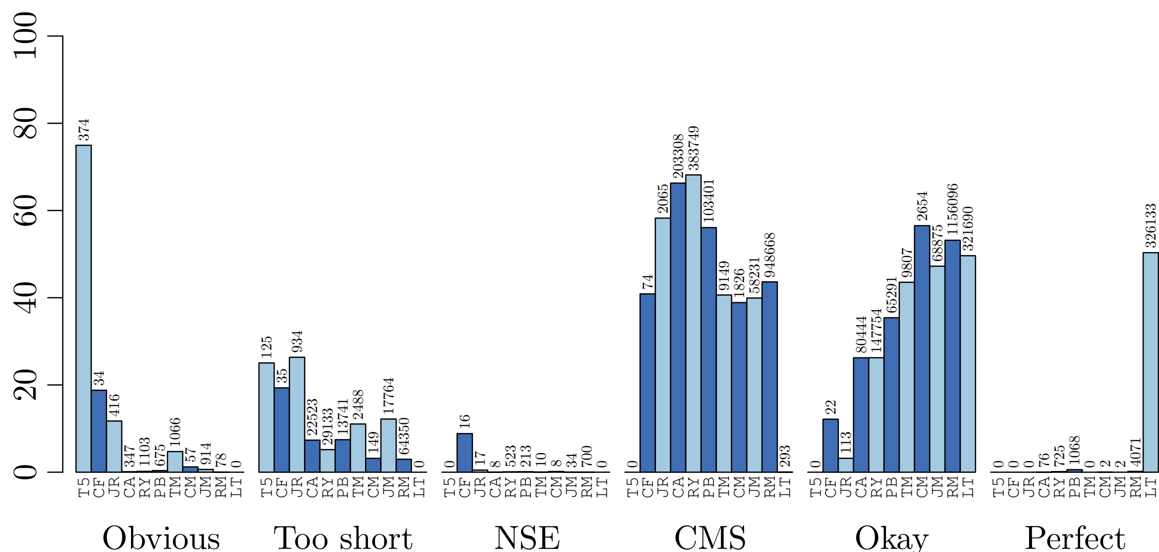


Figure 7: Twitter checker password strength distribution

performed that would enforce the presence of the four charsets in a password for it not to be considered as "too weak" (treated as too short afterwards). This check is also intended to enforce a length of 10 characters, however it would not be sensitive to the length because of a programming error. A range of thresholds then maps the score to a strength label. An embedded 401-word blacklist is used, and passwords matching the blacklist are labeled as obvious. The strength label "not secure enough" (NSE) is rare (in our tests) and is apparently only meant for passwords that "could be more secure" (CMS) but are getting weakened by the negative coefficient for repetitions (e.g., *aaaaaaa*, *deedee*, *annnna*). The blacklist covers a large portion of the Top500 dictionary, hence the obvious scoring of that dictionary. Most passwords in our test are considered "Could be more secure" (CMS) and "Okay," while there are very few perfect passwords. A perfect strength can be reached with a short password covering all 4 charsets, e.g., *Tw1$er*, *P@ssw0rd*, or longer 3-charset passwords (e.g., *Password123456*), or even longer 2-charset passwords (e.g.. *Impossibleisnothing*), or a combination of lowercase words (e.g., *theologicoastronomical*).

4.1.7.2 User information

If a password directly matches the username, it is considered obvious and disallowed. However, such restriction can be easily bypassed, by adding extra characters to the password. In addition, the password is not checked against the email address and full name.

4.1.7.3 Weaknesses

The blacklist check is quite simplistic; e.g., *mustang* is in the blacklist, but *mustang1* is considered "okay" (a category right below "perfect" in the strength scale). Only the case is made independent for the validation by converting the password to lowercase. The mangling rule that makes the first letter uppercase and adds a trailing digit in a given word, performs very well in deriving okay passwords. Hence, *Qwert0*, *Magic1*, *Hello2* are okay. Also, a special character is good enough to replace both uppercase and digit, e.g., *super!*, *lucky!*, *naked?* are okay. The length can also replace the need for a digit or uppercase letter; e.g., *Mypassword*, *Anythings*, and *interneting* are considered okay. However, in order to be considered perfect, a password needs a longer length, or more digits and special characters; examples include *passw0rd!*, *p@ssw0rd*, *troubl#3*, *e=mc**2*, *anatomicophysiological*. However, repeated simple (even blacklisted) words can easily yield okay (*passwordpassword*) and perfect (*passwordpasswordpassword*) passwords.

4.1.8 Yahoo!

Yahoo! relies on a client-side checker with four main strength categories. The additional invalid category is used for passwords matching the provided user information, or if the password is *password*. We omit this category in Figure 8, which summarizes our results for Yahoo!.

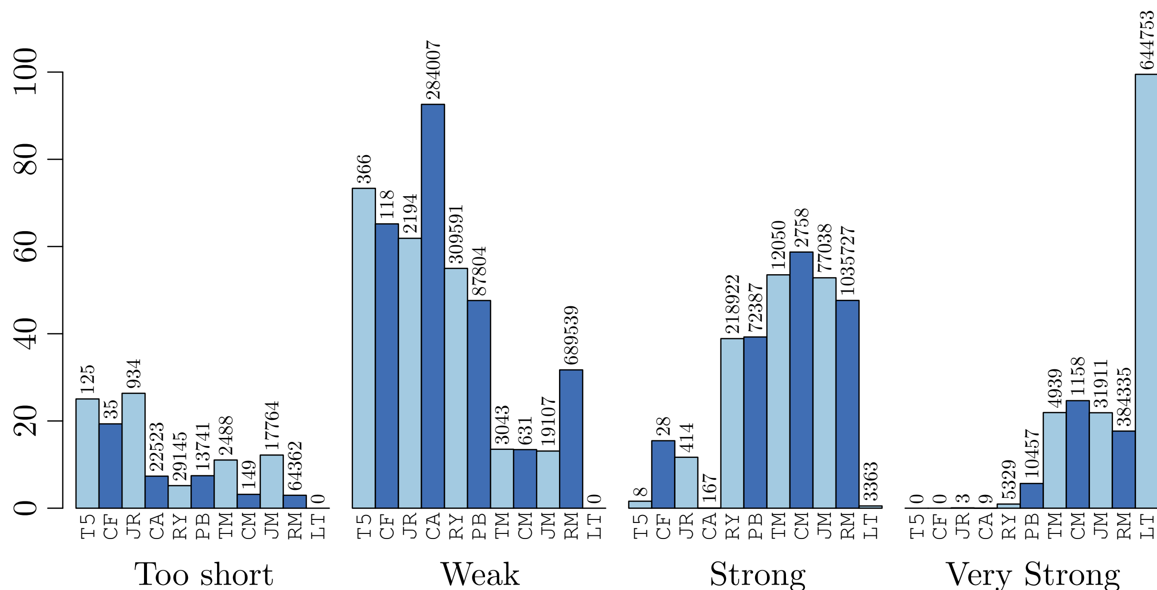


Figure 8: Yahoo! checker password strength distribution

4.1.8.1 Algorithm

This algorithm mostly checks for multiple charsets in the candidate password. However, a lowercase password with a single uppercase, digit or symbol is already considered strong; hence, many mangled passwords (derived from simple words) are rated as strong. C&A is largely limited in the weak category as most of its passwords are lowercase-only. In the source code, a single-charset password passing the 6-character requirement is supposed to get a "mediocre" score. However, when mapping internal qualification to user-readable feedback, this score is translated to "weak". For Yahoo!, the symbols charset includes only `!@#$%^&*?_,~`.

4.1.8.2 User information

Yahoo! takes into account the user's name and email address to weaken the strength of a candidate password. User information is detected in a password if it is included in it (compared to simple equality check).

4.1.8.3 Weaknesses

A password is strong if it has at least 2 charsets, e.g., *Abcdef*. Reaching the very strong level, where a password must have a lower-uppercase combination and a symbol or digit, is quite trivial; e.g., *Abcde0*. Also, no blacklists or checks for patterns are used.

4.1.9 12306.cn

China Railway provides a ticket reservation system hosted at `www.12306.cn`. It is the government's official website for train tickets and generally visited by millions of customers. Figure 9 summarizes our results for 12306.cn.

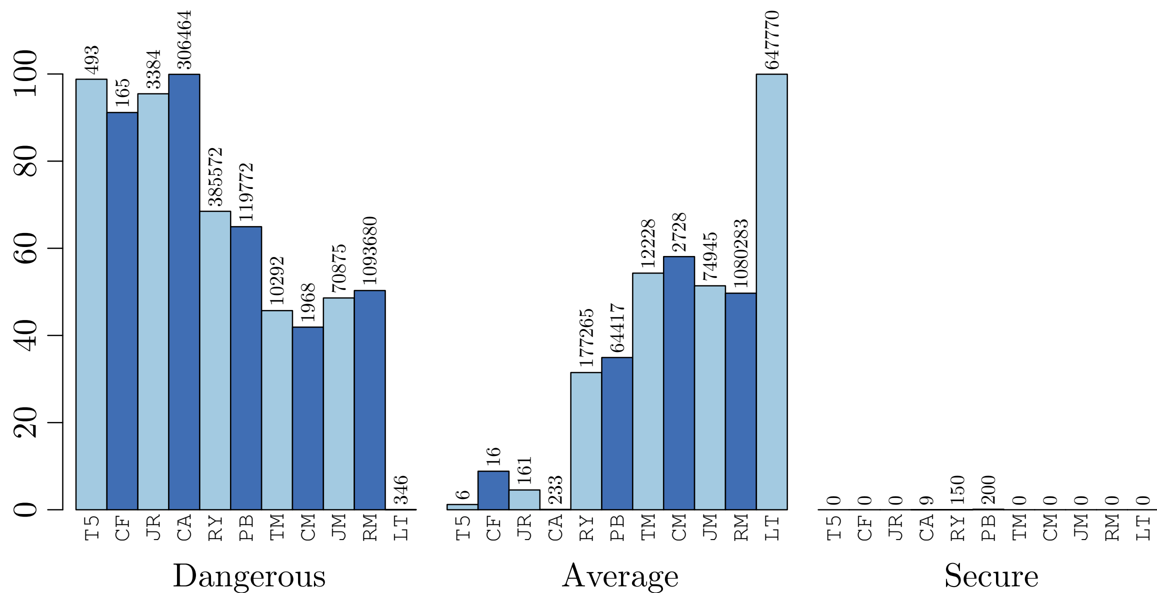


Figure 9: 12306.cn checker password strength distribution

4.1.9.1 Algorithm

A password is considered dangerous (translated from the Chinese word found on the website) if its length is less than or equal to 6 characters, if it is only composed of letters (lower and uppercase), if it is only composed of digits, or –surprisingly– if it is only composed of an underscore character. If none are true, the password is labeled

as average. It can reach the label “secure” if it contains mixed-case letters, digits and at least an underscore character.

4.1.9.2 Weaknesses

Similarly to QQ, 12306.cn is only sensitive to the password composition after a minimal length of 6 characters and hence receives the same comments in this regard. One interesting finding is the promotion of the underscore character which is the only special character considered. It is unclear to us why such choice has been made. However, it clearly demonstrates inconsistencies when a 20-character long random password covering all charsets but omits an underscore (e.g., *SFh*#6^Z44CwhKB73@x3*) is only rated as average. We wish users could choose such passwords on average. Consequently, the meter appears extremely stringent to users wishing to reach a secure score, until they find the magic character.

4.2 Web-Based Password Meters (Server-Side)

4.2.1 eBay

eBay employs a fully server-side checker. Once the minimum requirement of 6 characters is met, an AJAX query to the server-side checker is made. The username and email address are sent to the checker and the response provides the password's strength along with different messages for the user. Figure 10 summarizes our results for eBay.

4.2.1.1 Algorithm

A password is considered as invalid and is not sent to the server-side checker before it matches the length requirement. (In July 2013, eBay introduced a “Too short” feedback to replace the term invalid for short passwords.) The checker requires passwords to cover any two charsets; thus, many words from Top500, C&A and JtR are

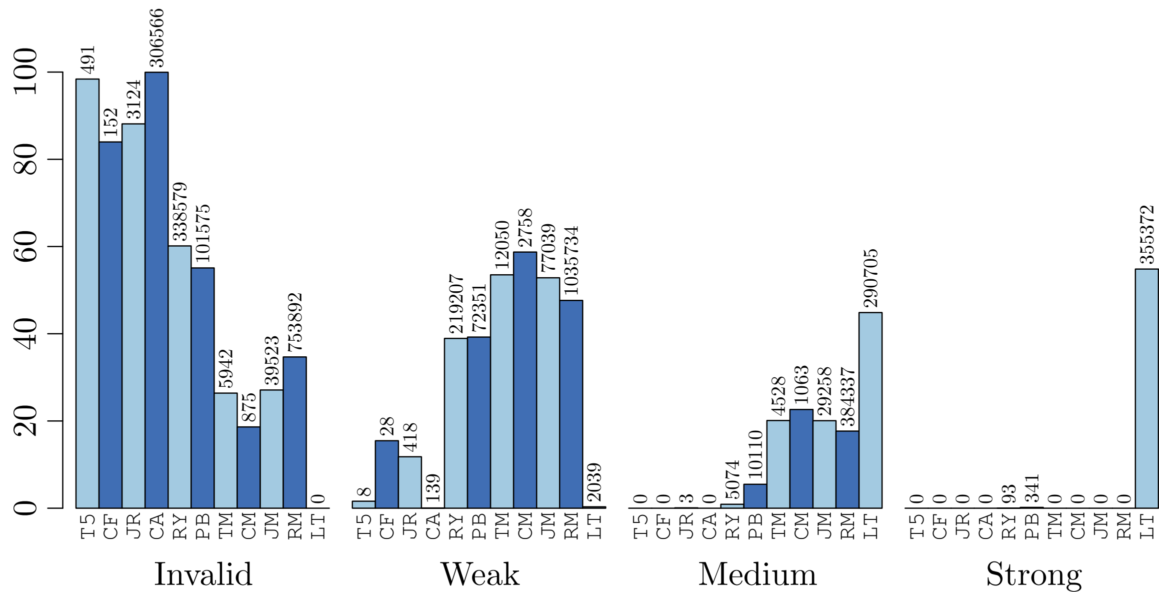


Figure 10: eBay checker password strength distribution

considered invalid. As a side-effect, concatenation of simple words always leads to weak passwords (in contrast to Dropbox).

Based on our tests, it seems that the server-side algorithm is fairly simple but quite stringent on the number of charsets used: a single-charset password is invalid, two is weak, three is medium and four is strong. The password length becomes irrelevant, once passed the minimum requirement (similar to Drupal, see in appendix). To validate our observations, we compared the results with a checker we built specifically to apply the above policy only. We found that the results are very similar. By examining the differences, we noticed that eBay's checker considers only `~!@#$^&*-+` as symbols and the remaining special characters do not influence the strength. The server-side functionality is apparently equivalent to our 20-line JavaScript code. Also, passwords fully composed of unrecognized symbols do not receive any strength score (e.g., for *%()_{}*, the checker returns an empty value).

4.2.1.2 User information

eBay prevents a user from choosing a password similar to her user ID, taking into account case changes and few leet transformations as found in our test. For example, having *leettest* as the user ID, the passwords *LeetTest*, *L3ettest* and *LEETTEST* are invalid; *L33ttest* and *lEEtt3st* are medium; and *LeetT3st+* is strong. In Table 1, we rate eBay's checker as taking registration items into account, even though it excludes the user's real name from the check.

4.2.1.3 Weaknesses

eBay does not consider any password as strong from our test dictionaries except few passwords from phpBB and RY5, in which we can find few relatively simple passwords being labeled as strong, e.g., *P@ssw0rd*, *0b1W@n*, *sp3ciaL***, *phpBB2!* and *p0pm@iL* (found in the phpBB dictionary). These are simple leet transformations of common words with possibly leading special characters. Also, several common passwords are categorized as of medium strength, including, *Lucky1*, *Abc123* and *Password1*. Simply covering all four charsets is also enough by design to get a strong score, e.g., *AAaa1+*, resulting high scores for our leet dictionary.

4.2.2 Google

Google also uses a server-side checker. Unlike eBay, the username and email address are not sent to the checker. The response provides the password's strength. Figure 11 summarizes our results for Google.

4.2.2.1 Algorithm

Google's checker is difficult to reverse-engineer because of its inconsistent output at times. Consider the following examples: *testtest* is weak and *testtest0* is strong, but *testtest1* is fair, *testtest2* is good and *testtest3* is strong again. It is no surprise that a simple repetitive string, such as *testtest*, followed by a digit is considered weak, but

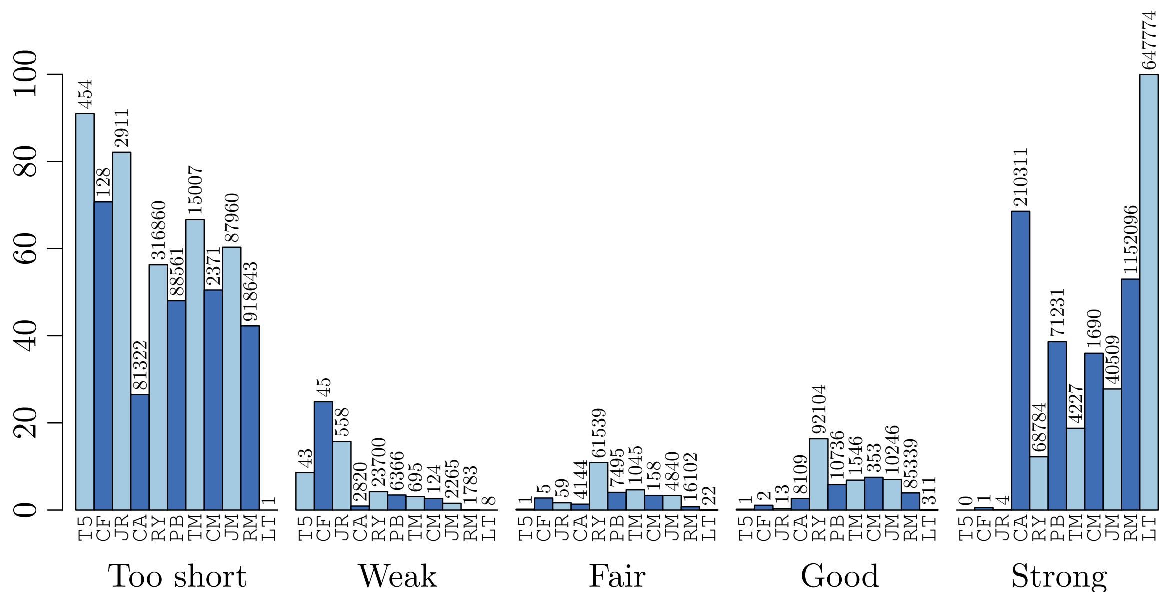


Figure 11: Google checker password strength distribution

generating such a variety of scores for so minor changes is difficult to comprehend. In addition, we found that *test1234* and *Test1234* are weak, while *TesT1234* is fair, *TeSt1234* is good, and *TEst1234* and *tEst1234* are strong. From these results, one can approximate that a first letter uppercase is not as rewarding as another letter being uppercased, and that a pattern of first and last letter uppercase is labeled as intermediate. If this approximation is true, then many commonly used patterns would be penalized. However, Google's checker is ranked first and fifth in terms of yielding "strong" passwords from our base and mangled dictionaries, respectively (see Chapter 5 for more comparison).

As we can find examples such as *huntings* being rated as strong and *rainbows* as weak, it appears that a blacklist check is run, although it is unclear whether any widely known dictionaries are included in the list. Simple dictionaries such as Top500, JtR and Cfkr are too weak to pass the 8-character requirement; however, mangled versions of these dictionaries yield many strong passwords. We also noticed some jumps between weak and strong strength scores by the addition of a simple character, e.g., *password0* and *password0+*. This may be due to an exact blacklist check that fails to recognize the latter as a common word because of the extra character (+).

We also found that strength scores significantly vary with time. When performing our tests (between June–July 2013), we waited two weeks before testing the dictionaries again. Overlapping passwords between dictionaries tested before and after the two-week period, were qualified differently. A total of 1700 common passwords between JtR+M, RY5 and phpBB are evaluated differently. Usually, the difference remains limited to one strength level (better or worse). For example, *overkill* went from weak to fair, *canadacanada* from fair to good, and *anythings* from good to strong, while *startrek4* went from strong to good, *Iloveyou5* from good to fair, and *baseball!* from fair to weak. We again tested the dictionaries 5 weeks later, and found that some of the passwords, which had their scores changed in the second test, were reverted back to their original ones (first test).

In another run of our experiments in November 2013, we found that repeated tests of a password, e.g., a dozen times in the same minute, can lead to different outcomes. These fluctuations may indicate the use of a dynamic or adaptive password checker. Irrespective of the nature of the checker, users can barely make any sense of such fluctuations. Finally, Google explicitly rejects any complex symbols or international characters by mentioning that only "common punctuation" is allowed (the candidate password's strength drops down to too short otherwise).

4.2.2.2 Weaknesses

Simple dictionary passwords can easily reach a good or strong strength. Examples include: *access14* in Top500 (good) or *Access14* when mangled (strong), *slideshow* and *sample123* (good), and *morecats* (strong). Only a few passwords in our tests fall between weak and strong, and it is fairly easy to slightly change a weak password to make it strong, e.g., by simply changing one letter to uppercase and/or adding a leading digit or symbol. A typical example is *database*, which is weak, but *Database*, *database0* and *database+* are strong. However, for the weak password *internet*, *Internet* remains weak, *internet0* is fair and *internet+* is strong.

4.2.2.3 Problems

Google's registration form suffers from a hysteresis phenomenon. When a password reaches 8 characters, it gets evaluated for the first time (other than too short). Then, if the user deletes the 8th character, the strength returns to too short. However, if the user re-enters the same character again, the strength doesn't change and stays as too short. Hence, the same 8-character password can be categorized as too short or strong depending on how it is typed, e.g., typing *R4m51sWd* then removing and adding back the last "d".

4.2.3 Skype

Skype is now a Microsoft-owned VoIP service provider that relies solely on a server-side validator, independent of the Microsoft checker. Similar to Google, no user information is sent to the server during password evaluation, allowing a user to register with a password close to her username.

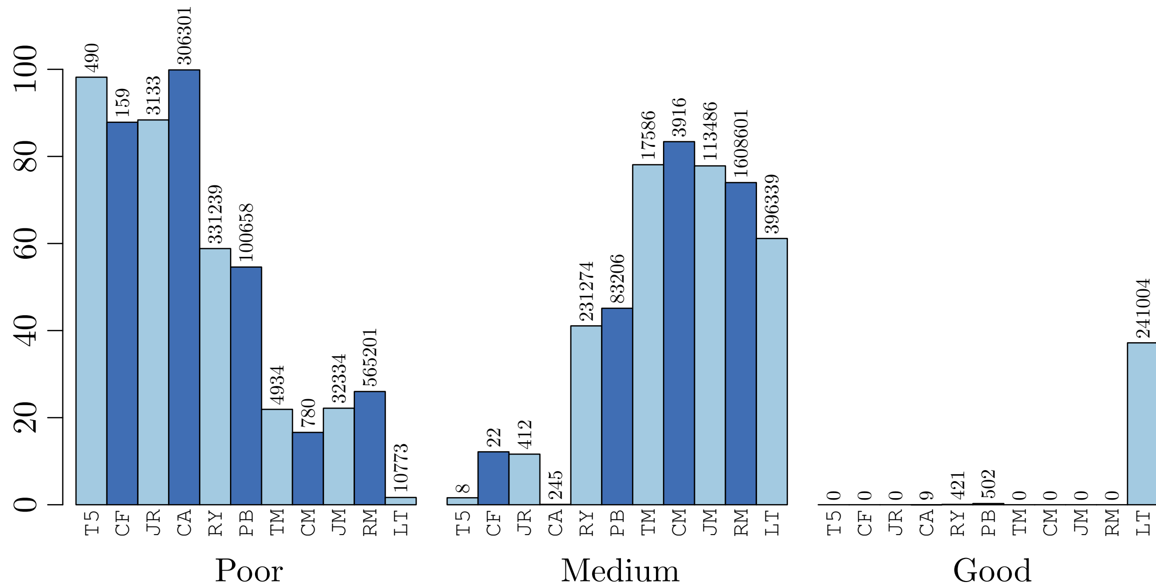


Figure 12: Skype checker password strength distribution

4.2.3.1 Algorithm

Skype has one of the simplest strength scale: a password is either poor, medium or good. The results of our tests are also straightforward. All lowercase or single-charset passwords are poor (preventing the use of long user-memorable passwords), and 2-charset passwords are medium, e.g., *Tennis*, *sleep0*. No mangling rules in our tests could generate a strong password (even a randomly selected 20-character password, *2eppf5N58J5z8yt2h52T*, is considered medium). We found that for a password to be "good", it must be composed of at least 2 symbols along with at least 6 other characters. This rule makes *soccer++* and *pa$$word* good. Apparently, there is very little check against slightly-modified common words, even though an extra warning is displayed for very simple words ("Password is too easy to guess"). This warning appears for *Password* and *MySkype* for example, but not for *p4ssword*. Among the cracking dictionaries, only 9 C&A passwords reached a good level. These passwords all contain double underscores to separate words, e.g., *mot__de__passe*, which is the 3-word French equivalent of *password*. Finally, only phpBB and RY5 were successful at reaching a good strength more often with passwords such as *hockey!!*, *bl@hbl@h* or *//please*.

4.2.3.2 Weaknesses

The checker is very simple, does not consider repetitive patterns, common words (other than trivial ones), and password length beyond 6 characters.

4.2.3.3 Problems

Some leaked dictionary passwords start with a `<` character (e.g. `<?php ?>`, `<3love`) and do not receive any score. Also, one of the passwords in phpBB dictionary was a compilation command (`gcc -O2`) and generated a systematic internal server error during our tests at the end of June 2013. These passwords may interfere with an intrusion prevention system that prevents the server-side program from receiving these apparently dangerous inputs.

4.3 Web-Based Password Meters (Hybrid)

4.3.1 Apple

The client-side/in-browser part of Apple's hybrid meter first checks if a password meets the policy requirements: 8-character long, has one lowercase, one uppercase, one digit, and does not include more than two identical consecutive characters. The password is then checked against a server-side blacklist. Blacklisted passwords are disallowed, irrespective of their strength as measured by the client-side checker. Figure 13 summarizes our results for Apple. Note that we have created an extra category to group blacklisted passwords, even if they are categorized as moderate or strong.

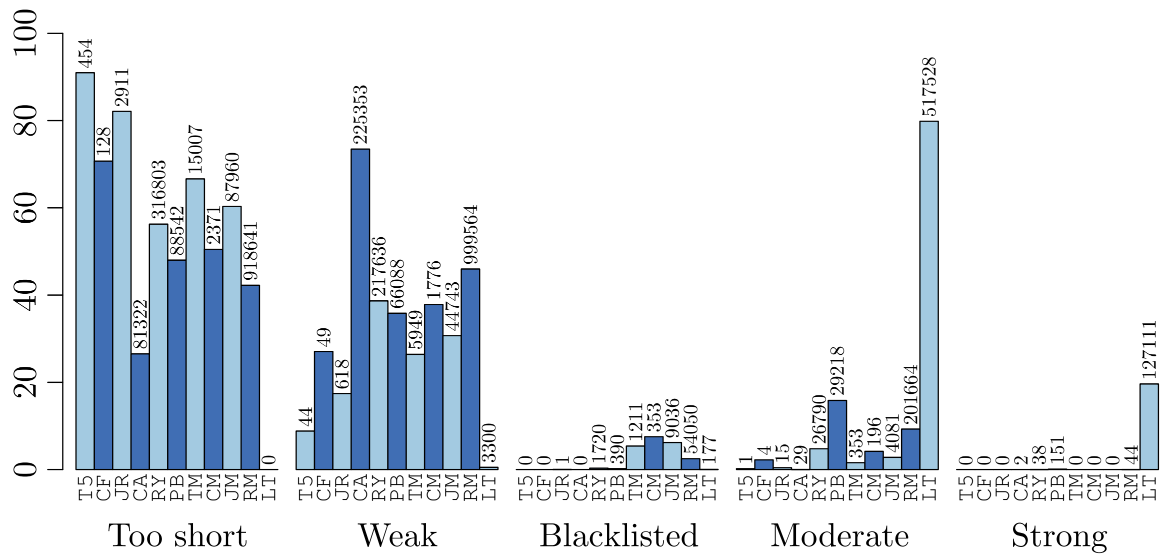


Figure 13: Apple checker password strength distribution

4.3.1.1 Client-side algorithm

The client-side part of the algorithm is based on an increasing-only score, adjusted by some rules with associated weights. The score is higher if a password contains more characters (by ranges of 6-7, 8-15 and 16+ characters). Having the following features also cumulatively contributes to higher scores: at least an uppercase letter;

one or two digits; three digits or more; at least a symbol; symbols separated by other characters (thus, encouraging symbols inside a password rather than at the ends); lower and uppercase letters; alpha-numerical characters; and alpha-numerical-symbol characters. This algorithm clearly encourages charset complexity. Similar to FedEx and eBay's checkers, Apple's considers only `!@#$%^&*?_,~` as symbols (remaining special characters do not influence the strength but are counted in the password length). Once a password passes all requirements, it is at least tagged as "moderate", hence we report this strength in the enforcement column for Apple in Table 1 (even though a password can be rated as "strong" while not passing all requirements).

4.3.1.2 Blacklist check

Apple does not provide any information about the blacklist checker. However, it is apparent from our results in Figure 13 that the blacklist contains Top500, JtR and C&A, albeit modified versions, which may explain the small percentage of dictionary words that still passes the blacklist check. This is also supported by some example passwords from these dictionaries; e.g., a rather unusual password, *Protransubstantiation1*, taken and mangled from C&A, is blacklisted. It appears that digits are removed from the dictionaries and only core words are kept in the blacklist. Passwords that do not pass the requirements during the client-side check, can still reach a moderate or strong strength; such passwords when modified to comply with the requirements, may then be blacklisted. For example, *Password1* and *A1b2c3d4* pass the client-side check, as they are long enough and satisfy charsets requirements, but fail the blacklist test (although they are labeled as moderate).

Many simple mangled passwords with the first letter uppercase and a terminal digit are caught by the server-side checker. For a given password, e.g., *Franklin123*, apparently the following steps are performed: the password is stripped from trailing digits, giving *Franklin*, and is then checked against a blacklist, disregarding its letter-case. For some base words such as *Franklin*, up to three trailing digits are stripped during the blacklist check, but only one digit is removed for words such as *Adorable*.

This behavior seems to originate from non-trivial rules that we cannot explain from our tests. However, adding extra terminating digits, and/or starting digits, easily bypasses the blacklist check.

4.3.1.3 Strengths

Apple's blacklist is the most comprehensive one among our tested checkers, with known common password dictionaries included in it. Also, while Apple imposes stringent requirements for passwords to be accepted, all passwords receive an evaluation score while being typed; thus, a user is guided towards a better choice of characters from the beginning of the password composition. (On the contrary, FedEx waits for the requirements to be met first, before evaluating a password.)

4.3.1.4 Weaknesses

Apple's policy requires that passwords should "not contain identical consecutive characters."[17] However, in practice, this restriction is too weak to catch repetitive patterns in some cases; e.g., *P4ssw0rd* is blacklisted but *P4ssw0rdP4ssw0rd* is labeled as strong. We also noticed that the blacklist check response times vary significantly—ranging from less than a second up to more than a minute. Very few dictionary passwords are considered strong, even when mangled. However, examples of strong passwords also include: *P@ssw0rd!*, *P@55w0rd* and *Robot123!* (mangled versions of simple passwords). Finally, a strong password can be blacklisted as it is the case for *Pa$$w0rd*, which is a unique state among other checkers.

4.3.2 PayPal

PayPal is a global online money transfer service and a subsidiary of eBay Inc. Although PayPal and eBay portals belong to the same company, they use different password checkers on their respective websites.

[17] `https://appleid.apple.com/cgi-bin/WebObjects/MyAppleId.woa/wa/createAppleId`

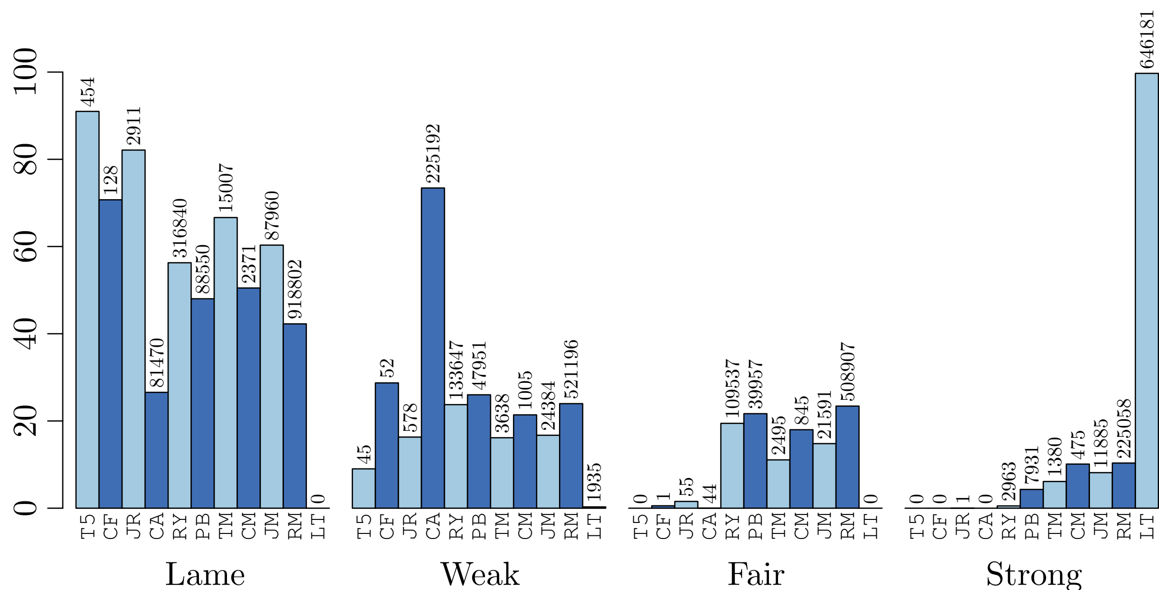


Figure 14: PayPal checker password strength distribution

4.3.2.1 Algorithm

PayPal's checker is hybrid in nature. It first involves a client-side process to check that a given password is at least better than weak, which requires spanning on more than two charsets, considering letters as one charset, independent of their case. Then it checks against an online checker for blacklisted words. Strength is directly linked to the number of charsets the password spans onto. However, repetitions (4 characters or more), sequences (4 or more successive digits or contiguous keys of the same row of a keyboard), or the use of account information (email address or username) in the password, make it weak altogether. Sequences include qwerty-like combinations that are supposed to be localized for the country selected by a user. We tested the French version of the website and found it still checks only for an US layout (arguably, there are only few changes between the layouts). The German PayPal website also behaves similarly, except that some series of accented letters found on a German keyboard are also checked for spatial combinations. In this case, sequences of 4 characters or more are searched in: `ertzuiopüyxcvjklöäölkjvcxyüpoiuztre`. Based on the source code variables name, a localized check only for German seems to exist. When

passwords are less than 8 characters, they are considered lame, which accounts for a majority of the dictionaries, expect C&A. The checker requires at least a digit or symbol for better scores; hence, passwords from C&A are mostly considered as weak. The symbol charset considers only the following characters: `~!@#$%^&*(),+=`.

4.3.2.2 Weaknesses

More than 1.5 million passwords were checked against the online blacklist; however, only 619 of them were effectively blacklisted. Thus, most passwords are categorized solely by the client-side checker, and only these 619 passwords were rated by a server-side action. Among the blacklisted passwords, 573 had the word *password* in them. The remaining had *trust*, *access*, *football*, *superman*, and few other words, as part of the password. Thus, we can conclude that the blacklist check is mostly useless and could be improved to catch more trivial passwords such as *Password0* (even though *password0* is caught).

4.4 Application-Based Password Meters

4.4.1 1Password

1Password is a password manager that protects various passwords and other forms of identity information in a password-encrypted vault stored on a user's device (computer or smartphone) or optionally in the cloud. The master password creation is guided with a password meter. This meter shows both a continuous progress bar and a label. However, as this application is a closed-source software, we treat this algorithm as a black-box. We later noticed that the browser extension that comes along with the software application, is also equipped with a password-strength meter, which is significantly different than the one implemented in the main stand-alone application. We were not expecting this application to implement two different meters, so we stopped at the first one we encountered and conducted its analysis before accessing the second

meter that is only available after the creation of a password vault protected by the master password. This second meter generates and evaluates random passwords in the browser, however the user can further modify or rewrite them completely and see the resulting strength. We first present an extensive analysis of the black-box meter, as it demonstrates the possibility to analyze closed-source compiled implementations. Figure 15 summarizes our results for the master password evaluation in 1Password version 1.0.9.340 (as of March 25, 2014) while Figure 19 summarizes our results for the JavaScript-based strength meter in its Firefox extension.

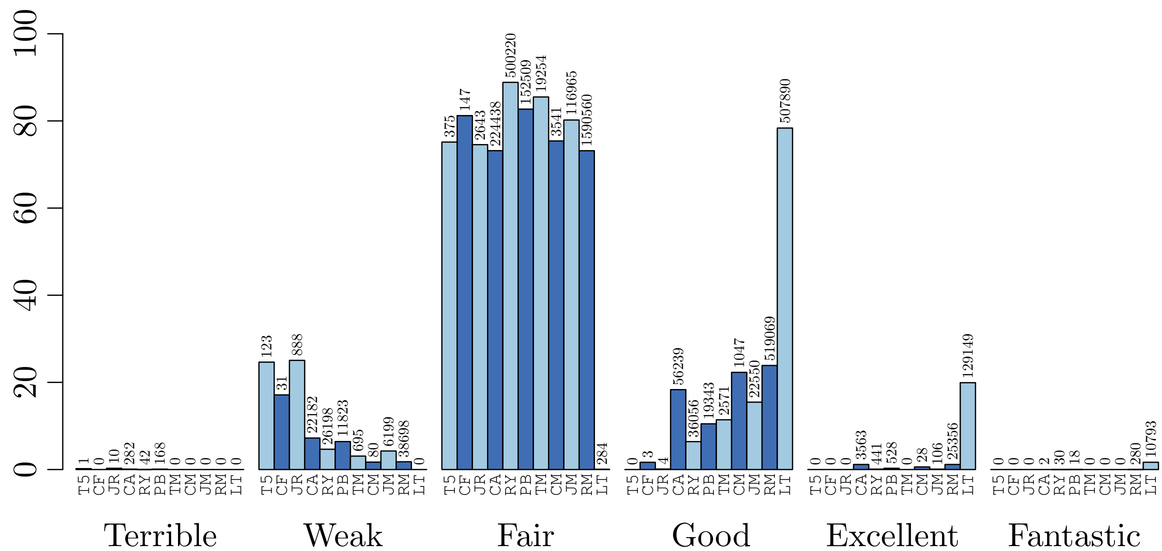


Figure 15: 1Password checker (software-based) password strength distribution

4.4.1.1 Algorithms

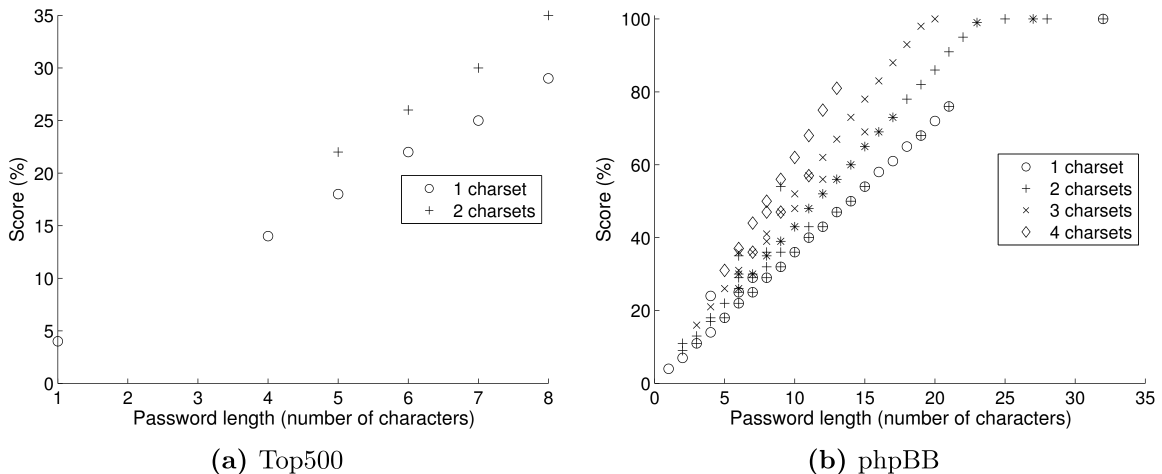
Master password strength meter. For our analysis, we started with the meter that is used during the creation of a master password; users must generate the master password before a password vault is created. Our method for uncovering the algorithm behind this black-box is to identify the features involved in the algorithm and understand their implication. We can extract fine-grained strength output from this

meter as it uses a continuous progress bar whose value is rounded to the nearest integer. The thresholds for the label assignment are therefore straightforward to deduce at the end.

The first parameter of interest is the length. We treat passwords as UTF-8 encoded when measuring their length since our dictionaries are encoded as such. We found a very high correlation coefficient of 0.9853 between the length and the score based on the output for Top500. However, Top500 contains a rather limited subset of possible passwords and do not include many charsets. We further compute a correlation coefficient of 0.8944 on phpBB, which covers a greater variety of passwords. The latter correlation indicates that the length is not the only parameter to influence the scoring. Furthermore, we found an interesting pattern on the graphs representing the output scores with respect to the length of passwords for Top500 and phpBB (see Figure 16).

We can approximately identify that the scores from Top500 form two straight lines, and those from phpBB form four straight lines (capped at 100). Reconsidering the composition of these dictionaries (one and two charsets in Top500 and from one to four charsets in phpBB), we hypothesize that the number of charsets is another feature taken into account in the algorithm. Figures 16a and 16b represent the output scores with respect to the length of passwords for Top500 and phpBB respectively, according to the number of charsets identified.

Based on these graphs, there seems to be other factors influencing the evaluation since some passwords with equal length and number of charsets are assigned different scores, and some scores are visually aligned with ones that belong to a group with a different number of charsets. By investigating these discrepancies, we found that passwords with non-ASCII are getting unexpected results. We further infer that non-ASCII characters are not handled correctly by the software and the length of the password is measured in terms of bytes, hence such characters can be counted as 2 to 4 characters in length. Also, these characters do not seem to be counted as part of a symbols or “others” charset. We adjust our measurement of the length and



(a) Top500 **(b)** phpBB

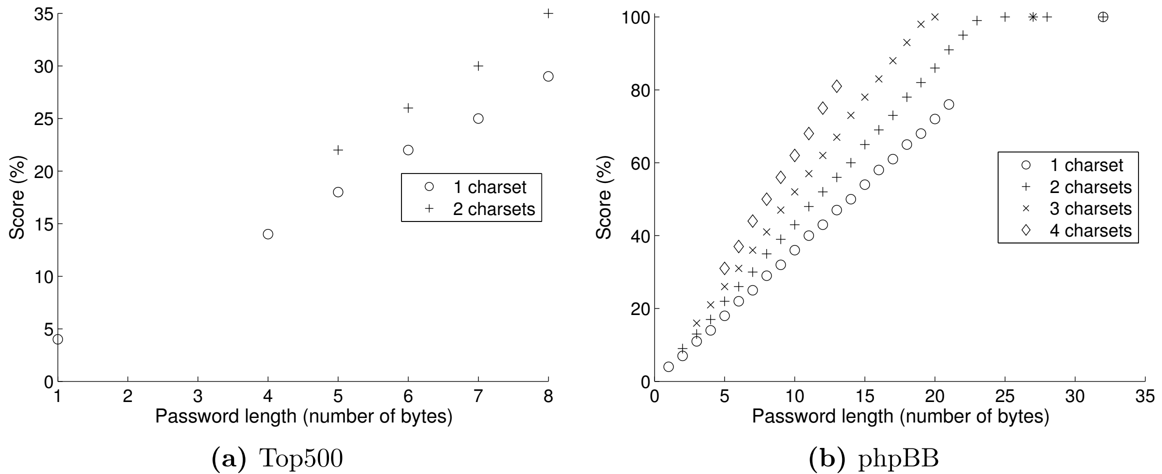
Figure 16: 1Password: Score versus password length identified as the number of characters and grouped by the number of charsets they span onto for (a) Top500 and (b) phpBB. Charsets are identified as lowercase letters, uppercase letters, digits, and all remaining characters. Note: (b) shows some overlapping scores with different number of charsets

identification of charsets with respect to this observation, and are able to produce Figures 17a and 17b that seem more straightforward to explain. For charsets identification, we restrict the symbol charset to match only special characters found on an English keyboard. Remaining characters are not counted as part of any charset.

Furthermore, we are able to infer the rule separating the lines belonging to each number of charsets. For passwords sharing the same length, we observe that scores are linked together by a single coefficient, e.g., for 1–4 charsets and a length of 13, scores equal 47, 56, 67 and 81 respectively. Once rounded to the nearest integer, we observe that this equation holds: $47 \cdot 1.2^3 \simeq 56 \cdot 1.2^2 \simeq 67 \cdot 1.2^1 \simeq 81$. We infer that the algorithm fits the following model between 0 and 100. We denote by $\|.\|$ the nearest integer of a given value.

$$\begin{cases} score = \left\|\alpha \cdot \left(length \cdot \beta^{\#charsets-1}\right) + \gamma\right\| \\ \alpha, \gamma \in \mathbb{R},\ \beta = 1.2 \end{cases}$$

Next, we compute four linear regressions on the score of passwords in phpBB, categorized by the number of charsets they include. We consider lowercase letters,



(a) Top500

(b) phpBB

Figure 17: 1Password: Score versus password length identified as the number of bytes and grouped by the number of charsets they span onto for (a) Top500 and (b) phpBB. Charsets are identified as lowercase letters, uppercase letters, digits, and symbols found on a US keyboard only.

uppercase letters, digits, and symbols on a US keyboard as the four charsets; and also remove passwords with scores of 100, since the original score may have been higher than the maximum of 100. To get an equation that depends only on the length, we normalize each line with respect to the number of charsets. For each line identified by the number of charsets c, we use MATLAB[18] to estimate the coefficients $\hat{\alpha}_c$ and $\hat{\gamma}_c$ of a one-degree polynomial for the n_c passwords with distinct scores that belong to the line c, given their lengths $length_{c,i\in[1,n_c]}$ and a normalized score of $\frac{score}{1.2^{c-1}}$. In a mathematical form, MATLAB tries to find $\hat{\alpha}_c$ and $\hat{\gamma}_c$ such that:

$$\forall i \in [1, n_c], \frac{score}{1.2^{c-1}} \simeq \hat{\alpha}_c \cdot length_{c,i} + \hat{\gamma}_c$$

Then, we compute the coefficients of determination for exact and rounded values of the modeled scores, $R^2_{c,\text{exact}}$ and $R^2_{c,\text{round}}$ respectively, according to the formulas shown in Figure 18. We denote $\overline{score_c}$ as the mean of the observed scores $score_{c,i}$ for the line identified by c charsets, and $\widehat{score_{c,i}}$ being the modeled scores for that line. Results of the regressions are summarized in Table 4.

[18]`http://www.mathworks.com/products/matlab/`

$$
\begin{aligned}
\overline{score_c} &= \frac{1}{n_c} \sum_{i=1}^{n_c} score_{c,i} && \text{(mean of scores for line } c\text{)} \\
\widehat{score_{c,i}} &= \hat{\alpha}_c \cdot \left(length_i \cdot 1.2^{c-1}\right) + \hat{\gamma}_c && \text{(modeled scores for line } c\text{)} \\
SS_{c,\text{tot}} &= \sum_{i=1}^{n_c} (score_{c,i} - \overline{score_c})^2 && \text{(total sum of squares)} \\
SS_{c,\text{res,exact}} &= \sum_{i=1}^{n_c} \left(score_{c,i} - \widehat{score_{c,i}}\right)^2 && \text{(residual sum of squares for exact values)} \\
SS_{c,\text{res,round}} &= \sum_{i=1}^{n_c} \left(score_{c,i} - \left\|\widehat{score_{c,i}}\right\|\right)^2 && \text{(residual sum of squares for rounded values)} \\
R^2_{c,\text{exact}} &= 1 - \frac{SS_{c,\text{res,exact}}}{SS_{c,\text{tot}}} && (R^2 \text{ for exact values of modeled scores}) \\
R^2_{c,\text{round}} &= 1 - \frac{SS_{c,\text{res,round}}}{SS_{c,\text{tot}}} && (R^2 \text{ for rounded values of modeled scores})
\end{aligned}
$$

Figure 18: Equations used to compute the coefficients of determination $R^2_{c,\text{exact}}$ and $R^2_{c,\text{round}}$, for exact values of modeled scores and rounded values, respectively

Number of charsets c per line	$\hat{\alpha}_c$	$\hat{\gamma}_c$	$R^2_{c,\text{exact}}$	$R^2_{c,\text{round}}$
1	3.6000	0.0190	0.9998	1
2	3.5874	0.1572	0.9999	1
3	3.5846	0.1532	0.9999	0.9999
4	3.6073	−0.0579	0.9997	0.9996

Table 4: Estimated coefficients of the one-degree polynomial for each line identified by its number of charsets c included in the passwords with coefficients of determination for exact and rounded values of modeled scores.

The coefficients of determinations are very close or equal to 1, which means that the estimated coefficients $\hat{\alpha}_c$ and $\hat{\gamma}_c$ are accurate. However, since these coefficients are based on rounded observed scores, they may not be very precise, which accounts for their relative variation. Also, given the fact that there is no justification for providing non-zero score to a zero-length password, we round the estimated value for γ to 0. Finally, we believe the coefficient α is derived from a simple expression because of the apparent simplicity of this meter. Also, because the four candidate values for α oscillate around 3.6, we choose to set $\alpha = 3.6$.

Our model becomes the following:

$$score = \left\| 3.6 \cdot length \cdot 1.2^{\#charsets-1} \right\|$$

At last, we need to infer the mapping from scores to labels. Thresholds between each of the six possible categories are as follow: Terrible (0–10), Weak (11-20), Fair (21-40), Good (41-60), Excellent (61-90), or Fantastic (91-100).

The fully reversed-engineered algorithm is presented in Algorithm 2. Note that to accommodate the observed scores for passwords containing a space and only non-recognized characters, we add lines 15 to 17 to our model in the algorithm. Our algorithm yields zero error when evaluated against the remaining of our dictionaries, which may mean that we found the exact formula behind the score calculation.

Browser extension strength meter. Surprisingly, 1Password uses a different password meter in its browser extension. This meter evaluates randomly generated strong passwords, however the user is allowed to modify a generated password as she wishes and the meter reacts to the changes. Adapting a randomly generated password before accepting it for the creation of a new account is a documented behavior in the user guide.[19]

This meter, written in JavaScript, differs from the one ranking the master password in the stand-alone application in three ways. First, it is aware of some leet

[19] `http://help.agilebits.com/1Password3/3_minute_expert.html`

Algorithm 2 1Password master password's checking algorithm

Input: Candidate password

Output: Score in percentage and label related to the strength of the password

```
1: length ← number of bytes taken by the password
2: charset ← 0
3: if password contains lowercase letters then
4:     charset ← charset + 1
5: end if
6: if password contains uppercase letters then
7:     charset ← charset + 1
8: end if
9: if password contains digits then
10:    charset ← charset + 1
11: end if
12: if password contains characters in !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ then
13:    charset ← charset + 1
14: end if
15: if password contains spaces and charset = 0 and length > 0 then
16:    charset ← 1
17: end if
18:
19: score ← ||3.6 · length · 1.2^(charset−1)||
20: if score > 100 then
21:    score ← 100
22: end if
23:
24: if score ≤ 10 then
25:    label ← Terrible
26: else if score ≤ 20 then
27:    label ← Weak
28: else if score ≤ 40 then
29:    label ← Fair
30: else if score ≤ 60 then
31:    label ← Good
32: else if score ≤ 90 then
33:    label ← Excellent
34: else
35:    label ← Fantastic
36: end if
37:
38: return score, label
```

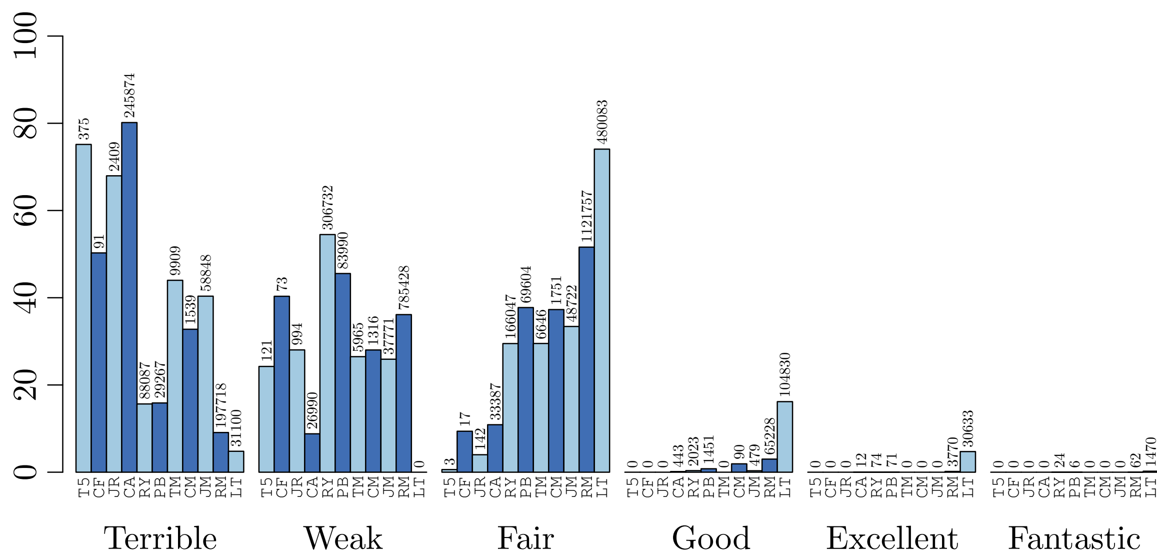


Figure 19: 1Password checker (JavaScript-based) password strength distribution

transformations, then it performs a blacklist check against a 228,268-word dictionary, and finally it differs in the way of identifying groups of characters. The code is abstracted in Algorithm 3.

Label assignment is performed identically to the software meter's label assignment, however the label is only used to color the meter's bar accordingly and is not displayed to the user. We noticed that a coefficient of 1.2 is repeated several times throughout the algorithm to reward the presence of various groups of characters, same as the one we identified in the software version of the meter.

4.4.1.2 Weaknesses

We think it is a bad design choice to implement two meters running different algorithms, as a user may not understand the reasons for the differences in ranking. In addition, the master password that protects all secrets is guided by the worst of both meters. This meter is only dependent on the length and number of charsets used in a password, and lacks any further checks such as blacklisting, pattern matching, transformations, etc. Thus, it is unable to catch many weak passwords. Moreover,

Algorithm 3 1Password Firefox extension's checking algorithm

Input: Candidate password
Output: Score in percentage related to the strength of the password

1: *length* $\leftarrow$ number of characters in the password
2: **if** $length = 0$ **then**
3:     **return** 0
4: **else if** $length < 3$ **then**
5:     **return** 2
6: **else**
7:     $score \leftarrow 3 \cdot (length - 3)$
8: **end if**
9:
10: *coreword* $\leftarrow$ *password* in lowercase
11: Replace 4, 1, 3, 7, 5, 0 by a, l, e, t, s, o in *coreword*
12: Remove non-letter characters in *coreword*
13: **if** *coreword* is included in one of the dictionary words **then**
14:     $score \leftarrow score - 3 \cdot$ (number of characters in *coreword*)
15: **end if**
16: **if** $score < 0$ **then**
17:     $score \leftarrow 0$
18: **end if**
19: **if** password contains lowercase letters **then**
20:     $score \leftarrow 1.2 \cdot score$
21: **end if**
22: **if** password contains uppercase letters **then**
23:     $score \leftarrow 1.2 \cdot score$
24: **end if**
25: **if** password contains a space followed by a tab **then**
26:     $score \leftarrow 1.2 \cdot score$
27: **end if**
28: **if** password contains characters in `` $+<=>^`|~ `` **then**
29:     $score \leftarrow 1.2 \cdot score$
30: **end if**
31: **if** password contains characters in `!"%'(),-./:;?][\{}` **then**
32:     $score \leftarrow 1.2 \cdot score$
33: **end if**
34: **if** password contains digits **then**
35:     $score \leftarrow 1.2 \cdot score$
36: **end if**
37: **if** $score > 100$ **then**
38:     $score \leftarrow 100$
39: **end if**
40: $score \leftarrow \lfloor score \rfloor$
41: **return** *score*

due to the way labels are assigned, most passwords fall under the "fair" category including most of Top500 dictionary, even though the meter possesses two labels for inferior strengths. Examples of weak passwords labeled as good include *password123*, *princess12* and *123456789a*. Excellent passwords include *123456789123456789* and *1q2w3e4r5t6y7u8i9o0p*. Finally, *abcdefghijklmnopqrstuvwxyz* is fantastic.

4.4.1.3 Strengths

The browser extension performs better than its stand-alone application counterpart. It is able to assign scores between "terrible" and "fair" in a sound way, i.e., the most simple dictionaries (Top500, Cfkr and JtR) are mostly assigned terrible and weak scores, while their mangled versions only achieve a fair score. The algorithm is able to detect some leet transformations and blacklist a password found in its dictionary even with 3 additional non-letter characters placed at the beginning and/or the end of the password, which accounts for many of our mangling rules.

4.4.2 LastPass

LastPass is a password manager extension for browsers. Users create a LastPass account using a master password that is evaluated by a meter. The password further encrypts the password vault, stored online. The service handles identity information for the user and helps with the creation of new web service accounts with a dedicated dialog box that is prefilled with a random passwords that the user can further modify (as with 1Password browser extension). The meter does not come with a label associated with the evaluated strength, and provides only a score between 0 and 100. We report strengths categorized by range in Figure 20 for LastPass 3.1.1 for Firefox (as of March 25, 2014).

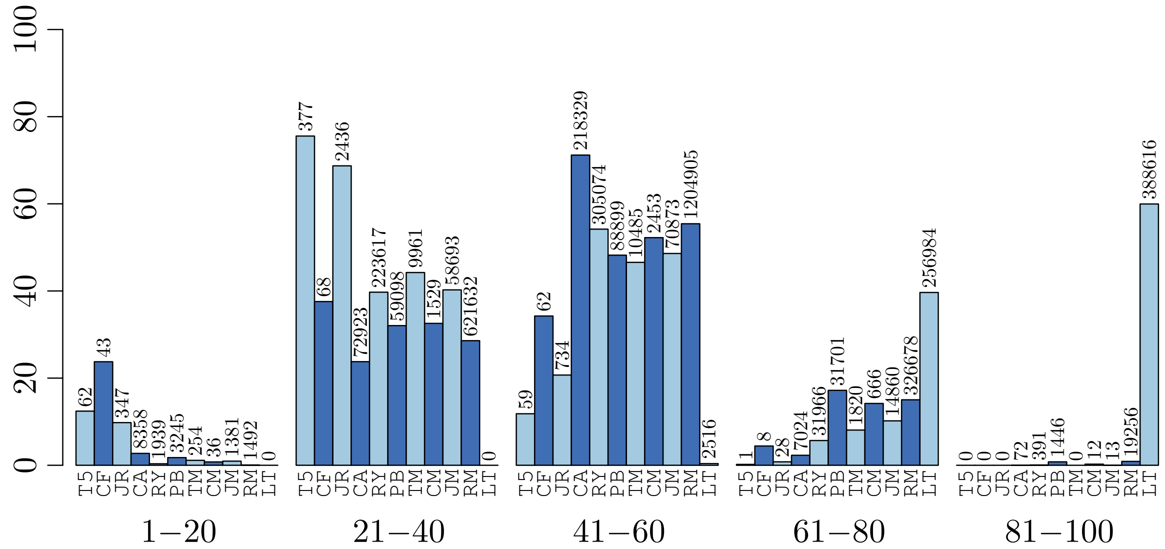


Figure 20: LastPass checker password strength distribution. Scores represent a percentage and are grouped into five equal ranges.

4.4.2.1 Algorithm

LastPass is a simple browser extension written in JavaScript and is hence relatively easy to understand. The strength measurement algorithm is 28 lines of code and performs various checks as follows. If the password is exactly equal to the email address of the user (while creating a LastPass account), the password receives a score of 1. If the password is contained in the email address, the strength is penalized by 15 (that is, the strength starts at −15 instead of zero). Conversely, if the email address is contained in the password, the strength is penalized by a percentage equal to the length of the email address. If the password contains only one unique character (despite repetitions), the score is 2. The password length is then added to the strength and numerous patterns are checked sequentially. Patterns are listed in Table 5. Finally, the strength is doubled and truncated between 0 and 100.

Pattern	**Reward**
$0 < length \leq 4$	$+length$
$5 < length \leq 7$	+6
$8 < length \leq 15$	+12
$15 < length$	+18
Lowercase letters	+1
Uppercase letters	+5
Digits	+5
At least 3 digits	+5
Symbols (taken from `!@#$%^&*?,_~`)	+5
At least 2 symbols	+5
Both lowercase and uppercase	+2
Lowercase, uppercase and digits together	+2
Lowercase, uppercase, digits and symbols together	+2

Table 5: LastPass patterns rewarding rules. The reward value is added to the strength score if the password matches the corresponding pattern.

4.4.2.2 User information

The password is verified against the email address by more than a simple equality check, accounting for one included in the other. However, it is questionable whether the full email address will be included in the password (although we find examples of this in our dictionaries). For the case where only the username part may be reused in the password, the check is bypassed by adding an extra character, e.g., email address *john.doe@email.com* and password *john.doe1* are allowed.

4.4.2.3 Weaknesses

The password *password* already ranks 42%, although it is arguably not *the answer to the Ultimate Question of Life, the Universe, and Everything.*[20] *Password1* also ranks 72% because of the number of charsets included in it and the relatively long length. Clearly, this checker focuses on rewarding a password rather than appropriately reducing its strength by additional checks of simple patterns. Rules are mostly sensitive to the password composition, ignoring other patterns or common passwords. Leet transformations are also not taken into account.

[20] The Hitchhiker's Guide to the Galaxy, Douglas Adams

4.4.3 KeePass

KeePass password meter appears during the creation of a new password-encrypted vault and creation of a new entry in a vault, and represents the strength of a given password as both an entropy score in bits and a progress bar filled at 128 bits. We analyze KeePass version 2.25 for Windows (as of March 25, 2014). KeePass is an open-source software whose algorithm is briefly described on the corresponding KeePass Help Center's webpage.[21] The translation to labels provided on this webpage suggests the following mapping: very weak (0–64), weak (64–80), moderate (80–112), strong (112–128), very strong (128+); however, we report strengths grouped by uniform ranges as users are not provided with this non-linear label assignment. Moreover, this label mapping is apparently unrealistic since a password would be considered very weak until it reaches 50% of the meter and weak until 63% (which is already in a green area). Figure 21 summarizes our results.

4.4.3.1 Algorithm

KeePass runs a complex algorithm against passwords to be ranked, which detect several patterns in way similar to Dropbox checker. This algorithm checks any repeated substrings of length 3 or more in a password (e.g., as in *passwordpassword* or *winter14winter*), identify groups of 3 digits or more, identify sequences of 3 characters or more whose UTF-8 code values are separated by a regular interval (e.g., abcdef and 123456 with a distance of 1, acegi and 86420 with a distance or 2 and -2, respectively), and performs a leet-aware dictionary check against a list of 10,183 words.

KeePass distinguishes between lowercase (L)/uppercase (U) letters, digits (D), symbols from a US keyboard (S), along with an extended set of international letters and symbols (H) from code points `0xA1` to `0xFF` (upper part of the extended ASCII), and denotes other characters in set X; characters in X are ignored in the rest of the algorithm. Patterns are further classified as R for repetitions, W for words in the dictionary, C for sequences of characters and N for groups of digits. The password

[21] `http://keepass.info/help/kb/pw_quality_est.html`

Figure 21: KeePass checker password strength distribution

is decomposed into all possible (with likely overlapping) patterns, each having an entropy computed based on the type of pattern. The pattern detection algorithm is abstracted in Algorithm 4. Note that when detecting patterns of a certain type (R, X, C or N), the longest ones are matched first, from left to right, and a detected pattern cannot overlap with another one of the same type. E.g., in *Password123*, *Password1* is detected as a dictionary word (pattern W), however, *Password* alone will be ignored to reduce the number of pattern combinations that would overestimate the strength of the password. The representation of *Password123* becomes the following:

W										
P	a	s	s	w	o	r	d	1	2	3
U	*L*	*L*	*L*	*L*	*L*	*L*	*L*	*D*	*D*	*D*
								N		
								C		

The overall entropy for each combination of patterns is calculated, where simple characters that are not part of further detected patterns are treated as a single-character pattern belonging to the group of characters previously identified (L, U,

D, S, or H). In the example above, all combinations include ULLLLLLLDDD, ULLLLLLLN, ULLLLLLLC and WDD. In contrast to Dropbox, which simply combines entropies together by addition, KeePass relies on a "optimal static entropy encoder" (as mentioned on KeePass Help Center webpage). This encoder computes the entropy of each pattern considering the size of the group it belongs to if not already assigned an entropy by the pattern detection algorithm, and takes into account the distribution of characters inside each group and adjusts the calculation for repeated individual characters. The score is assigned with the entropy of the combination of patterns that yields the minimum value. Compared to Dropbox's algorithm, KeePass yields overall a slightly stronger combined entropy.

4.4.3.2 Strengths

A special extended symbol charset takes into account international symbols in a different way than symbols on a US keyboard. This is one of the rare attempts to consider internationalized passwords. The translation table for leet transformations is the most comprehensive one among the meters we tested, including more than a hundred transformations which also take into account the simplification of accented letters (e.g., é → e). This algorithm decently rejects our simple dictionaries and their mangled and leet-transformed versions without involving a stringer policy. In fact, no policies are enforced on password ranking. Such behavior is remarkable since its response profiles is likely to be realistic in a real-world usage, contrary to other profile showing good results with too simple checks.

4.4.3.3 Weaknesses

Although extensive checks are performed to rate a password, specially for sequences, the algorithm doesn't include checks for spacial combinations on a keyboard, such as *a1s2d3f4g5h6j7k8*, which is ranked 81 bits (63%). Also, regarding passphrases, KeePass is still primarily intended for traditional passwords that contain random characters or one or two core words. No special consideration is given to multiple

Algorithm 4 KeePass password checker pattern detection algorithm

Input: Password

Output: Entropy of the password

1: Decompose the password onto the 6 sets of characters (L, U, D, S, H, X)
2: **for each** repetitive patterns in password **do**
3: Consider identified substring as pattern R
4: Assign repetitive occurrences with entropy of $\log_2$(pattern offset $\times$ pattern length) ▷ Offset count starts at 1
5: **end for**
6:
7: **for each** *number* of 3 digits or more **do**
8: Consider identified substring as pattern N
9: Assign *number* an entropy of $\log_2(number)$
10: **if** *number* has leading zeros **then**
11: Add $\log_2$(number of leading zeros $+ 1$) to the entropy assigned
12: **end if**
13: **end for**
14:
15: **for each** Sequence of 3 characters or more whose code points are distanced by a constant **do**
16: Consider identified substring as pattern C
17: Assign pattern an entropy of $\log_2$(charset size matching the first character $\times$ (sequence length $- 1$))
18: **end for**
19:
20: **for each** substring of password **do**
21: $size \leftarrow$ number of words of same length in dictionary
22: **if** substring included in lowercase in dictionary **then**
23: Consider identified substring as pattern W
24: $distance \leftarrow$ number of differences between substring and word in dictionary
25: Assign substring an entropy of $\log_2(size \cdot \binom{n}{distance})$
26: **else if** unleeted substring included in lowercase in dictionary **then**
27: Consider identified substring as pattern W
28: $distance \leftarrow$ number of differences between substring and word in dictionary
29: Assign substring an entropy of $1.5 \cdot distance + \log_2(size \cdot \binom{n}{distance})$
30: **end if**
31: **end for**

words; e.g., *love is in the air* ranks 76 bits (59%). This example is as good as the 13-character random password _!X>g$r6^G+MX_ (75 bits), which also highlights the stringency of this meter. In fact, only carefully-chosen 20-character long random passwords can successfully fill the meter, e.g., _&b^p6Mvm97Hc#$1Oa*S5_ (129 bits), which makes KeePass as the most stringent meter we analyzed.

4.4.4 RoboForm

RoboForm is a password manager that comes as a software tool and a browser extension. Unfortunately, the browser extension calls functions implemented in the software through a Dynamic-Link Library for Windows and hence the code for evaluating passwords is not written in JavaScript. Based on the evaluation of previous meters, the profile of this checker shown in Figure 22 doesn't seem to follow a simple logic. We tried to infer its logic by applying the same technique as used with 1Password without success, as no pattern emerged from various representations of password scores. To understand how the algorithm treats passwords, we reverse-engineer `identities.exe` from RoboForm version 7.9.5.7 for Windows (as of March 25, 2014), which handles the creation of a master password with the help of a meter, and understand the types of checks the meter performs.

4.4.4.1 Algorithm

The step-by-step execution of the program reveals several checks run against a candidate password. We could identify four simple functions for detecting if a given character is a lowercase or uppercase letter, a non-letter, a digit, or a symbol (only special characters found on a US keyboard are taken into account). This gives insights as to the types of checks involved in the algorithm, which are mainly focused on password composition. For each detected pattern, a penalty or reward is added to the overall score. To the best of our understanding, checks include the presence of digits only, count of non-lowercase characters, count of non-letter characters, presence of both digits and symbols, count of repetitions. More advanced patterns are

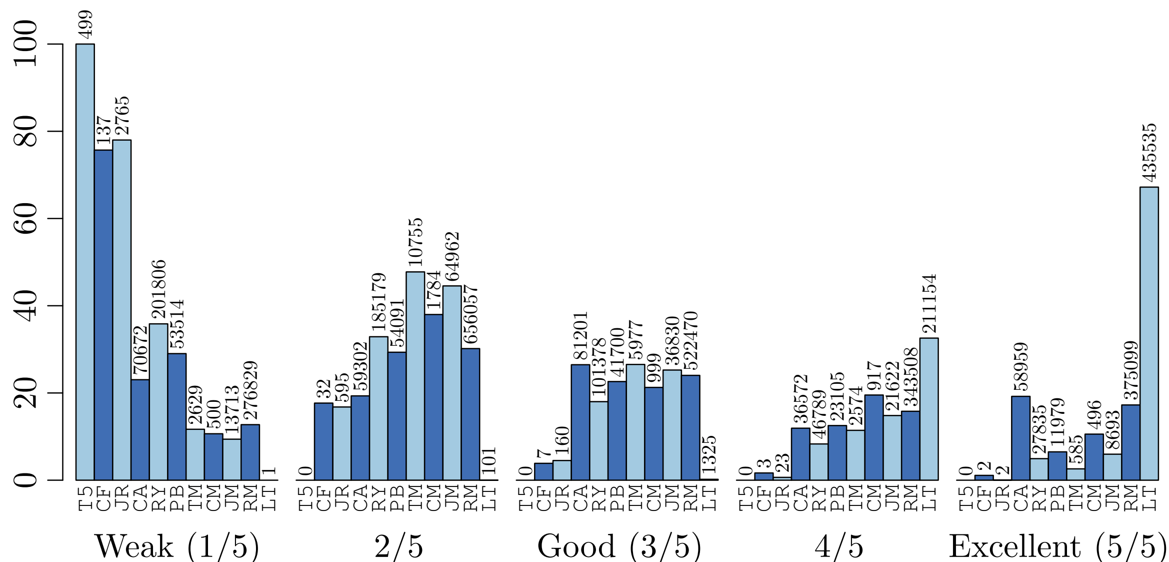


Figure 22: RoboForm checker password strength distribution

searched such as the succession of two characters that are typed from the same key on a US keyboard (e.g., "a" and "A", or "5" and "%") or from adjacent keys, along with simple keyboard sequences (e.g., *q1w2e3r4t5y*). A blacklist check is also performed against an embedded dictionary of 29,080 words and keyboard sequences. We did not cover all the checks run by RoboForm, nor did we write a full-fledge equivalent algorithm since we gained enough insights to understand its strengths and weaknesses illustrated in its response profile. Also, understanding all the details of the algorithm from reverse-engineering the binary file is a time-consuming task. To illustrate this, we present Figure 23, which represents the part of the assembly instructions responsible for counting all identical consecutive characters and penalizing the score accordingly. These 53 lines of x86 assembly are semantically identical to the few-line algorithm presented in Algorithm 5. This example highlights the complexity of simple tasks when written in assembly and the difficulty to infer the high-level idea of the algorithm by such analysis.

```
548FEBD0  MOV EDX,[DWORD SS:EBP-64]      ;load i from EBP-64 in EDX
548FEBD3  ADD EDX,1                      ;increment i in EDX
548FEBD6  MOV [DWORD SS:EBP-64],EDX      ;store back i (i=i+1)
548FEBD9  MOV EAX,[DWORD SS:EBP+8]       ;load address of a pointer to password in EAX
548FEBDC  MOV ECX,[DWORD DS:EAX]         ;load address of password in ECX
548FEBDE  MOV EDX,[DWORD DS:ECX-14]      ;load password length in EDX
548FEBE1  MOV [DWORD SS:EBP-130],EDX     ;save password length at EBP-130
548FEBE7  MOV EAX,[DWORD SS:EBP-130]     ;load password length in EAX
548FEBED  SUB EAX,1                      ;decrement length of password in EAX
548FEBF0  CMP [DWORD SS:EBP-64],EAX      ;compare (password length-1) and i
548FEBF3  JGE RoboForm.548FECB9          ;if i > length-1, finished scanning password
548FEBF9  CMP [DWORD SS:EBP-64],0        ;compare i and 0
548FEBFD  JL SHORT RoboForm.548FEC18     ;if i < 0, error, go to error1
548FEBFF  MOV ECX,[DWORD SS:EBP+8]       ;load address of a pointer to password in ECX
548FEC02  MOV EDX,[DWORD DS:ECX]         ;load address of password in EDX
548FEC04  MOV EAX,[DWORD DS:EDX-14]      ;load password length in EAX
548FEC07  MOV [DWORD SS:EBP-134],EAX     ;save password length at EBP-134
548FEC0D  MOV ECX,[DWORD SS:EBP-64]      ;load i from EBP-64 in ECX
548FEC10  CMP ECX,[DWORD SS:EBP-134]     ;compare i with password length
548FEC16  JLE SHORT RoboForm.548FEC2F    ;if i <= password length, jump after error1
error1: [...]
548FEC2F  MOV EDX,[DWORD SS:EBP+8]       ;load address of a pointer to password in EDX
548FEC32  MOV EAX,[DWORD DS:EDX]         ;load address of password in EAX
548FEC34  MOV ECX,[DWORD SS:EBP-64]      ;load i from EBP-64 in ECX
548FEC37  MOV DX,[WORD DS:EAX+ECX*2]     ;load character i from the password
548FEC3B  MOV [WORD SS:EBP-136],DX       ;save character i at EBP-136
548FEC42  MOV EAX,[DWORD SS:EBP-64]      ;load i from EBP-64 in EAX
548FEC45  ADD EAX,1                      ;increment i in EAX
548FEC48  MOV [DWORD SS:EBP-140],EAX     ;save resulting (i+1) at EBP-140
548FEC4E  JS SHORT RoboForm.548FEC6C     ;if (i+1) becomes negative, go to error2
548FEC50  MOV ECX,[DWORD SS:EBP+8]       ;load address of a pointer to password in ECX
548FEC53  MOV EDX,[DWORD DS:ECX]         ;load address of password address in EDX
548FEC55  MOV EAX,[DWORD DS:EDX-14]      ;load password length in EAX
548FEC58  MOV [DWORD SS:EBP-13C],EAX     ;save password length in EBP-13C
548FEC5E  MOV ECX,[DWORD SS:EBP-140]     ;load (i+1) in ECX
548FEC64  CMP ECX,[DWORD SS:EBP-13C]     ;compare i and password length
548FEC6A  JLE SHORT RoboForm.548FEC83    ;if i <= password length, jump after error2
error2: [...]
548FEC83  MOV EDX,[DWORD SS:EBP+8]       ;load address of a pointer to password in EDX
548FEC86  MOV EAX,[DWORD DS:EDX]         ;load address of password in EAX
548FEC88  MOV ECX,[DWORD SS:EBP-140]     ;load (i+1) in ECX
548FEC8E  MOV DX,[WORD DS:EAX+ECX*2]     ;load character i+1 from the password
548FEC92  MOV [WORD SS:EBP-142],DX       ;save character i+1 at EBP-142
548FEC99  MOVZX EAX,[WORD SS:EBP-136]    ;load character i in EAX
548FECA0  MOVZX ECX,[WORD SS:EBP-142]    ;load character i+1 in ECX
548FECA7  CMP EAX,ECX                    ;is character i = character i+1?
548FECA9  JNZ SHORT RoboForm.548FECB4    ;if not, go check next pair of characters
548FECAB  MOV EDX,[DWORD SS:EBP-10]      ;else, load number of repetition
548FECAE  ADD EDX,1                      ;increment it
548FECB1  MOV [DWORD SS:EBP-10],EDX      ;store it back
548FECB4  JMP RoboForm.548FEBD0          ;go check next pair of characters

548FECB9  MOV EAX,[DWORD SS:EBP-10]      ;load number of repetitions
548FECBC  IMUL EAX,EAX,-1                ;tmp = -(number of repetitions)
548FECBF  IMUL EAX,EAX,3                 ;tmp = tmp*3
548FECC2  ADD EAX,[DWORD SS:EBP-28]      ;score = score+tmp
```

Figure 23: RoboForm disassembled instructions for checking identical consecutive characters

Algorithm 5 Algorithm for RoboForm's identical consecutive characters check

Input: Password, currently evaluated score
Output: Penalized score according to the number of identical consecutive characters

1: $repetition \leftarrow 0$
2: **for** $i = 0$ to password length $- 1$ **do**
3:     **if** $password[i] = password[i + 1]$ **then**
4:         $repetitions \leftarrow repetitions + 1$
5:     **end if**
6: **end for**
7: $score \leftarrow score - 3 \times repetitions$

4.4.4.2 Strengths

The algorithm checks for several patterns with a particular emphasis given to keyboard sequences. RoboForm is the only algorithm that embed a list of sequences in its dictionary, in addition to the ones that are simple to check programatically. It also catches the most simple dictionaries (Top500, Cfkr and JtR) and assigns them a score below good.

4.4.4.3 Weaknesses

RoboForm however fails at detecting many mangled passwords and does not consider of leet transformations. A fair amount of our dictionaries are ranked better than good and even excellent. Among such good passwords, we find *Password1* ranked 4/5 (between good and excellent), *Basketball* and *A1b2c3d4*. The latter is surprising given the many keyboard patterns the meter checks. These three examples are taken from Top500+M that includes variations of the most simple dictionary. The dictionary check is able to penalize words that are included in one of the blacklisted words irrespective of the case, however, any additional characters bypass this check resulting in score jumps, e.g., *password* is weak (1/5) while *password1* is ranked 4/5.

Chapter 5

Results Analysis

Below, we further analyze our results from Chapter 4, list several common features and weaknesses, and compare the meters. We also discuss a web-based password multi-checker tool that we implemented for testing any password across all the web-based meters we evaluated. Beyond the generic dictionaries as discussed in Section 3.3.

5.1 Results Summary

Most checkers we evaluated heavily rely on the presence of characters included into the four main charsets (lowercase/uppercase letters, digits and symbols); hence we name their category as LUDS. LUDS checkers, specially if combined with a dictionary check, mainly reward random-looking passwords and push users to select such passwords (or to bypass the check, as discussed in Section 5.6), and tend to disregard passphrase-looking passwords that include only lowercase letters. Among the 18 LUDS checkers we evaluated, 12 of them combine the charset information with the password length. Only 8 of the LUDS checkers involve a dictionary check, for which only 4 involve a smarter check that takes into account mangling and leet transformations. Also, 6 of them perform further checks for patterns, of which only 4 search for patterns to increase the score given to a password. Among the remaining 3 non-LUDS checkers, 2 of them (Dropbox and KeePass) are mainly based on more advanced patterns such as

sequences, repetitions, dictionary words with mangling and leet-awareness, keyboard patterns, human-readable patterns (e.g., dates), combined with a conservative entropy calculation. These two checkers yield sound response profiles to our test dictionaries, i.e., they rank the base dictionaries as weak and the mangled and leet ones only slightly better. The last checker, Google, remains a mystery to us because of its black-box nature and inconsistent output. Section 5.7 provides some possible explanation of Google's algorithm. Table 6 summarizes the types and capabilities of the evaluated meters.

As expected, embedded dictionaries of the meters involving a blacklist check overlap with parts of our dictionaries. This overlap is detailed in Table 3. We note that the majority of these embedded dictionaries contain a significant portion of the base dictionaries (Top500, Cfkt and JtR); however, they never include any of them completely. There is a significant similarity between 1Password and C&A dictionaries, with 99.58% of the former included in the latter, and 74.11% of C&A included in 1Password dictionary. Similarly, Top500 and Twitter dictionaries are close to each other, with 93.77% of Twitter being included in Top500, and 75.35% of Top500 being included in Twitter dictionary. Apart these two exceptions, no other dictionaries share a significant part of their content. Also, Dropbox dictionary is a superset of Intel, KeePass and Twitter, with only few exceptions not included in Dropbox. Mangled versions of our dictionaries do not include and are not included into any other dictionaries by more than few percents. This observation seems logical since checks against dictionaries should be sanitized from possible transformations before being checked. From these results, we wonder how meters construct their embedded dictionaries and why do they not include obvious dictionaries such as Top500 and JtR in their entirety.

Name	Type	Length	Patterns	Dictionary and variations			
				Basic	Leet	Mangling	Multiple
Dropbox	Advanced patterns	✓	↑↓	↓	✓	✓	✓
Drupal	LUDS	×	×	×	–	–	–
FedEx	LUDS	✓	×	×	–	–	–
Intel	LUDS	✓	×	⇓	×	✓	✓
MS v1	LUDS	✓	×	⇓	✓	×	×
MS v2	LUDS	✓	×	×	–	–	–
MS v3	LUDS[1]	✓	×	×	–	–	–
QQ	LUDS	×	×	×	–	–	–
Twitter	LUDS	✓	↑	⇓	×	×	×
Yahoo!	LUDS	×	×	×	–	–	–
12306.cn	LUDS	×	×	×	–	–	–
eBay	LUDS	×	×	×	–	–	–
Google	?	✓	?	↓	×	×	✓
Skype	LUDS	✓[2]	×	×	–	–	–
Apple	LUDS	✓	↑	⇓	×	×	×
PayPal	LUDS	×	↓	⇓	×	✓	×
1Password (software)	LUDS	✓	×	×	–	–	–
1Password (browser)	LUDS	✓	↑	↓	✓	✓	×
LastPass	LUDS	✓	↑	×	–	–	–
KeePass	Advanced patterns	✓	↑↓	↓	✓	✓	✓
RoboForm	LUDS	✓	↑↓	↓	×	×	×

[1] Charsets check is only taken into account for the strongest label
[2] Length check is only taken into account for the strongest label

Table 6: Summary of the types of meters and their capabilities. Notation used under "Type": LUDS (Lowercase/Uppercase/Digit/Symbol) is a type of meter that is mainly sensitive to the number of charsets, "?" means we are unsure. "Length": whether the meter includes the password length in its calculation beyond a minimum requirement. "Patterns": ↑ denotes checks for rewarding patterns, ↓ denotes checks for penalizing patterns, "?" means we are unsure. "Dictionary and variations": The column "Basic" denotes whether at least a simple dictionary check is performed; ↓ means the strength is only partially impacted by a positive match, ⇓ means the strength goes down to the minimum by a positive check. "Mangling" represents whether a dictionary check prevents bypassing by addition of few other characters before or after the detected word. "Multiple" represents whether the meter is able to detect multiple dictionary words inside the password. In the case dictionary checks are not performed, "Leet", "Mangling" and "Multiple" are not applicable (noted with a "–").

5.2 Meters Heterogeneity and Inconsistencies

In general, each meter reacts differently to our dictionaries, and strength results vary widely from one to another. For example, Microsoft v2 and v3 checkers assign their best score to only a very small fraction of our passwords, while Google assigns its best score to almost 2.2 million of them (about 56%). For individual checkers, some simple dictionaries score significantly higher than others, e.g., Top500 and JtR when tested against Twitter. 75% of Top500 words are considered obvious and the rest are too short; however, 58% of JtR words are considered "Could be More Secure" (2 or 3 steps up from Top500). As for individual passwords, probably one of the most stunning result is *Password1* that receives the widest panel of possible scores, ranging from very weak for Dropbox to very strong for Yahoo!. It also receives three different scores by Microsoft checkers (i.e., strong, weak and medium chronologically). While our leet dictionary is mostly considered strong by Microsoft v1, it becomes mainly weak in v2, and finally medium in v3. Such inconsistent jumps demonstrate the relativity of password strength even by the same vendor at different times.

Some inconsistencies are particularly evident when a password passes the minimum requirements. For example, *password$1* is correctly assigned very-weak by FedEx, but the score jumps to very-strong when the first letter is uppercased. Such modifications are normally considered very early in a cracking algorithm; hence such a jump is unlikely to match reality. Similarly, *qwerty* is tagged as weak by Yahoo!, while *qwerty1* jumps to strong; *password0* as weak and *password0+* as strong by Google. Finally, as expected, a random password *+^v16#5{]($ is rated as strong by most checkers (or at least medium by Microsoft v3 and eBay); surprisingly, FedEx considers it as very-weak. These problems can be mostly attributed to the stringent minimum requirements.

One possible consequence of these heterogeneous behaviors is the confusion of users with regard to the security of their passwords. When opposite strength values are given for the same password by different services, users may not understand the reason

behind it, which may decrease their trust and willingness to comply with password policies. It may also encourage them to search for easy tricks to bypass stringent restrictions rather than reconsidering their password. Also, permissive meters may drive users to falsely assume their weak password as strong, and provide only a false sense of security (cf. Heijningen [29]), which in turn may encourage users to reuse such weak passwords for other more sensitive accounts. However, the real effects of wrong/incoherent meter outcomes on users may be demonstrated only by a large-scale user study.

We combine the functionalities of all tested checkers in a web-based password multichecker that computes the strength of a given password against all checkers in realtime; see Figure 24 for a screenshot of the tool. Inconsistencies (if exist) in password rating become instantly evident from the output of a given password. This tool can also be used to choose a password that is considered strong by all meters, increasing the chances of that password being effectively strong. Finding such passwords may also help users select few strong passwords to reuse, but note that although password reuse is a common practice, it is generally considered bad for security. The tool is available at: `https://madiba.encs.concordia.ca/software/passwordchecker/index.php`.

5.3 Comparison

In Chapter 4, we provide results of individual meter evaluation. Here, we compare the meters against each other. As strength scales vary significantly in terms of labels and the number of steps in each scale (see Table 1), we simplified the scales for our comparison. Figures 25a and 25b show the percentage of the dictionaries that are tagged with an *above-average* score by the different web services, sorted by decreasing cumulative percentages. To be conservative, we choose to count only the scores labeled at least “Good”, “Strong” or “Perfect”. For KeePass, we count scores greater than 64 bits of entropy (meter’s bar goes to 128). For LastPass, we count scores greater

Password Multi-Checker

password$1

Services	Strength scores	
Apple	Moderate	2/3
Dropbox	Very Weak	1/5
Drupal	Strong	4/4
eBay	Medium	4/5
FedEx	Very Weak	1/5
Google	Fair	3/5
Intel	Oh No!	1/2
Microsoft (v1)	Strong	3/4
Microsoft (v2)	Medium	2/4
Microsoft (v3)	Medium	2/4
PayPal	Weak	2/4
QQ	Strong	4/4
Skype	Poor	1/3
Twitter	Perfect	6/6
Yahoo!	Very Strong	4/4
12306.cn	Average	2/3

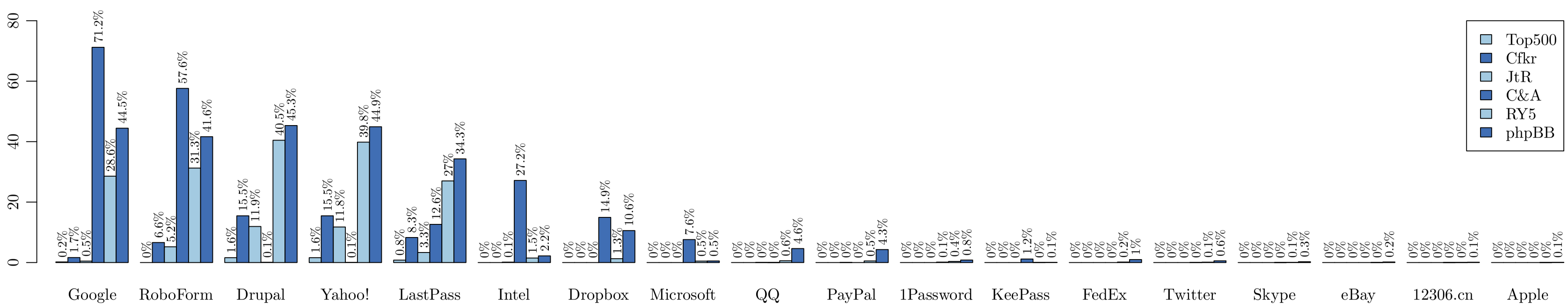
Figure 24: Password multi-checker output for *password$1*. A strength score of "2/3" denotes the relative position (2) in a given strength scale (3).

than 50%. Clearly, such scores should not be given to most of our test set (possible exceptions could be the complex passwords from leaked dictionaries).

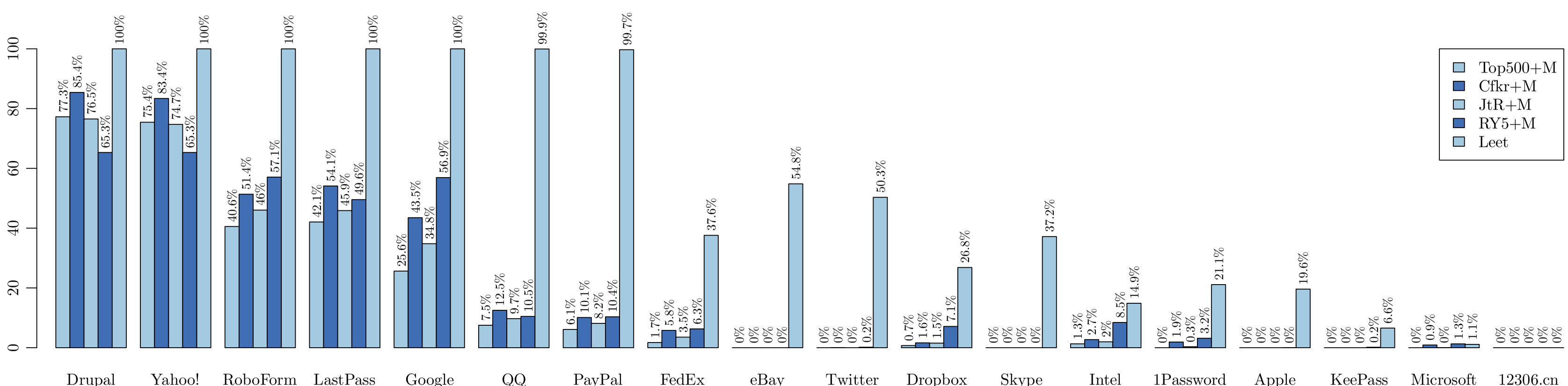
In reality, Google, RoboForm, Drupal, Yahoo! and LastPass assign decent scores to passwords from our base dictionaries; see Figure 25a. Significant percentages of Top500, Cfkr and JtR are qualified as good by Drupal and Yahoo!, which are about 1.6%, 15.5%, and about 12% respectively for both checkers. Also, roughly 40% of RY5 and 45% of phpBB passwords are tagged as good by both Drupal and Yahoo!. This similarity in the results possibly originates from the simple design of their meters, which perform similar checks. Google assigns good scores to 71.2% of C&A, 28.6% of RY5 and 44.5% of phpBB. Other checkers mostly categorize our base dictionaries as weak.

The mangled and leet dictionaries trigger more distinctive behaviors. Drupal, Yahoo!, RoboForm, LastPass and Google still provide high scores with a minimum of 25.6% given to Top500+M and up to 100% to Leet. Google also rates 100% of Leet as good or better. Leet also completely bypasses Microsoft v1 and PayPal. Overall, it also scores significantly higher than other dictionaries against FedEx, eBay, Twitter, KeePass, Dropbox, Skype, Intel, 1Password, Microsoft v2 and Apple. Only Microsoft v3 is able to catch up to 98.9% of this dictionary (due to the use of a very stringent policy).

Our comparison graphs are not a competitive ranking of the meters. Although the ones that evaluate many of our dictionaries as good certainly have design deficiencies, it does not mean the remaining meters have a correct design. For example, 12306.cn seems to be the "best" checker based on Figure 25b as it does not classify any of our passwords as good. However, it achieves this performance only by making the presence of the underscore character "_" mandatory (along with digits and mixed-case letters) for a password to be ranked as good. Clearly, this particular requirement doesn't reflect a good design. Hence, Figures 25a and 25b alone should be treated mostly as the number of false positives for each meters, i.e., the number of weak passwords misclassified as good, but does not claim anything about the true positives



(a) Comparison based on our base dictionaries



(b) Comparison based on our mangled and leet dictionaries

Figure 25: Comparison between services assigning decent scores to (a) our base dictionaries, (b) our mangled and leet dictionaries. Microsoft is represented by its latest checker and 1Password is represented by its browser extension's checker.

(real strong passwords detected as such), the false negatives (strong passwords labeled as weak) and true negatives (weak passwords labeled as such).

5.4 International Characters

We have not tested passwords with international characters due to the lack of dictionaries with a considerable number of such passwords. International characters are also usually not properly handled for web passwords (see e.g., Bonneau and Xu [12]). We briefly discuss how such characters are taken into account for strength calculation by different checkers.

International characters, when allowed, are generally considered as part of the symbols charset (or "others" by Microsoft v2). However, this charset is limited to specific symbols for 12306.cn, 1Password, Apple, eBay, FedEx, Google, LastPass, Microsoft, PayPal, RoboForm, Skype, Tencent QQ, Twitter, and Yahoo! (that is, all except Dropbox, Drupal, Intel and KeePass). Google prevents the use of international characters altogether, while Apple allows some of them below ASCII code 193 but does not count them in any charset.

As for character encoding, passwords in the tested server-side and hybrid checkers are always encoded in UTF-8 prior to submission. This is because the registration pages are rendered in UTF-8, and browsers usually reuse the same encoding for form inputs by default [12]. Passwords are also correctly escaped with percentage as part of the URI encoding by Apple, eBay, Google and Skype. However, PayPal shows an interesting behavior in our tests: it sends the HTTP Content-Type header `application/x-www-form-urlencoded; charset=UTF-8`, meaning that a properly URI encoded string is expected as POST data. However, no encoding is performed and characters that require escaping are sent in a raw format, e.g., `search_str=myspace1&PQne)!(4`, where the password is *myspace1&PQne)!(4*. The character `&` interferes with the parsing of `search_str` argument and the remaining of the password (`PQne)!(4`) is dropped from the check. Then, because *myspace1* is

blacklisted, the entire password is blacklisted. However, removing the ampersand makes the entire password being evaluated, which in turn is not blacklisted, and even tagged as strong. UTF-8 characters are also sent in a raw format (the proper Content-Type should be `multipart/formdata` in this case [12]). To get the same output as the PayPal website, we carefully implemented this buggy behavior in our tests and multichecker tool.

5.5 Implications of Design Choices

Client-side checkers as tested in our study can perform either very stringently (e.g., FedEx, Microsoft v2), or very loosely (e.g., Drupal). Server-side checkers may also behave similarly (e.g., Skype vs. Google). Finally, hybrid checkers behave mostly the same as client-side checkers with an additional (albeit primitive) server-side blacklist mechanism. Apparently, no specific checker type, web- or application-based, outperforms others. Nevertheless, server-side checkers inherently obscure their design (although it is unclear how these checkers are benefited by such a choice). Along with hybrid checkers, a blacklist can be updated more easily than if it is hard-coded in a JavaScript code. Most checkers in our study are also quite simplistic: they do not perform extended computation, but rather apply simple rules with regard to password length and charset complexity, and sometimes detect common password patterns. This remark also stands for server-side checkers that would eventually mandate a server's computation power. Dropbox is the only exception, which uses a rather complex algorithm to analyze a given password by decomposing it into distinguished patterns. It is also the only checker able to rate our leet dictionary most effectively, without depending on stringent policy requirements (as opposed to Microsoft v2 and v3 checkers).

5.6 Stringency Bypass

Users may adopt simple mangling rules to bypass password requirements and improve their password strength score [58]. However, most checkers (except Dropbox, Intel, KeePass, PayPal and 1Password for Firefox), apparently disregard password mangling. Even trivial dictionaries when mangled, easily yield better ranked passwords. For example, Skype considers 10.5% of passwords as medium or better, when we combine (Top500, C&A, Cfkr and JtR) dictionaries; for the mangled version of the combined dictionary, the same rating is resulted for 78% of passwords. This gap is even more pronounced with Google, where only five passwords from the combined dictionary are rated strong (0.002%), while tens of thousands from the mangled version (26.8%) get the same score. Our mangled dictionaries are built using only simple rules (e.g., do not result in 4-charset passwords). Our leet-transformed dictionary, which contains 4-charset passwords, appears to be highly effective in bypassing password requirements and resulting high-score passwords; see Figure 25b.

5.7 Google Checker Hypothesis

Based on our test results, it is difficult to model Google's server-side checker. We explained discrepancies in Section 4.2.2 by providing strange examples, where for a given charset structure, a leading digit not only has an importance for the charset diversity, but the value of the digit itself also appears to be significant. We showed that *testtest0* is strong, while *testtest1* is only fair. We also noticed that strength scores fluctuate in time for no apparent reason. We speculate that Google's scoring algorithm might be based on a dynamic mechanism (cf. password popularity [55]), which may explain the change of strength for some passwords with time, and why the particular value of digits is of importance in a password.

5.8 Password Policies

Some password policies are explicitly stated (e.g., Apple and FedEx), and others can be deduced from their algorithms or outputs. However, policies as used for measuring strength remain mostly unexplained to users. Differences in policies are also the primary reason for the heterogeneity in strength outcomes. Some checkers are very stringent, and assign scores only when a given password covers at least 3 charsets (e.g., FedEx), or disallow the password to be submitted for blacklist check unless it covers the required charsets and other possible requirements (e.g., Apple, PayPal), while other checkers apparently promote the use of single-charset passphrases. Policies also widely vary even between similar web services. Interestingly, email providers such as Google and Yahoo! that deal with a lot of personal information, apply a more lenient policy than FedEx, which arguably hosts far less sensitive one.

Chapter 6

Discussion

In this section, we discuss some challenges in designing a reliable meter, share some insights as gained from our analysis, and provide few suggestions on improving current meters.

6.1 Implications of Online vs. Offline Attacks

The strength of a password should represent the amount of effort an adversary must employ to break the password (see e.g., [16]). When mapping entropy or guessability to a strength score, in effect, we estimate the time needed by an attacker for guessing a particular password. However, such an estimate will be very different depending on the type of password guessing attack considered, i.e., online vs. offline (see e.g., [32]). Few passwords per second may be guessed in an online attack (before being rate-limited and/or facing CAPTCHA challenges), while billions per second may be tested in an offline attack. While it may be reasonable for web services to consider only online attacks, history proved us time and again that (hashed) password databases leak more frequently than we may anticipate (cf. [45]), and millions of hashed passwords may be subjected to offline cracking. Thus, the large difference in efficiency between online and offline attacks complicates assigning password strengths. Web services should at least explain to users what the assigned strength of a given password may mean.

6.2 Password Leaks

Real-world password leaks[22] complicate the design of a reliable meter. A strong leaked password used by a significant proportion of users, will most likely be integrated into a general attack dictionary, and thus should be disallowed, or at least be assigned a lower score. For checker designers, tracking and incorporating all leaked password databases may be infeasible in practice. Users also may be confused to discover that their *perfect* password not even being allowed after a while. Thus, considering password leaks, and the increasing number of users creating new passwords, adaptive and time-variant checkers (e.g., [16, 31]) may provide more reliable strength outcome.

6.3 Passphrases

Passphrases can offer decent entropy while being easier to memorize for users [40, 57], as long as they do not follow simple grammatical structures [54]. However, the majority of web-based checkers (except Dropbox, Twitter, and Intel to some extent), would rank passphrases at the lower-end in the strength scale. Checkers basing their score on charset complexity (Apple, Drupal, FedEx, Microsoft v1, PayPal, Tencent QQ, Yahoo!, 12306.cn) always assign low scores to passphrases. Other checkers (Apple, Microsoft v2 and v3) do not grant their best scores unless more charsets are used. Finally, we found that application-based checkers are generally more friendly to passphrases.

6.4 Relative Performance of our Dictionaries

As discussed in Section 3.3.3 and 3.3.4, the mangling rules and leet transformations employed in our tests are not carefully optimized or targeted as they would be by a determined attacker (cf. [2, 3]). A better designed targeted dictionary may prove

[22]For an example collection of leaked password databases, see: `http://ThePasswordProject.com/leaked_password_lists_and_dictionaries`

significantly more effective against the meters, as evident from our test against FedEx (see Section 4.1.3). We believe that even if our base/mangled dictionaries are disallowed or assigned reduced scores, users will try to creatively bypass such restrictions with other simple patterns (cf. [38, 66]). In this regard, our analysis is an underestimate as for how real user-chosen passwords are evaluated by meters.

6.5 Directions for Better Checkers

Several factors may influence the design of an ideal password checker, including: inherent patterns in user choice, dictionaries used in cracking tools, exposure of large password databases, and user-adaptation against password policies. Designing such a checker would apparently require significant efforts. In terms of password creation, one may wonder what choices would remain for a regular user, if a checker prevents most logical sequences and common/leaked passwords.

Checkers must analyze the structure of given passwords to uncover common patterns, and thereby, more accurately estimate resistance against cracking. Simple checkers that rely solely on charset complexity with stringent length requirements, may mislead users about their password strength. Full-charset random passwords are still the best way to satisfy all the checkers, but that is a non-solution for most users due to obvious usability/memorability issues. On the positive side, as evident from our analysis, Dropbox's rather simple checker is quite effective in analyzing passwords, and is possibly a step towards the right direction (KeePass also adopts a similar algorithm).

If popular web services were to change their password-strength meter to a commonly-shared algorithm, part of the confusion would be addressed. At least, new web services that wish to implement a meter, should not start the development of yet another algorithm, but rather consider using or extending *zxcvbn* [69] (under a very short and permissive license). Meters embedded in software such as password managers should also consider KeePass' open-source implementation (under GNU GPLv2).

As discussed, current password meters must address several non-trivial challenges, including finding patterns, coping with popular local cultural references, and dealing with leaked passwords. Considering these challenges, with no proven academic solution to follow, it is possibly too demanding to expect a correct answer to: is a given password "perfect"? We believe password meters can simplify such challenges by limiting their primary goal only to detecting weak passwords, instead of trying to distinguish a good, very good, or great password (this approach has been taken by Intel, although with a poor implementation). Meters can easily improve their detection of weak passwords by leveraging known cracking techniques and common password dictionaries. In contrast, labeling user-chosen passwords as perfect may often lead to errors (seemingly random passwords, e.g., *qeadzcwrsfxv1331* or *Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn1* may not be as strong as they may appear [2, 3]).

Chapter 7

Conclusion

Passwords are not going to disappear anytime soon and users are likely to continue to choose weak ones because of many factors, including the lack of motivation/feasibility to choose stronger passwords (cf. [30]). Users may be forced to choose stronger passwords by imposing stringent policies, at the risk of user resentment. An apparent better approach is to provide appropriate feedback to users on the quality of their chosen passwords, with the hope that such feedback will influence choosing a better password, *willingly*. For this approach, password-strength meters play a key role in providing feedback and should do so in a consistent manner to avoid possible user confusion. In our large-scale empirical analysis, it is evident that the commonly-used meters are highly inconsistent, fail to provide coherent feedback on user choices, and sometimes provide strength measurements that are blatantly misleading.

We highlighted several weaknesses in currently deployed meters, some of which are rather difficult to address (e.g., how to deal with leaked passwords). Designing an ideal meter may require more time and effort; the number of academic proposals in this area is also quite limited. However, most meters in our study, which includes meters from several high-profile web services (e.g., Google, Yahoo!, PayPal) are quite simplistic in nature and apparently designed in an ad-hoc manner, and bear no indication of any serious efforts from these service providers. At least, the current meters should avoid providing misleading strength outcomes, especially for weak passwords. We hope

that our results may influence popular web services to rethink their meter design, and encourage industry and academic researchers to join forces to make these meters an effective tool against weak passwords.

Bibliography

[1] A. Adams and M. A. Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.

[2] ArsTechnica.com. Anatomy of a hack: How crackers ransack passwords like "qeadzcwrsfxv1331". News article (May 27, 2013). http://arstechnica.com/security/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/.

[3] ArsTechnica.com. How the Bible and YouTube are fueling the next frontier of password cracking. News article (Oct. 8, 2013). http://arstechnica.com/security/2013/page/4/.

[4] F. Bergadano, B. Crispo, and G. Ruffo. High dictionary compression for proactive password checking. *ACM Transactions on Information and System Security*, 1(1):3–25, Nov. 1998.

[5] M. Bishop. Proactive password checking. In *4th Workshop on Computer Security Incident Handling*, Aug. 1992.

[6] M. Bishop and D. Klein. Improving system security via proactive password checking. *Computers & Security*, 14(3):233–249, May/June 1995.

[7] Bloomberg.com. Yahoo says it's investigating security breach. News article (July 12, 2012). http://www.bloomberg.com/news/2012-07-12/yahoo-spokeswoman-says-company-investigating-security-breach.html.

[8] C. Blundo, P. D'Arco, A. De Santis, and C. Galdi. A novel approach to proactive password checking. *Infrastructure Security*, 2437:30–39, 2002.

[9] C. Blundo, P. D'Arco, A. De Santis, and C. Galdi. Hyppocrates: a new proactive password checker. *The Journal of Systems and Software*, 71(1-2):163–175, 2004.

[10] J. Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2012.

[11] J. Bonneau and S. Preibusch. The password thicket: technical and market failures in human authentication on the web. In *Workshop on the Economics of Information Security (WEIS'10)*, Harvard University, USA, June 2010.

[12] J. Bonneau and R. Xu. Character encoding issues for web passwords. In *Web 2.0 Security & Privacy (W2SP'12)*, San Francisco, CA, USA, May 2012.

[13] M. Burnett. *Perfect Password: Selection, Protection, Authentication*, pages 109–112. Syngress, 2005. The password list is available at: `http://boingboing.net/2009/01/02/top-500-worst-passwo.html`.

[14] M. Burnett. 10,000 top passwords, June 2011. `https://xato.net/passwords/more-top-worst-passwords/`.

[15] W. E. Burr, D. F. Dodson, and W. T. Polk. Electronic authentication guidelines. NIST Special Publication 800-63, Apr. 2006. `http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf`.

[16] C. Castelluccia, M. Dürmuth, and D. Perito. Adaptive password-strength meters from Markov models. In *Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA, USA, Feb. 2012.

[17] A. Ciaramella, P. D'Arco, A. De Santis, C. Galdi, and R. Tagliaferri. A novel approach to proactive password checking. *IEEE Transactions on Dependable and Secure Computing*, 3(4):327–339, 2006.

[18] CNET.com. Syrian Electronic Army hacks Forbes, steals user data. News article (Feb. 14, 2014). `http://news.cnet.com/8301-1009_3-57618945-83/syrian-electronic-army-hacks-forbes-steals-user-data/`.

[19] A. Das, J. Bonneau, M. Caesar, N. Borisov, and X. Wang. The tangled web of password reuse. In *Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, USA, Feb. 2014.

[20] C. Davies and R. Ganesan. BApasswd: A new proactive password checker. In *National Computer Security Conference*, Baltimore, MA, USA, Sept. 1993.

[21] M. Dell'Amico, P. Michiardi, and Y. Roudier. Password strength: An empirical analysis. In *IEEE INFOCOM'10*, San Diego, CA, USA, Mar. 2010.

[22] M. Dürmuth, A. Chaabane, D. Perito, and C. Castelluccia. When privacy meets security: Leveraging personal information for password cracking, Apr. 2013. `http://arxiv.org/abs/1304.6584`.

[23] S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley. Does my password go up to eleven? The impact of password meters on password selection. In *ACM Conference on Human Factors in Computing Systems (CHI'13)*, Paris, France, 2013.

[24] D. Florêncio and C. Herley. A large-scale study of web password habits. In *International World Wide Web Conference (WWW'07)*, Banff, Alberta, Canada, May 2007.

[25] D. Florêncio, C. Herley, and B. Coskun. Do strong web passwords accomplish anything? In *USENIX Workshop on Hot Topics in Security (HotSec'07)*, San Diego, CA, USA, Feb. 2007.

[26] S. Furnell. Assessing password guidance and enforcement on leading websites. *Computer Fraud & Security*, 2011(12):10–18, Dec. 2011.

[27] S. T. Haque, M. Wright, and S. Scielzo. A study of user password strategy for multiple accounts. In *CODASPY'13*, San Antonio, TX, USA, Feb. 2013.

[28] M. Harry. *The Computer Underground*. Loompanics Unlimited, Port Townsend, WA, USA, 1985.

[29] N. V. Heijningen. A state-of-the-art password strength analysis demonstrator. Master's thesis, Rotterdam University, June 2013.

[30] C. Herley and P. Van Oorschot. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, 10(1):28–36, 2012.

[31] S. Houshmand and S. Aggarwal. Building better passwords using probabilistic techniques. In *Annual Computer Security Applications Conference (ACSAC'12)*, Orlando, FL, USA, Dec. 2012.

[32] P. Inglesant and M. A. Sasse. The true cost of unusable password policies: Password use in the wild. In *ACM Conference on Human Factors in Computing Systems (CHI'10)*, Atlanta, GA, USA, Apr. 2010.

[33] B. Ives, K. R. Walsh, and H. Schneider. The domino effect of password reuse. *Communications of the ACM*, 47(4):75–78, 2004.

[34] K. Jamuna, S. Karpagavalli, and M. Vijaya. A novel approach for password strength analysis through support vector machine. *International Journal of Recent Trends in Engineering*, 2(1):79–82, 2009.

[35] G. J. Johnson. A distinctiveness model of serial learning. *Psychological Review*, 98(2):204–217, Apr. 1991.

[36] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2012.

[37] D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *USENIX Security Workshop*, Aug. 1990.

[38] S. Komanduri, R. Shay, P. G. Kelley, M. L. Mazurek, L. Bauer, N. Christin, L. F. Cranor, and S. Egelman. Of passwords and people: measuring the effect of password-composition policies. In *ACM Conference on Human Factors in Computing Systems (CHI'11)*, Vancouver, BC, Canada, May 2011.

[39] KrebsOnSecurity.com. Facebook warns users after adobe breach. News article (Nov. 11, 2013). `http://krebsonsecurity.com/2013/11/facebook-warns-users-after-adobe-breach/`.

[40] C. Kuo, S. Romanosky, and L. F. Cranor. Human selection of mnemonic phrase-based passwords. In *Symposium On Usable Privacy and Security (SOUPS'06)*, Pittsburgh, PA, USA, July 2006.

[41] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur. Measuring password guessability for an entire university. In *ACM Conference on Computer and Communications Security (CCS'13)*, Berlin, Germany, Nov. 2013.

[42] J. McCune, A. Perrig, and M. Reiter. Safe passage for passwords and other sensitive data. In *Network and Distributed System Security Symposium (NDSS'09)*, San Diego, CA, USA, Feb. 2009.

[43] R. Merchant. Personal data at risk on 63% of UK top 100 ecommerce sites – Dashlane Q1 2014 personal data security roundup. Technical report, Dashlane, Mar. 2014. `https://www.dashlane.com/blog/security/personal-data-security-roundup-uk-edition/`.

[44] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63(2):81–97, Mar. 1956.

[45] D. Mirante and J. Cappos. Understanding password database compromises. Technical Report TR-CSE-2013-02, Polytechnic Institute of NYU, Sept. 2013. `http://isis.poly.edu/~jcappos/papers/tr-cse-2013-02.pdf`.

[46] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, Nov. 1979.

[47] NakedSecurity.Sophos.com. YouPorn passwords available for download, thousands of users exposed. News article (Feb. 22, 2012). `http://nakedsecurity.sophos.com/2012/02/22/youporn-password-download/`.

[48] A. Narayanan and V. Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM Conference on Computer and Communications Security (CCS'05)*, Alexandria, VA, USA, Nov. 2005.

[49] NYTimes.com. Hackers disrupt sites run by gawker media. News article (Dec. 12, 2010). http://www.nytimes.com/2010/12/13/business/media/13gawker.html.

[50] OpenWall.com. John the Ripper password cracker. http://www.openwall.com/john.

[51] Oxid.it. Cain & Abel. http://www.oxid.it/cain.html.

[52] phpBB.com. Downtime and server compromise. News article (Feb. 2, 2009). http://area51.phpbb.com/phpBB/viewtopic.php?t=29973.

[53] R. W. Proctor, M.-C. Lien, K.-P. L. Vu, E. E. Schultz, and G. Salvendy. Improving computer security for authentication of users: influence of proactive password restrictions. *Behavior Research Methods, Instruments, & Computers*, 34(2):163–169, 2002.

[54] A. Rao, B. Jha, and G. Kini. Effect of grammar on security of long passwords. In *ACM Conference on Data and Application Security and Privacy (CODASPY'13)*, San Antonio, TX, USA, Feb. 2013.

[55] S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *USENIX Workshop on Hot Topics in Security (HotSec'10)*, Washington, DC, USA, Aug. 2010.

[56] B. Schneier. MySpace passwords aren't so dumb. News article (Dec. 14, 2006). http://www.wired.com/politics/security/commentary/securitymatters/2006/12/72300.

[57] R. Shay, P. G. Kelley, S. Komanduri, M. L. Mazurek, B. Ur, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. Correct horse battery staple: Exploring the usability of system-assigned passphrases. In *Symposium On Usable Privacy and Security (SOUPS'12)*, Washington, DC, USA, July 2012.

[58] R. Shay, S. Komanduri, P. G. Kelley, P. G. Leon, M. L. Mazurek, L. Bauer, N. Christin, and L. F. Cranor. Encountering stronger password requirements: user attitudes and behaviors. In *Symposium On Usable Privacy and Security (SOUPS'10)*, Redmond, WA, USA, July 2010.

[59] E. H. Spafford. OPUS: Preventing weak password choices. *Computers & Security*, 11(3):273–278, May 1992.

[60] TechCrunch.com. RockYou hack: From bad to worse. News article (Dec. 14, 2009). http://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/.

[61] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? The effect of strength meters on password creation. In *USENIX Security Symposium*, Bellevue, WA, USA, Aug. 2012.

[62] B. Ur, S. Komanduri, R. Shay, S. Matsumoto, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, M. L. Mazurek, and T. Vidas. Poster: The art of password creation. In *IEEE Symposium on Security and Privacy*, San Francisco, California, USA, May 2013.

[63] R. Veras, C. Collins, and J. Thorpe. On the semantic patterns of passwords and their security impact. In *Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, USA, Feb. 2014.

[64] S. Vidyaraman, M. Chandrasekaran, and S. Upadhyaya. Position: the user is the enemy. In *New Security Paradigms Workshop (NSPW'07)*, New Hampshire, USA, Sept. 2007.

[65] W3Techs.com. Market share trends for content management systems for websites. Online report. `http://w3techs.com/technologies/history_overview/content_management`.

[66] M. Weir. *Using probabilistic techniques to aid in password cracking attacks*. PhD thesis, Florida State University, Mar. 2010.

[67] M. Weir, S. Aggarwal, M. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *ACM Conference on Computer and Communications Security (CCS'10)*, Chicago, IL, USA, Oct. 2010.

[68] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009.

[69] D. Wheeler. zxcvbn: realistic password strength estimation. Dropbox blog article (Apr. 10, 2012). `https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/`.

[70] World Wide Web Consortium (W3C). Cross-Origin Resource Sharing. W3C Candidate Recommendation (Jan. 29, 2013). `http://www.w3.org/TR/cors/`.

[71] J. J. Yan. A note on proactive password checking. In *New Security Paradigms Workshop (NSPW'01)*, Cloudcroft, NM, USA, Sept. 2001.

[72] J. J. Yan, A. F. Blackwell, R. J. Anderson, and A. Grant. The memorability and security of passwords: some empirical results. Technical Report 500, University of Cambridge, Sept. 2000. `http://www.fraw.org.uk/files/hacktivism/yan_2000.pdf`.

[73] J. J. Yan, A. F. Blackwell, R. J. Anderson, and A. Grant. Password memorability and security: Empirical results. *IEEE Security & Privacy*, 2(5):25–31, Sept/Oct. 2004.

[74] ZDNet.com. 6.46 million LinkedIn passwords leaked online. News article (June 6, 2012). http://www.zdnet.com/blog/btl/6-46-million-linkedin-passwords-leaked-online/79290.

[75] M. Zviran and W. J. Haga. Cognitive passwords: The key to easy access control. *Computers & Security*, 9(8):723–736, 1990.