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

XAVIER DE CARNÉ DE CARNAVALET and MOHAMMAD MANNAN, Concordia University

Passwords are ubiquitous in our daily digital lives. 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, attackers have developed insights into cracking/guessing passwords both offline and online. In many cases, users are forced to choose stronger passwords to comply with password policies; such policies are known to alienate users and do not significantly improve password quality. Another solution is to put in place proactive password-strength meters/checkers to give feedback 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. More recently, these meters are also being built into popular password managers, which protect several user secrets including passwords. 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 source-available 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.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Authentication*; H.1.2 [**Models and Principles**]: User/Machine Systems—*Human factors*

General Terms: Security, Human Factors

Additional Key Words and Phrases: Password strength, strength meter, password manager

ACM Reference Format:
Xavier de Carné de Carnavalet and Mohammad Mannan, 2015. A Large-Scale Evaluation of High-Impact Password Strength Meters. *ACM Trans. Info. Syst. Sec.* V, N, Article A (January 2015), 32 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Proactive password checkers have been around for decades; for some earlier references, see e.g., Morris and Thompson [1979], Spafford [1992], and Bishop and Klein [1995]. Recently, password checkers are being deployed as password-strength meters on many

Version: Feb. 27, 2015. This article is the extension of an NDSS 2014 publication [Carnavalet and Mannan 2014]; see also the first author's Master's thesis [Carnavalet 2014]. Authors' address: Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Canada; emails: x_decarn@ciise.concordia.ca and m.mannan@concordia.ca.

DOI: http://dx.doi.org/10.1145/0000000.0000000

websites to encourage users to choose stronger 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 paper.

The presence of a password meter during password change for an important account, and for password creation of an allegedly important account (although in a limited study), has been shown to lead users towards more secure passwords [Ur et al. 2012; Egelman et al. 2013]. However, strengths and weaknesses of widely-deployed password meters have been scarcely studied so far. Furnell [2011] 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., [Castelluccia et al. 2012; Weir et al. 2010]. 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 [Burr et al. 2006], and its weaknesses [Weir et al. 2010]. Better password checkers have been proposed (e.g., [Schechter et al. 2010; Castelluccia et al. 2012; Houshmand and Aggarwal 2012; Veras et al. 2014]), but 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 apparently guiding the password choice of millions of users.

We systematically characterize the password meters of 14 prominent web service providers, ranging from financial, email, cloud storage to messaging services that are ranked in Alexa's top 100 (e.g., Google, ranked 1), or are related to a high ranked service (e.g., Skype.com is ranked 186, which is a major VoIP service from Microsoft.com, ranked 43). We also analyze meters from four leading password management software tools, which we selected from various password managers rankings [LifeHacker.com 2008; PCMag.com 2014]. Our target meters include: Google, Tencent QQ, Yahoo!, Twitter, eBay, Yandex, Apple, PayPal, Microsoft (three versions), Dropbox, Skype, FedEx, China railway customer service center, Drupal, LastPass, KeePass, 1Password (AgileBits, three versions) and RoboForm (Siber Systems). We extract and analyze JavaScript code (partly obfuscated) for 12 services and browser extensions involving local/in-browser processing. We further analyze the C# source code for a password manager and reverse-engineer, to some extent, the six services involving server-side processing that appear as a black-box along with the two closed-source password managers. Unlike web-based password meters, the meters in password managers guide the creation of a master password that further encrypts several other secrets and passwords. Thus, it is particularly important to assess such meters. For Drupal, even though the website itself is not ranked high in Alexa, its content management system is widely used by many websites from various industries (see: DrupalShowcase.com). Finally, we include FedEx and China railway for diversity. During our experiments, we notified 11 web services and received feedback from seven of them. The company behind a password manager was also indirectly notified (via Twitter). After 18 months of our initial experiments and almost a year after notifying the companies, we reevaluate the meters and report the changes in their algorithms, if any.

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 nine and a half million passwords from 13 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.

The password dictionaries we consider are mostly composed of weak passwords by essence. Although different websites may require various levels of security and enforce heterogeneous password policies, the evaluation of a password's strength is expected to be consistent and clear to the user. We assume a user should not receive a "strong" feedback on an easy-to-crack password at a non-sensitive website, if it is not accompanied with a properly identified context. Such a user could otherwise be tempted to reuse this password on a more sensitive system that does not employ a strength meter. Thus, we expect meters to provide consistent output on a given password, either by providing similar feedback, or by indicating the "portability" of the given feedback.

Florêncio et al. [2007] suggest that strong passwords do not serve much purpose in face of other less-frequently considered types of attack, such as phishing, keylogging, shoulder surfing, leaked browser-saved passwords and bulk guessing attack. Florêncio et al. [2014] also put into question the requirement for strong passwords since increasing the strength of passwords beyond a certain point results in limited additional security benefit. While we find this observation valid, we note however that database leaks of hashed passwords are still a problem (as apparent from the LinkedIn leak [ZDNet.com 2012]; most of these hashed passwords have been cracked). Online password guessing can also be problematic, if proper countermeasures are not enforced; cf. the recent iCloud leak [TheNextWeb.com 2014; CSO Online 2014] that allegedly occurred via an unrestricted online password guessing attack.

To measure the quality of a given password, checkers usually employ the following methods: enforce strong requirements, mostly regarding the length and character-set complexity; or try to detect weak patterns such as common words, repetitions and easy keyboard sequences. Some web service checkers are implemented at client-end only, some at server-end only, and the rest are hybrid, i.e., include measurements both at the server- and client-ends. We also analyze strengths and limitations of these approaches.

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 secure by Yandex), *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! and Yandex) that one may wonder what purpose they may serve. Considering some of these meters are deployed by highly popular websites and password managers, we anticipate inconsistencies in these meters would confuse users, and eventually make the meters a far less effective tool.

Main contributions of this work are summarized as follows.

(1) METER CHARACTERIZATION. We systematically characterize a total of 22 password strength meters as deployed at popular web services and leading password managers, which are used by hundreds of millions of users worldwide. This characterization is particularly important for checkers with a server-side component, which appears as a black-box; no vendors in our study provide any information on their design choices. Even for client-side checkers, no analysis or justification is provided (except Dropbox). After notifying several companies behind the meters, we evaluate changes in their algorithms and review the changes (if any). Our meter characterization can be viewed as a snapshot of password strength ranking trends between May 2013 and Nov. 2014.

(2) REVERSE-ENGINEERING OF METERS IN PASSWORD MANAGER APPLICATIONS. We successfully reverse-engineered the password meter algorithms of two popular closed-source password managers. These managers provide a meter to guide the creation of a master password that protects several other user secrets and passwords. Such meters are thus even more sensitive. To the best of our knowledge, no such analysis on open/closed-source managers exists.

(3) EMPIRICAL EVALUATION OF METERS. For each of the 22 meters, we used nearly nine million unique passwords from several password dictionaries (a total of at least 200 million test cases, of which more than 52 million are tested against online services). This is the largest such study on password meters to the best of our knowledge. All checkers' response profiles are also compared against each other.

(4) 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 *secure*, *very strong*, *excellent* or even *perfect*; (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.

2. METERS CHARACTERIZATION

2.1. 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 for the main application, and passwords in web forms; sometimes they are also used to display the strengths of all stored passwords. 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; some common requirements and features of the meters are also summarized in Table 2.1.

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).

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 a continuous progress bar without labels.

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, or review the source code for this purpose if available. 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 (details in Section 3).

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 only locally (e.g., Dropbox, Drupal, FedEx, Microsoft, QQ, Twitter, Yahoo! and China railway customer service center); *server-side*: the checker is implemented fully on server-side (e.g., eBay, Google, Skype and Yandex); and *hybrid*: a combination of both (e.g., Apple and PayPal). All our tested password managers operate at user-end (open or closed source). We use different approaches to automate our tests for these varying types of meters; see Section 2.2.

Diversity. None of the 14 web services and 4 password managers we evaluated uses a common meter. Instead, each service or 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* [Wheeler 2012]; Dropbox also provides details of this open-source meter.

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 password length. As discussed in Section 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.

2.2. Test Automation

We tested nearly nine and a half million of passwords against each of the 22 checkers. Below, we discuss how we performed such large-scale automated tests.

2.2.1. Web-Based Client-Side Meters. For client-side checkers, we extract the relevant JavaScript functions from a registration page, and query them to get the strength score for each dictionary password. Outputs are then stored for later analysis. To identify the sometimes obfuscated part of the code, we use the built-in debugger in Google Chrome. In particular, we set breakpoints on DOM changes, i.e., when the password meter's outcome is updated. Such obfuscation may be the result of code minification

Table I. Password requirements and characteristics of the evaluated meters. Strengths underlined in "Strength scale" are the minimum strengths considered as good when comparing meters in Section 5.3. "User info": ø (no user information is used for strength check), and ◐ (some user information is used). None of these meters include all available user info, nor do they involve more than a mere equality check with user-provided fields (e.g., username, email address). "Charset required" represents the requirements to make a password valid under the service's policy. We use L, U, D to represent lowercase, uppercase letters and digits respectively; 2+ to denote "two or more" charsets; and ø to represent no requirements. The "Space allowed" column represents whether a meter allows spaces inside a password (but not accounting for leading of trailing spaces, as these are usually trimmed). The "Enforcement" column represents the minimum strength required by each checker for registration completion: ø (no enforcement); and other labels as defined under "Strength scale".

Type		Service	Strength scale	Length min/max	Charset required	Space allowed	User info	Enforcement
Web-based	Client-side	Dropbox	Very weak, Weak, So-so, Good, Great	6 / 72	ø	✓	◐	ø
		Drupal	Weak, Fair, Good, Strong	6 / 128	ø	✓	◐	ø
		FedEx	Very weak, Weak, Medium, Strong, Very strong	8 / 35	L, U, D	×	ø	Medium
		Microsoft	Weak, Medium, Strong, Best	1 / –	ø	✓	ø	ø
		QQ	Weak, Moderate, Strong	6 / 16	ø	×	ø	ø
		Twitter	Invalid/Too short, Obvious, Not secure enough (NSE), Could be more secure (CMS), Okay, Perfect	6 / >1k	ø	✓	◐	CMS
		Yahoo!	Too short, Weak, Strong, Very strong	6 / 32	ø	✓	◐	Weak
		12306.cn[1]	Dangerous, Average, Secure	7 / 25	L or U, and D	×	ø	ø
	Server-side	eBay	Invalid, Weak, Medium, Strong	6 / 20	2+	✓	◐	ø
		Google	Weak, Fair, Good, Strong	8 / 100	ø	✓	ø	Fair
		Skype	Poor, Medium, Good	6 / 20	2+ or U	×	ø	Medium
		Yandex	Too short, Weak, Secure	6 / 255	ø	×	◐	Weak
	Hyb.	Apple	Weak, Moderate, Strong	8 / 32	L, U, D	×	◐	Medium
		PayPal	Weak, Fair, Strong	8 / 20	2+[2]	×	ø	Fair
App-based		1Password	Terrible, Weak, Fair, Good, Excellent, Fantastic	1 / –	ø	✓	ø	ø
		KeePass	0–65–128 bits	1 / –	ø	✓	ø	ø
		LastPass	0–51–100%	1 / –	ø	✓	◐	ø
		RoboForm	Weak, Good, Excellent	6 / 49	ø	✓	ø	ø

[1] 12306.cn is the domain name for China railway customer service center
[2] PayPal counts uppercase and lowercase letters as a single charset

(e.g., removing comments, extra spaces, and shrinking variable names). Fortunately, strength meters generally involve simple logic, and remain understandable even after such optimization. As some checkers are invoked with key-press events, the use of a debugger also simplified locating the relevant sections. 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 half an hour for a 5-million dictionary against the most complex meter (Dropbox).

2.2.2. Web-Based Server-Side Meters. Server-side checkers directly send the password to a server-side checker 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 (curl.haxx.se) for handling HTTPS requests to the server. The checker's URL is obtained from the JavaScript code and/or a network capture. We use Google Chrome to set breakpoints on AJAX calls to be pointed to the `send`[1] call before its execution. This

[1] http://www.w3.org/TR/XMLHttpRequest/#the-send()-method

enables 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, 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 9 million passwords for each service (the small overlap between dictionaries was not stripped), and we could request up to about 1000 passwords through one connection with Skype, 1500 with eBay, and unlimited with Google; as a result, the number of connections dropped significantly. We also did not parallelize the requests. On average, we tested our dictionaries at a speed of 5 passwords per second against Skype, 8 against Yandex, 10 against eBay, 64 against Google (2.5 against PayPal and 8 against Apple for the server-side part of their checkers), generating a maximum traffic of 5kB/s of upload and 10kB/s of download per web service. To our surprise, our tests were not blocked by the servers (except for eBay that permanently blocked one IP towards the end of our tests).

2.2.3. Web-Based Hybrid Meters. 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 is inherently from another origin, we face restrictions imposed by the same-origin policy [Barth 2011]. To allow client-side cross-origin requests, the cross-origin resource sharing (CORS [World Wide Web Consortium (W3C) 2013]) mechanism has been introduced and is currently implemented in most browsers. To allow our local script as a valid origin, we implemented a simple proxy to insert the required CORS header, `Access-Control-Allow-Origin`, 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. We implemented a simple HTTP server in 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.

2.2.4. Application-Based Meters. 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 a JavaScript browser extension.

Open-source meters. Open-source password managers in our evaluation include a C# application (KeePass) and JavaScript-based browser extensions (LastPass and two 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 13 dictionaries within 15 minutes. Tests against the JavaScript-based checkers are similar to 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. We automate the evaluation of our dictionaries by simulating user input of passwords, and collecting back the results for further analysis. We leverage AutoIt (autoitscript.com/site/autoit/) to automate such tests. AutoIt is a free software interpreter with a custom scripting language designed for automating the Windows GUI. 1Password's 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, RoboForm's meter provides only five possible positions and uses a custom graphical meter, which cannot be processed directly by AutoIt to extract values. To collect and reconstitute the meter's output, we retrieve the color of selected pixels on the screen with AutoIt at five different locations on the meter. Tests are performed at a speed of 90 passwords per second against 1Password and 8.5 against RoboForm.

RoboForm's algorithm is apparently too complex to understand from its output only, hence we rely on the reverse-engineering of its software binary for characterizing it. To do this, 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. Such a method is cumbersome as the variety of operations performed increases the length and complexity of the instructions to analyze. We were successful at attaching Immunity Debugger [Immunity Inc. 2014] to RoboForm's binary 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 score is calculated. Details of this analysis are given in Section 4.4.

2.3. Tested Dictionaries

2.3.1. Overall description. Table II lists the 13 dictionaries we used. Our dictionary sources include: password cracking tools, e.g., John the Ripper (JtR [OpenWall.com 2014]) and Cain & Abel (C&A [Oxid.it 2014]); a list of 500 most commonly used passwords (Top500 [Burnett 2005]); an embedded dictionary in the Conficker worm (Cfkr [Sophos.com 2009]), and leaked databases of plaintext or hashed passwords (RockYou.com (RY5), phpBB.com (phpBB), Yahoo.com (YA) and LinkedIn.com (LinkedIn)). We mostly chose simple and well-known dictionaries (as opposed to more complex ones, see e.g., [ArsTechnica.com 2013]), to evaluate checkers against passwords that are reportedly used by many users. Except for leaked databases, we expected passwords from these non-targeted dictionaries would be mostly rejected (or rated as weak) by the meters. We also derive four additional dictionaries using well-known password mangling rules [OpenWall.com 2014]. As we noticed that our main dictionaries did not specifically consider leet transformations, we built a special leet dictionary using the base dictionaries.

Several of our dictionaries are created from sources between 2005 and 2009 (somewhat outdated by now). Many other password leaks were reported in recent years; however, in most cases, the plaintext passwords are unavailable. We selected two leaked password databases from relatively important accounts in 2012 (YA and LinkedIn). We test these dictionaries almost a year after we notified several web services, which allows us to observe the potential changes in their algorithms. For example, Yahoo! removed the password checker from its registration webpage, and 1Password corrected its algorithm. Apple and eBay also demonstrate strange behaviors.

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. Some leaked passwords also contained UTF-8-encoded

Table II. Dictionaries used against password checkers; +M represents mangled version of a dictionary; the "Leet" dictionary is custom-built by us. "Composition" column gives the three main charsets that compose the dictionaries, with the percentages attached to each. The caret (^) before a range(s) of characters means "everything but" the following range(s), e.g., [^A-Z0-9] means lowercase and symbols; \w means all four charsets.

Dictionary	# words	Max/Avg/Std length	Composition		
			Rank #1	Rank #2	Rank #3
Top500	499	8 / 6.00 / 1.10	[a-z] 91%	[0-9] 7%	[a-z0-9] 2%
Cfkr	181	13 / 6.79 / 1.47	[a-z] 53%	[0-9] 30%	[a-z0-9] 16%
JtR	3,545	13 / 6.22 / 1.40	[a-z] 83%	[a-z0-9] 8%	[a-zA-Z] 4%
C&A	306,706	24 / 9.27 / 2.77	[a-z] 99.84%	[^A-Z0-9] 0.09%	[a-z0-9] 0.05%
RY5	562,987	49 / 7.41 / 1.64	[a-z] 40%	[a-z0-9] 36%	[0-9] 17%
phpBB	184,389	32 / 7.54 / 1.75	[a-z] 41%	[a-z0-9] 36%	[0-9] 11%
YA	342,514	30 / 8.49 / 1.88	[a-z0-9] 57%	[a-z] 25%	[a-zA-Z0-9] 6%
LinkedIn	5,092,552	54 / 8.77 / 1.95	[a-z0-9] 45%	[a-z] 21%	[a-zA-Z0-9] 15%
Top500+M	22,520	12 / 7.18 / 1.47	[a-z0-9] 38%	[a-zA-Z0-9] 20%	[a-zA-Z] 15%
Cfkr+M	4,696	16 / 7.88 / 1.78	[a-z0-9] 39%	[a-zA-Z0-9] 21%	[a-zA-Z] 15%
JtR+M	145,820	16 / 7.30 / 1.66	[a-z0-9] 39%	[a-zA-Z0-9] 20%	[a-zA-Z] 15%
RY5+M	2,173,963	39 / 8.23 / 1.98	[a-z] 24%	[a-z0-9] 19%	[a-zA-Z0-9] 18%
Leet	648,116	20 / 9.09 / 1.81	[\w] 78%	[a-zA-Z0-9] 19%	[^0-9] 4%

words that were generally not handled properly by the checkers (cf. [Bonneau and Xu 2012]). 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 default dictionaries from cracking tools, among the leaked passwords we used; see Table 5.1. As a side note, many passwords in the source dictionaries are related to insults, love and sex (cf. [Veras et al. 2014]). We avoid mentioning such words as example passwords.

2.3.2. Sources of tested dictionaries.

Top500. This dictionary was released in 2005 as the "Top 500 Worst Passwords of All Time" [Burnett 2005], and later revised as Top 10000 [Burnett 2011] 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.

Cfkr. The dictionary embedded in the 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; examples include: *computer*, *123123*, and *mypassword*.

JtR. John the Ripper [OpenWall.com 2014] 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*.

TCJ. We combine Top500, Cfkr and JtR (without duplicates) as TCJ. Since these dictionaries share similar characteristics, meters often output similar results for them. Hence, we sometimes refer them as TCJ for simplicity.

C&A. A 306,706-word dictionary, primarily consisting of long lowercase words, e.g., *constantness*, as comes with the password cracking tool, Cain & Abel [Oxid.it 2014].

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.

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.

YA. Yahoo! was breached in July 2012 by an SQL injection, leaking cleartext passwords of about 450,000 users. Removing duplicates yields 342,514 unique passwords.

LinkedIn. In some cases, especially for accounts deemed unimportant, users may knowingly choose not-so-good passwords (see e.g., [Egelman et al. 2013]). So far, we have not seen any leaked password databases for very important services, such as online banking. In this context, the LinkedIn dictionary is a good example of relatively recent real-user passwords used for a web service strongly linked to a user's real identity, and possibly considered *important* by many users. LinkedIn lacks a password policy (except a 6-character length check) and does not incorporate a meter. Hence, this dictionary has the potential to represent what users are left with when they try to choose a password for an important account on their own. Also, the passwords released after the June 2012 breach were hashed (using SHA1 without salt) and independently cracked (partially), demonstrating the effective guessability of such passwords by attackers in practice. The cracked version we rely on was found on adeptus-mechanicus.com, which contains about 5.1 million passwords from the 6.46 million hashes. We tested the dictionary against the hashes acquired from another source and confirmed the match.

2.3.3. 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 [Castelluccia et al. 2012; Komanduri et al. 2011; Burr et al. 2006]; for details on this wide-spread behavior, see e.g., Weir [2010]. 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: lowercase passwords that are not; capitalize; pluralize; suffix with "1"; combine (a) and (d); duplicate short words (6 characters or less); reverse the word; prefix with "1"; uppercase alphanumerical passwords; and 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 [2010]). 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.

2.3.4. 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 [Schechter et al. 2010]. Leet transformations are not covered in our main dictionaries, apart from few exceptions; thus, we built our own leet transformed dictionary to test their effect.

Our Leet dictionary is based on the passwords from Top500, Cfkr, JtR, C&A, phpBB, the full RockYou dictionary, the Top10000 dictionary, and a 37,141-word version of the leaked MySpace password dictionary,[2] 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, 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., [Weir 2010; Heijningen 2013]). 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: `\/`). 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 5.1) 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 [Heijningen 2013], 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.

3. EMPIRICAL EVALUATION OF WEB-BASED METERS

In our previous work [Carnavalet and Mannan 2014], we focused on large services and evaluated the password meters of Apple, Dropbox, Drupal, FedEx, Google, eBay, Microsoft, PayPal, Skype, Twitter and Yahoo!. In this work, we extend our scope to two major Chinese web-services: Tencent QQ and the China railway customer service center (12306.cn), and the Russian-based email provider Yandex Mail. Tests were performed in Feb/Mar. 2014 for QQ and 12306.cn, and Nov. 2014 for Yandex. We first summarize findings from our previous tests (performed in June/July 2013). Then, we present the general behavior and characterize the response of these meters against our test dictionaries, analyze their algorithm and discuss their strengths and weaknesses.

3.1. Highlights of previous findings

Interesting meters from our previous work include Dropbox, FedEx and Google. The first is the most advanced client-side checker that we evaluated, the second appears simple and effective until we show its severe limitations, and the last one is a black-box for which we are unable to provide a definitive explanation.

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

Dropbox. Dropbox has developed a client-side password strength checker called zxcvbn [Wheeler 2012], and open-sourced it to encourage others to use and improve the checker. WordPress uses zxcvbn as its default password meter as of version 3.7 (October 2013) and Kaspersky Labs reuses it to provide an independent checker. 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 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.

Additionally, parts of the password are checked against various dictionaries of common passwords and common English words and names, summing up to 85,088 words. 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. 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.

Compared to other meters, zxcvbn yields more accurate strength evaluations. However, its main limitation, which applies to other meters in general, is the limited size of the embedded dictionary. Non-dictionary words are considered random strings that can be found only by an exhaustive search (brute-force attack), which would take months or years to finish. However, it is unlikely for an attacker to perform such a long exhaustive search. Instead, an attacker may use better dictionaries and additional mangling rules, which is likely to decrease the time-to-crack.

FedEx. This web service is related to one of the few non-IT companies in the scope of our evaluation. The client-side algorithm run by FedEx takes into account leet transformations and embeds a small common passwords dictionary of 180 effective words. Also, a stringent requirement limits a password's strength to very weak, until the password is 8 characters long and includes lowercase, uppercase letters and digits with no three identical consecutive characters. These rules effectively prevent nearly all of our dictionaries from reaching a better score, a desirable outcome since our passwords are arguably weak. However, this meter wrongly assigns weak scores to decent passwords only because they fail one of the requirements; e.g., *Vm*H=Cj%u(YXDcQ* is considered as very weak because it lacks digits. As legitimate users will face such difficulties, they are likely to follow simple rules to modify their password to make it compliant. We created a targeted dictionary for FedEx by combining the basic dictionaries of Top500, JtR and Cfkr. We applied slightly more refined mangling rules consistent with known user behaviors [Weir 2010; Heijningen 2013], 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, FedEx checker is particularly prone to qualify easy-to-crack mangled passwords as of decent strength. An attacker can integrate the effect of the password requirements and start with the weakest accepted passwords.

Google. Google accounts are used to access Google services, in particular Gmail, Drive, Calendar, etc. Account creation is helped by a meter that is difficult to reverse-engineer

for two main reasons: (a) passwords are mostly ranked either as too short or strong; and (b) the meter's inconsistent output for the same password at different times. Consider the following examples: *testtest* is weak and *testtest0* is strong, but *testtest1* is fair, *testtest2* is good and *testtest3* is strong again; such a variety of scores for so minor changes is difficult to comprehend. Other examples show significant variations depending on the position of an uppercased letter in a simple password. If this is the result of advanced checks, then many commonly used patterns would be penalized. However, Google's checker is ranked second and fifth in terms of yielding "strong" passwords from our base and mangled dictionaries, respectively; see Section 5.3.

Regarding time-varying strengths, difference remains limited to one strength level (better or worse). For example, in a two-week interval we found that *overkill* went from weak to fair, *canadacanada* from fair to good, and *anythings* from good to strong, while *startrek4* went from strong to good, and *baseball!* from fair to weak. Weeks or months later, scores can return to their past values as we observed. 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, it is fairly easy to change a weak password to make it strong, e.g., by making one letter uppercase and/or adding a leading digit or symbol. For example *database* 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.

3.2. China railway customer service center

China railway provides a ticket reservation system (www.12306.cn). It is the government's official website for train tickets and generally visited by millions of customers.

Algorithm. A password is considered 'dangerous' (translated from the Chinese word found on the website) for the following cases: its length is less than 7 characters; composed only of letters, digits, or, surprisingly, an underscore character. Beyond these cases, a password is labeled as average. It is labeled as secure if it contains mixed-case letters, digits and at least an underscore character.

Weaknesses. 12306.cn is sensitive only to password composition after a minimal length of 6 characters. It strongly promotes the use of the underscore character, which is the only special character considered. Consequently, a 20-character random password (e.g., *SFh*#6^Z44CwhKB73@x3*) covering all charsets but no underscores is rated only as average. As such, the meter appears extremely stringent to users wishing to reach a secure score until they find the magic character; however, users are advised to include underscores in their password (as stated right next to the password field). In this case, passwords may often contain underscores.

3.3. Tencent QQ

QQ is an instant messaging service developed by the Chinese company Tencent, and mostly targets Chinese-speaking countries. Its user-base is reportedly at least 820-million strong, as of Nov. 2014.[3]

Algorithm. 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 labels. 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. 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 to a strength

[3]http://www.chinainternetwatch.com/10711/tencent-q3-2014/

label as follows: the use of 1 charset is weak, 2 is moderate, 3 and 4 are strong. Note that QQ does not constantly evaluate a password as it is typed by the user, but ranks it when the user switches to another field of the form.

Weaknesses and interesting features. Similar to 12306.cn, this checker is sensitive only to the number of charsets after crossing the minimum length threshold of 8 characters. No further checks are done such as searching for repetitive patterns, weak or common words, nor are any reward given to the length beyond six characters. This leads to a low ranking of passphrases (e.g., a password such as *correcthorsebatterystaple* is weak) and inconsistencies such as *Password1* being rated as strong.

During code review, we noticed two interesting features on the registration page. First, the total time spent on 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 when the form is submitted. Second, the password is encrypted using a 1024-bit RSA key prior to being sent to the server (the form is submitted via HTTP, unlike HTTPS as in most sites).

3.4. Yandex

Yandex is a popular search engine in Russia, which also provides email and storage services. Both the Russian and English versions of the website show a meter in its registration page.

Algorithm. Yandex comprises a graphical bar that is filled up to a percentage calculated as the min value between $100 \cdot \ln(length + 1)/\ln(255)$ and 100, with $length$ being the password length. In practice, it translates to a bar that fills up at a logarithmic speed with respect to the password length, i.e., additional characters for short passwords matter more than the ones for a longer password. The bar is totally filled when the password reaches 254 characters. The password is also sent to the server as typed, starting from the first input character. The server returns an error message if illegal characters are included in the password (e.g., not all special characters are allowed). Otherwise, the server returns a score that is either secure or weak (blocking as an error, or simply a non-blocking warning). A simple check is performed against the username to prevent the password to be the same.

The algorithm for the server-side part of this meter is difficult to infer. It appears that there are checks for sequences of numbers, since passwords such as *1234567890* and *73221987* (but not *73221986*) are ranked as weak. Also, there is a blacklist check that forbids simple words, including some variations of them. For example, *cnffjbeq* is secure while *password* is weak, although both share the same length and character frequencies, and the former is a simple shift (ROT13) of the later. Both *Robert1* and *Monopoly* are weak, which are found as *robert* and *monopoly* respectively in the Top 10000 [Burnett 2011], showing that the algorithm is aware of certain transformations. However, even though *password* is weak, *Password* and *password1* are secure, which contradicts the hypothesis that the Top 10000 is used as a blacklist. In fact, we could not find a simple known dictionary that could be used as is for blacklisting. Hence, we conclude that the blacklist on the server-side is most likely a custom compilation.

Finally, if a password does not contain sequences or is not blacklisted, it is ranked as follows: secure, if it at least 7-character long; weak (non-blocking), if it has 6 characters; or too short otherwise.

Weaknesses. Although the meter's bar is very stringent on the number of characters (i.e., 254 are required to fill the bar completely), the bar is colored in green when the result of the server-side check is secure. Such a result is returned too frequently, granting Yandex the first position in our comparison in terms of overestimating password strength (see Fig. 1 under Section 5.3).

3.5. Changes in Apple, eBay and Yahoo!

During our previous experiments (June/July 2013), we were able to reliably reverse-engineer eBay's (fully) server-side meter. Its algorithm was as follows: 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 of six characters. Almost a year after we informed eBay, we evaluate again their algorithm with our YA and LinkedIn dictionaries (Nov. 2014). Although the core algorithm remains similar, we noticed that for more than 13% of YA dictionary, results do not match with our previous model. These passwords are one rank above their estimated rank. As it turns out, passwords with unexpected results are given varying feedback in time, e.g., *hongloan123* is expected to be weak, yet it may receive a medium score for no apparent reason after successive trials. We are left to conclude it is a buggy behavior.

Similarly, we previously assessed Apple's hybrid checker, which is sensitive to charset complexity. A blacklist check is performed on the server, which used to include parts of Top500, JtR and C&A. We tried to evaluate YA and LinkedIn dictionaries, however, beyond simple change in the expected format for the blacklist check, we found that the server reports negative results independently of the password requested after a number of consecutive tests. We also found that the meter reports unexpected negative results from a different browser under a different IP address shortly after we launched a series of tests. For this reason, we cannot properly evaluate our new dictionaries. The client-side evaluation remains mostly the same.

Yahoo! dropped its checker altogether soon after our initial tests and resorted to a more stringent policy only.

4. EMPIRICAL EVALUATION OF PASSWORD MANAGERS

Password meters are also available in several password managers. These managers help users choose better passwords (or generate random ones) on any websites, and store all user passwords in a password-protected encrypted vault. The master password creation is guided by a meter that plays a much more important role than the ones in web services. Thus, one would expect better meters in such specialized software. We evaluate the meters found in 1Password, LastPass, KeePass and RoboForm. We follow the same analysis structure as for web-based meters.

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 by a password meter. This meter shows both a continuous progress bar and a label. However, as this application is closed-source, we treat its algorithm as a black-box. We later noticed that the browser extension that comes with the application, is also equipped with a checker, which is significantly different from the one implemented in the stand-alone application. We were not expecting 1Password to implement two different meters, so we stopped at the first one we encountered (i.e., the application/main meter) and conducted its analysis before accessing the second meter (i.e., the browser meter) that is only available after the creation of a password vault protected by the master password. The main meter is also used to evaluate and present the strength of each password stored in the vault. The 1Password browser extension generates random passwords in the browser; however, users can modify or rewrite the passwords completely, and get feedback on the resulting strength from the browser meter. We first present the analysis of the black-box meter, as it demonstrates the possibility of analyzing closed-source compiled implementations.

Main password strength meter algorithm. Our method for uncovering the algorithm behind this black-box is to identify the features used and understand their implications. 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 and can be queried automatically. The thresholds for label assignment are therefore easy to deduce.

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 score, based on the output for Top500. Moreover, we can approximately identify that the scores from Top500 vs. the corresponding passwords length form two straight lines, and those from phpBB form four straight lines (scores capped at 100). Considering the composition of these dictionaries (one/two charsets in Top500, and one to four charsets in phpBB), we infer that the number of charsets is another feature used in the algorithm. We analyzed how special characters are counted: not all of them are considered, and multibyte characters are not processed properly on the master password's selection screen, i.e., the password length is counted as the number of bytes instead of the number of characters (however, the evaluation of passwords in the records list is done correctly).

We can infer the rule separating the charset lines. For the same length passwords, we observe that scores are linked together by a single coefficient (1.2), derived as follows. If we take passwords of length 13, for the number of charsets between 1–4, we get scores of 47, 56, 67, and 81, respectively (i.e., scores here depend only on the number of charsets). These scores when rounded to the nearest integer can be linked as: $47 \cdot 1.2^3 \simeq 56 \cdot 1.2^2 \simeq 67 \cdot 1.2^1 \simeq 81$. We infer that the algorithm fits the following model for scores between 0 and 100 ($\|.\|$ denotes the nearest integer of a given value):

$$score = \left\| \alpha \cdot \left(length \cdot \beta^{\#charsets-1} \right) + \gamma \right\| ; \text{here: } \alpha, \gamma \in \mathbb{R},\ \beta = 1.2$$

Since there is no justification for providing non-zero score to a zero-length password, we set γ to 0 (matching our observations). Also, we believe the coefficient α is derived from a simple expression due to the apparent simplicity of this meter. Finally, as the four candidate values for α obtained by linear regression oscillate around 3.6, we choose to set $\alpha = 3.6$, resulting the following model:

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

At last, we need to infer the mapping from scores to the six possible categories as follows: Terrible (0–10), Weak (11-20), Fair (21-40), Good (41-60), Excellent (61-90), or Fantastic (91-100). Note that to accommodate the observed scores for passwords containing a space and only non-recognized characters, we add an additional check to our algorithm (see Algorithm 1 in the appendix). Our algorithm yields zero error when evaluated against the rest of our dictionaries, indicating that we found the exact algorithm of 1Password's application meter.

Browser extension strength meter algorithm. Surprisingly, 1Password uses a different password meter in its browser extension. This meter evaluates randomly generated passwords; however, the user is allowed to modify a generated password as she wishes, and the meter also reacts to user modifications. It differs from the one ranking the master password in the stand-alone application in three ways: it is aware of some leet transformations and mangling; it performs a blacklist check against a 228,268-word dictionary (it lowers the evaluated password length by the length of the core word found in the blacklist) ; and it differs in the way of identifying groups of characters (it considers two sets of symbols found on a US keyboard instead of one).

Label assignment is identical in both versions; however, for the extension meter, the label is used only to color the meter's bar and is not displayed to the user. We noticed that a coefficient of 1.2 is repeated several times throughout the algorithm, rewarding

the presence of different charsets; we reverse-engineered the same coefficient in the application version of the meter.

Weaknesses. An obvious drawback of 1Password is its use of two significantly different checker algorithms. Users may not understand why the same password is ranked differently by the same password manager. In addition, the master password that protects all secrets is guided by the worse of both meters. The application meter is dependent only on the password length and number of charsets used, and lacks any further checks such as blacklisting, pattern matching, and leet transformations. Thus, it is unable to catch many weak passwords. Moreover, due to the way labels are assigned, most passwords fall under the "fair" category, even from the Top500 dictionary (the meter has two labels below fair). Examples of weak passwords labeled as good include *password123*, *princess12* and *123456789a*; excellent passwords include *123456789123456789* and *1q2w3e4r5t6y7u8i9o0p*; and finally, *abcdefghijklmnopqrstuvwxyz* is labeled as fantastic.

Strengths. The extension meter performs better than its stand-alone application counterpart. It is able to assign scores between "terrible" and "fair" in a sound way; e.g., simple dictionaries (Top500, Cfkr and JtR) are mostly assigned terrible and weak scores, while their mangled versions only achieve a fair score. It detects some leet transformations, and correctly identifies blacklisted words in its dictionary even when the password is derived from a dictionary word with three additional non-letter characters placed at the beginning and/or end of the word. Thus, it avoids assigning good or better scores for our mangled dictionaries.

Updated version. Seven months after the company developing 1Password became aware of our results, we tested a newer version of this application (v4.1.0.526, as of Nov. 23, 2014). The major change is the consistency between the algorithm run in the browser and the stand-alone application. The JavaScript version was dropped and the evaluation is now fully handled by the application. Also, the algorithm is slightly different compared to the one we reverse-engineered for v1.0.9.340. Indeed, we found that the coefficients α and β are changed to 3.08 and 1.1 respectively, instead of 3.6 and 1.2. Also, the issue of counting characters is now swapped: the master password evaluation counts correctly the number of characters, while the bars evaluating each recorded passwords interprets the number of bytes taken by a password as its length. Finally, a blacklist check is now performed against the Top 10000 dictionary, but not as thorough as the previous browser extension (i.e., now a simple equality check of the lowercased password). If the password matches a blacklisted word, its score downgrades to 20, which is weak. In Fig. 1 (Section 5.3), we evaluate YA and LinkedIn against this new version of the algorithm. The other dictionaries are evaluated against the previous version of the application for Windows.

4.2. LastPass

LastPass is a password manager extension for popular web browsers and mobile platforms. Users create a LastPass account using a master password evaluated by a meter. The master 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 password that the user can further modify (similar to 1Password's browser extension). The meter does not come with a label associated with the evaluated strength, and provides only a score between 0 and 100. Only LastPass 3.1.1 for Firefox (as of March 25, 2014) is evaluated in our analysis.

Algorithm. The LastPass meter consists of only 28 lines of JavaScript 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 (i.e., 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 several patterns are checked sequentially that increase the score: the password length is again added to the score if it is less than 5 characters; 6 points are added if it has between 5 to 7 characters; 12 points if it has between 8 and 15 characters; 18 points if it is longer; 1 point for having lowercase letters; 5 points each time any of the following matches: uppercase letters, digits, symbols, at least three digits, at least two symbols; 2 points for having both lower and uppercase letters; 2 points for having both the previous rule and digits; 2 points for having both the previous rule and symbols. Finally, the strength is doubled and truncated between 0 and 100.

Weaknesses and user information. Many weak passwords are assigned decent scores; e.g., *password* is ranked at 42%, and *Password1* at 72% (due to the number of charsets and 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.

The password is verified against the email address by more than a simple equality check, accounting for one included in the other. For the case where only the username part of an email is 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.

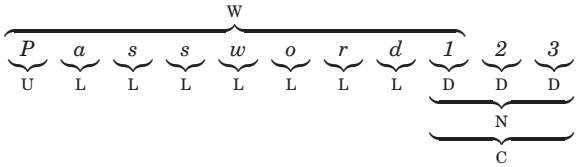
4.3. KeePass

KeePass is an open-source password manager, which comes with a brief explanation of how the manager's password meter is designed as described on the KeePass help center webpage at keepass.info. We analyze KeePass version 2.25 for Windows (as of March 25, 2014). The password meter appears during the creation of a new password-encrypted vault and a new entry in a vault. The meter represents the strength of a given password as both an entropy score in bits and a progress bar filled at 128 bits. The translation to labels provided on the KeePass help center 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 into five equal ranges since no label is provided to users. Moreover, this proposed 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).

Algorithm. KeePass uses a rather comprehensive algorithm as its checker (similar to Dropbox). It detects several patterns in a password, including: repeated substrings of length 3 or more (e.g., as in *passwordpassword* or *winter14winter*), groups of 3 digits or more, 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). The checker also 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. 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 is decomposed into all possible (with likely overlapping) patterns, each having an entropy computed based on the pattern type. See Algorithm 2 in the ap-

pendix for full details. Note that when detecting patterns of a certain type (R, W, 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. For example, 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:



The overall entropy for each combination of patterns is calculated. Simple characters that are not part of R, W, C or N belong to the corresponding groups previously identified (L, U, D, S, H or X) and whose entropy is defined as $\log_2(group\ size)$, respectively. In the example above, all combinations include ULLLLLLLDDD, ULLLLLLLN, ULLLLLLLC and WDD. In contrast to Dropbox, which simply combines entropies together by addition and considers non-dictionary string as random string, KeePass relies on an "optimal static entropy encoder" (as mentioned on the KeePass help center webpage). This encoder first takes into account the sum of entropies and includes a cost associated to the number and nature of the patterns found in the password (having more patterns yields more complex password). At last, it computes the entropy of unrecognized strings (remaining characters in L, U, D, S, H, X, that are not part of any other patterns) based on their characters distribution among the single-character sets. The final score is assigned with the entropy of the combination of patterns that yields the minimum value. KeePass yields a slightly bigger combined entropy for the same identified patterns compared to Dropbox's algorithm.

Strengths. A special extended symbol charset takes into account international symbols beyond the symbols available 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 among the meters we tested; the table incorporates more than a hundred transformations, including simplifications of accented letters (e.g., é $\rightarrow$ e). KeePass rejects our simple dictionaries and their mangled and leet-transformed versions without involving a stringent policy. In fact, no policies are enforced on password input/ranking. Such a design is more realistic, and remarkable in contrast to meters showing good results only with stringent policy checks.

Weaknesses. KeePass is primarily intended for traditional passwords that contain random characters or one or two core words. No special consideration is given to passphrases; e.g., *love is in the air* ranks 76 bits (59%). In this case, only *love* is part of the dictionary. Dropbox is more optimized in this regard, as it detects small words such as "the" or "is". This example passphrase 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 (without depending on policies). Finally, although extensive checks are performed to detect sequences, the algorithm does not check for spatial keyboard combinations, e.g., *a1s2d3f4g5h6j7k8*, which is ranked 81 bits (63%).

4.4. RoboForm

RoboForm is a password manager that comes both as a stand-alone application and a browser extension. Unfortunately, the browser extension calls functions implemented

in the application binary through a Dynamic-Link Library for Windows; thus, the source code for password evaluation is unavailable (i.e., not written in JavaScript). The output profile of this checker is totally different than all other meters we evaluated. 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. Details about how we reverse-engineered the application are given in Section 2.2.4.

Algorithm. The debugging of the program reveals several checks run against a candidate password. At first, we identified 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 considered). These functions are then part of more refined checks. This gives a first idea about 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 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-fledged equivalent algorithm since we gained enough insights to understand its strengths and weaknesses. Also, understanding all the details of the algorithm from reverse-engineering the binary file is a time-consuming task.

Strengths. The algorithm checks for several patterns with a particular emphasis given to keyboard sequences. RoboForm is the only algorithm that embeds a list of sequences in its dictionary, in addition to the ones that are easy to check programmatically. It also catches the most of TCJ passwords and assigns them a score below good.

Weaknesses. RoboForm fails to detect many mangled passwords and does not consider 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 simplest 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.

5. RESULTS ANALYSIS

Below, we further analyze our results from Sections 3 and 4, list several common features and weaknesses, and compare the meters.

5.1. Summary

Here we summarize our findings by grouping the meters according to the core algorithm they are relying on. We also consider the presence of several features.

Most checkers we evaluated heavily rely on the presence of characters included into the four main charsets (lower/uppercase letters, digits and symbols); hence we name their category as LUDS. LUDS checkers, especially if combined with a dictionary

check, mainly reward random-looking passwords, and discourage passphrase-looking passwords that include only lowercase letters. Most LUDS checkers we evaluated (12/18), combine charset complexity with password length. 7 LUDS checkers involve a dictionary check; only 4 consider mangling and leet transformations. 6 LUDS checkers perform further check for patterns; 4 of them search for patterns only to increase the strength score. Among the remaining 3 non-LUDS checkers, 2 of them (Dropbox and KeePass) mainly rely on advanced pattern checking, e.g., 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 due to its black-box nature and inconsistent output. Table 5.1 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; see Table 5.1. Most embedded dictionaries contain a significant portion of Top500, Cfkr and JtR (but not completely). 1Password is almost fully taken from C&A dictionary (99.58% come from it, which represents 74.11% of C&A). Dropbox's dictionary includes most of KeePass and Twitter dictionaries.

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 more than 5.8 million of them (about 66%). 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, possibly the most baffling example is *Password1*, which receives the widest 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 mostly weak in v2, and 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* is weak and *password0+* is strong as per Google. Finally, as expected, a random password *+ˆv16#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 stringent minimum requirements.

One possible consequence of these heterogeneous behaviors is user confusion with regard to their password strength. When opposite strength values are given for the same password by different services, users may not understand the reason behind such results, which may discourage them from choosing stronger passwords. 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 2013]), which in turn may encourage users to reuse such weak passwords for

Table III. 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. ˜0 means less than 0.01%

	Top500	Cfkr	JtR	C&A	RY5	phpBB	Top500+M	Cfkr+M	JtR+M	RY5+M	Leet	Yahoo!	LinkedIn
Top500	-	6.61	84.37	89.18	99.00	95.99	0	0	2.20	0	0	95.59	74.75
Cfkr	18.23	-	45.86	46.41	93.92	86.74	2.76	0	3.87	1.66	0	81.77	77.90
JtR	11.88	2.34	-	73.34	95.99	85.08	6.21	0.59	0	0.82	0	82.88	72.81
C&A	0.15	0.03	0.85	-	8.59	4.03	0.06	˜0	0.18	0.23	0	3.98	10.32
RY5	0.09	0.03	0.60	4.68	-	7.73	1.22	0.11	4.86	0	˜0	12.03	42.87
phpBB	0.26	0.09	1.64	6.70	23.61	-	0.74	0.11	2.07	2.19	˜0	10.82	25.86
Top500+M	0	0.02	0.98	0.75	30.44	6.09	-	4.40	83.84	19.25	0	10.26	32.86
Cfkr+M	0	0	0.45	0.32	12.73	4.43	21.08	-	43.19	15.82	0	5.52	19.95
JtR+M	0.01	˜0	0	0.38	18.75	2.61	12.95	1.39	-	17.99	0.01	5.04	21.30
RY5+M	0	˜0	˜0	0.03	0	0.19	0.20	0.03	1.21	-	0.01	0.36	4.60
Leet	0	0	0	0	˜0	˜0	0	0	˜0	0.03	-	˜0	0.02
Yahoo!	0.14	0.04	0.86	3.56	19.77	5.83	0.67	0.08	2.15	2.31	˜0	-	22.96
LinkedIn	0.01	˜0	0.05	0.62	4.74	0.94	0.15	0.02	0.61	1.96	˜0	1.54	-
1Password	0.12	0.02	0.63	99.58	7.60	3.50	0.04	˜0	0.20	0.23	0	3.66	10.13
DropBox	0.58	0.17	3.50	35.66	39.43	17.53	0.76	0.10	2.01	2.75	0	18.40	40.10
FedEx	14.49	4.77	45.23	98.76	89.05	75.09	0.88	0.18	1.94	0.53	0	76.68	92.05
KeePass	4.89	1.30	27.49	57.81	89.93	74.57	4.75	0.34	7.68	0.41	0	72.80	71.40
Microsoft v1	6.08	1.51	23.78	98.58	65.22	46.98	0.35	0.04	0.71	0.22	0	46.32	56.39
RoboForm	1.57	0.16	5.08	36.32	40.23	16.65	0.15	0.02	0.52	0.73	0	16.92	34.11
Twitter	93.77	7.98	85.54	87.03	98.00	95.01	0.25	0	2.74	0	0	95.26	97.51

	1Password	DropBox	FedEx	KeePass	Microsoft v1	RoboForm	Twitter
Top500	54.71	99.60	16.43	99.80	27.45	91.78	75.35
Cfkr	25.97	77.90	14.92	72.93	18.78	25.41	17.68
JtR	40.25	83.92	7.22	78.96	15.12	41.64	9.68
C&A	74.11	9.89	0.18	1.92	0.72	3.44	0.11
RY5	3.08	5.96	0.09	1.63	0.26	2.08	0.07
phpBB	4.33	8.09	0.23	4.12	0.57	2.63	0.21
Top500+M	0.39	2.88	0.02	2.15	0.04	0.20	˜0
Cfkr+M	0.17	1.75	0.02	0.75	0.02	0.13	0
JtR+M	0.31	1.17	0.01	0.54	0.01	0.10	0.01
RY5+M	0.02	0.11	˜0	˜0	˜0	0.01	0
Leet	0	0	0	0	0	0	0
Yahoo!	2.44	4.57	0.13	2.16	0.30	1.44	0.11
LinkedIn	0.45	0.67	0.01	0.14	0.02	0.19	0.01
1Password	-	9.12	0.17	1.69	0.65	1.34	0.11
DropBox	24.47	-	0.58	11.89	2.00	13.97	0.46
FedEx	68.02	86.75	-	72.97	96.11	94.88	14.49
KeePass	37.86	99.37	4.06	-	8.76	28.68	3.84
Microsoft v1	65.35	75.51	24.13	39.57	-	44.94	4.79
RoboForm	10.48	40.87	1.85	10.04	3.48	-	1.25
Twitter	60.85	97.51	20.45	97.51	26.93	90.52	-

other more important accounts. However, the real effects of wrong/incoherent meter outcomes on users may be demonstrated only by a large-scale user study.

5.3. Comparison

In Sections 3 and 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 2.1), we simplified the scales for our comparison. Fig. 1 shows the percentages 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 up to 128). For LastPass, we count scores greater than 50%. See thresholds in Table 2.1. Clearly, such scores should not be given to most of our test set (possible exceptions could be the complex passwords from leaked dictionaries).

Table IV. 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. "Leet" represents whether a leet transformations are taken into account. "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, the last four columns are noted with a "–".

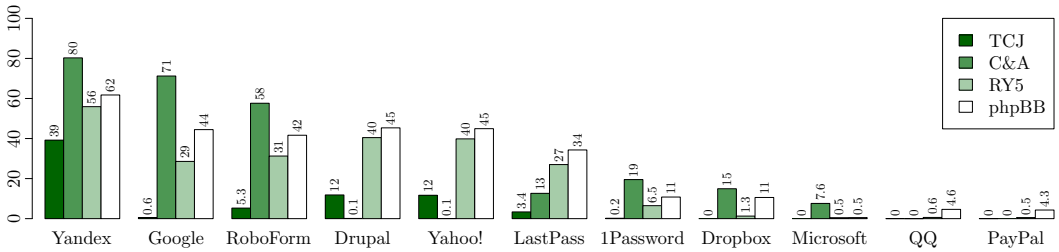
Name	Type	Length	Patterns	Dictionary and variations			
				Basic	Leet	Mangling	Multiple
Dropbox	Advanced patterns	✓	↑↓	✓	✓	✓	✓
Drupal	LUDS	×	×	–			
FedEx	LUDS	✓	×	–			
Microsoft v1	LUDS	✓	×	✓	✓	×	×
Microsoft v2	LUDS	✓	×	–			
Microsoft v3	LUDS[1]	✓	×	–			
Tencent QQ	LUDS	×	×	–			
Twitter	LUDS	✓	↑	✓	×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

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

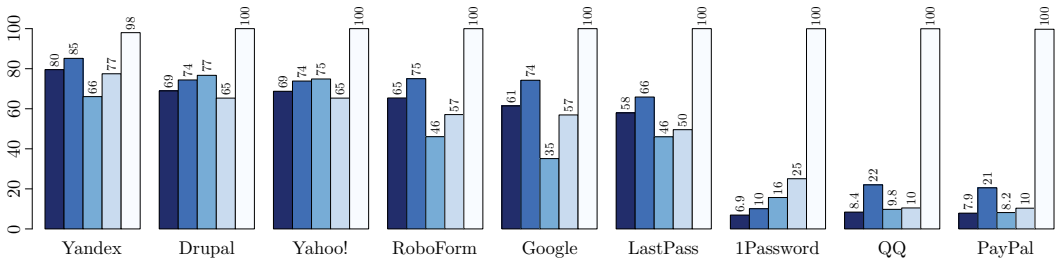
In reality, Google, RoboForm, Drupal, Yahoo! and LastPass assign decent scores to passwords from our base dictionaries; see Fig. 1a. Non-negligible percentages of Top500 (1.6%), Cfkr (15.5%) and JtR (12%) are qualified as good both by Drupal and Yahoo!; these checkers also tag roughly 40% of RY5 and 45% of phpBB passwords as good. This similarity 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 categorize our base dictionaries mostly 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, 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).

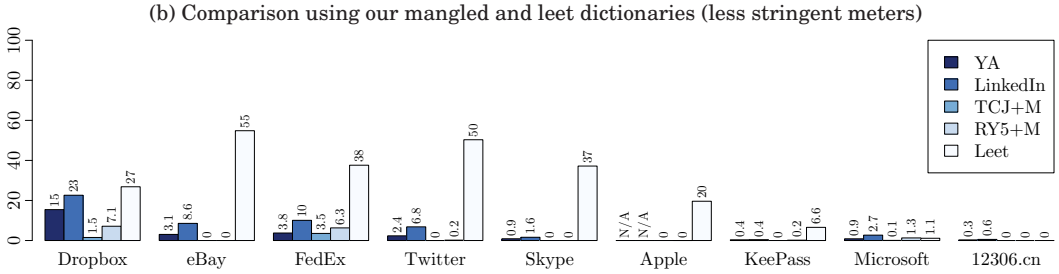
Our comparison graphs are not a competitive ranking of the meters. Although the ones that evaluate many of our dictionaries as good certainly have design deficiencies, it does not imply the remaining meters are good. For example, 12306.cn seems to be the "best" checker based on Fig. 1c as it does not classify any of our passwords as



(a) Comparison using our base dictionaries; the meters that are unlisted assign a decent strength score to very few passwords from these dictionaries (close to 0%).



(b) Comparison using our mangled and leet dictionaries (less stringent meters)



(c) Comparison using our mangled and leet dictionaries (more stringent meters)

Fig. 1. Comparison between services assigning decent scores to our base and mangled/leet dictionaries. Microsoft is represented by its latest checker, and 1Password is represented by its application version (v4 for YA and LinkedIn, v1 otherwise). N/A represents passwords that are not evaluated due to site policies.

good. However, this is achieved only due to the mandatory presence of the underscore character "_" (along with digits and mixed-case letters) for a password to be ranked as good. Clearly, this particular requirement does not reflect a good design. Hence, Fig. 1 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 (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 by web servers (see e.g., [Bonneau and Xu 2012]). We briefly discuss how such characters are taken into account by different meters.

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 all meters except Dropbox, Drupal 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 [Bonneau and Xu 2012]. Passwords are also correctly escaped with percentage as part of 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, as *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. Also, UTF-8 characters are sent in a raw format (the proper Content-Type should be `multipart/formdata` in this case [Bonneau and Xu 2012]). To get the same output as the PayPal website, we carefully implemented this buggy behavior in our tests.

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 if such obfuscation is beneficial). Along with hybrid checkers, a blacklist can be updated more easily than if it is hard-coded in JavaScript. Most checkers in our study are also quite simplistic: they apply simple rules with regard to password length and charset complexity, and sometimes detect common password patterns; this observation also stands for server-side checkers. 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 with KeePass 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 [Shay et al. 2010]. However, most checkers (except Dropbox, 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 Fig. 5.3.

5.7. 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); 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 stores arguably less sensitive information. Since our evaluation, Yahoo! has removed its strength meter, and now relies only on a stringent policy.

6. DIRECTIONS FOR BETTER CHECKERS

In this section, we briefly discuss few suggestions to improve current meters as apparent from our analysis. For challenges in designing a reliable meter, including implications of online vs. offline attacks, password leaks, passphrases, and relative performance of dictionaries, see our NDSS paper [Carnavalet and Mannan 2014].

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 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 [Wheeler 2012] (under a permissive license). Meters embedded in software such as password managers should also consider KeePass' open-source implementation (under GNU GPLv2).

However, the limitation of embedded dictionaries in meters is still present. One of the common basic weaknesses is to assign a large entropy to a section of a password that is not identified as a dictionary word or as another pattern. In practice, an attacker may try to use a more exhaustive dictionary than the ones considered by these meters, before resorting to a simple unintelligent brute-force attack. Meters may become more realistic if they were to include known dictionaries available to attackers. The total number of dictionary words may then reach a hundred million (considering the Openwall Project's, CrackStation's, and Wikipedia-derived wordlists), which is impractical to embed in client applications. Compression algorithms such as the ones proposed in the 1990s or early 2000 [Spafford 1992; Davies and Ganesan 1993; Bergadano et al. 1998; Blundo et al. 2004] to compress megabytes into kilobytes may be reused to compress gigabytes into megabytes.

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 (as adopted by Intel's independent password checker,[4] although with a poor implementation). Meters can also easily improve their detection of weak passwords by leveraging known cracking techniques and common password dictionaries. In contrast, labeling passwords as perfect may often lead to errors (seemingly random passwords, e.g., *Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn1* may not be as strong as they may appear [ArsTechnica.com 2013]).

7. RELATED WORK

Below we discuss few selected studies related to password policies, meters and cracking. A more comprehensive discussion is available elsewhere [Carnavalet 2014].

Florêncio and Herley [2010] review the password requirements of 75 commercial, government, and educational websites. They report that stringent policies are unexpectedly found at websites that may not need them, while easily-interchangeable services adopt more lenient policies for usability. This may explain why services as Google, Yahoo! and Yandex Mail have very lenient meters as evident from our results.

In a recent user study, Ur et al. [2012] tested the effects of 14 visually-different password meters on user-chosen password creation for a fake email account (users were informed that passwords were the object of study). They found that meters indeed positively influence user behavior and lead to better password quality. Users tend to reconsider their entire password when a stringent evaluation is given, rather than trying to bypass the checker. Passwords created under such strict evaluation were significantly more resistant to guessing attacks. However, meters with too strict policies generally annoyed users and made them put less emphasis on satisfying the meters. We focus on the algorithms behind several currently deployed meters, and identify weaknesses that may negatively impact regular users.

Egelman et al. [2013] also reported positive influence of password meters. This study also considered context-dependent variations in the effectiveness of meters, and found that passwords created for an unimportant account are not significantly influenced by the presence of a meter. They found that the presence of a meter during password change for an important account resulted in stronger user-chosen passwords. The idea of a peer-pressure meter design was also introduced, where a user is given feedback on the strength of her password compared to all other users of a particular service.

Furnell [2011] 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.

Castelluccia et al. [2012] leverage the use of Markov models to create an adaptive password-strength meter (APSM) for improved strength accuracy. Strength is estimated by computing the probability of occurrence of the n-grams that compose a given

[4]https://www-ssl.intel.com/content/www/us/en/forms/passwordwin.html

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).

In its community-enhanced version, John the Ripper [OpenWall.com 2014] 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" [OpenWall.com 2014]. In fact, this mode is an implementation of a 2005 proposal from Narayanan and Shmatikov [2005], which predicts the most probable character to appear at a certain position, given the previous characters of a password. The Markov mode in JtR 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.

8. 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. [Herley and van Oorschot 2012]). 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, 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, Yandex, PayPal) and popular password manager applications (e.g., LastPass, 1Password) 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 and application developers. 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 and password managers to rethink their meter design, and encourage industry and academic researchers to join forces to make these meters an effective tool against weak passwords.

ACKNOWLEDGMENTS

We are grateful to anonymous NDSS2014 and TISSEC reviewers for their insightful suggestions and advice. We also thank the members of Concordia's Madiba Security Research Group, especially Arash Shahkar and Jeremy Clark, for their suggestions and enthusiastic discussion on this topic. The second author is supported in part by an NSERC Discovery Grant and FRQNT Programme établissement de nouveaux chercheurs.

REFERENCES

ArsTechnica.com. 2013. How the Bible and YouTube are fueling the next frontier of password cracking. (8 Oct. 2013). News article. http://arstechnica.com/security/2013/10/how-the-bible-and-youtube-are-fueling-the-next-frontier-of-password-cracking/.

A. Barth. 2011. The Web Origin Concept. RFC 6454. (Dec. 2011). http://www.ietf.org/rfc/rfc6454.txt

Francesco Bergadano, Bruno Crispo, and Giancarlo Ruffo. 1998. High Dictionary Compression for Proactive Password Checking. *ACM Transactions on Information and System Security* 1, 1 (Nov. 1998), 3–25.

Matt Bishop and Daniel V. Klein. 1995. Improving System Security via Proactive Password Checking. *Computers & Security* 14, 3 (May/June 1995), 233–249.

Carlo Blundo, Paolo D'Arco, Alfredo De Santis, and Clemente Galdi. 2004. Hyppocrates: a new proactive password checker. *The Journal of Systems and Software* 71, 1-2 (2004), 163–175.

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

Mark Burnett. 2005. *Perfect Password: Selection, Protection, Authentication*. Syngress, Rockland, MA, 109–112. The password list is available at: http://boingboing.net/2009/01/02/top-500-worst-passwo.html.

Mark Burnett. 2011. 10,000 Top Passwords. (June 2011). https://xato.net/passwords/more-top-worst-passwords/.

William E. Burr, Donna F. Dodson, and W. Timothy Polk. 2006. Electronic authentication guidelines. NIST Special Publication 800-63. (April 2006). http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf.

Xavier de Carné de Carnavalet. 2014. *A Large-scale Evaluation of High-impact Strength Meters*. Master's thesis. Concordia University, Montreal.

Xavier de Carné de Carnavalet and Mohammad Mannan. 2014. From Very Weak to Very Strong: Analyzing Password-Strength Meters. In *Network and Distributed System Security Symposium (NDSS'14)*. San Diego, CA, USA.

Claude Castelluccia, Markus Dürmuth, and Daniele Perito. 2012. Adaptive password-strength meters from Markov models. In *Network and Distributed System Security Symposium (NDSS'12)*. San Diego, CA, USA.

CSO Online. 2014. After celeb hack, Apple patches password guessing weakness in iCloud. (2014). News article (Sep. 2, 2014). http://www.cso.com.au/article/553965/after_celeb_hack_apple_patches_password_guessing_weakness_icloud/.

Chris Davies and Ravi Ganesan. 1993. BApasswd: A New Proactive Password Checker. In *National Computer Security Conference*. Baltimore, MA, USA.

Serge Egelman, Andreas Sotirakopoulos, Ildar Muslukhov, Konstantin Beznosov, and Cormac Herley. 2013. 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.

Dinei Florêncio and Cormac Herley. 2010. Where do security policies come from?. In *Symposium On Usable Privacy and Security (SOUPS'10)*. Redmond, WA, USA.

Dinei Florêncio, Cormac Herley, and Baris Coskun. 2007. Do Strong Web Passwords Accomplish Anything?. In *USENIX Workshop on Hot Topics in Security (HotSec'07)*. Boston, MA, USA.

Dinei Florêncio, Cormac Herley, and P van Oorschot. 2014. An Administrator's Guide to Internet Password Research. In *USENIX LISA*. Seattle, WA, USA.

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

Nico Van Heijningen. 2013. *A state-of-the-art password strength analysis demonstrator*. Master's thesis. Rotterdam University.

Cormac Herley and Paul van Oorschot. 2012. A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy* 10, 1 (2012), 28–36.

Shiva Houshmand and Sudhir Aggarwal. 2012. Building Better Passwords using Probabilistic Techniques. In *Annual Computer Security Applications Conference (ACSAC'12)*. Orlando, FL, USA.

Immunity Inc. 2014. Immunity Debugger. (2014). https://www.immunityinc.com/products-immdbg.shtml.

Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L. Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. 2011. 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.

LifeHacker.com. 2008. Five Best Password Managers. (2008). Blog article (Aug. 08, 2008). http://lifehacker.com/5042616/five-best-password-managers.

Robert Morris and Ken Thompson. 1979. Password security: A case history. *Commun. ACM* 22, 11 (Nov. 1979), 594–597.

Arvind Narayanan and Vitaly Shmatikov. 2005. Fast dictionary attacks on passwords using time-space tradeoff. In *ACM Conference on Computer and Communications Security (CCS'05)*. Alexandria, VA, USA.

OpenWall.com. 2014. John the Ripper password cracker. (2014). http://www.openwall.com/john.

Oxid.it. 2014. Cain & Abel. (2014). http://www.oxid.it/cain.html.

PCMag.com. 2014. The Best Password Managers. (2014). Magazine article (Aug. 22, 2014). http://www.pcmag.com/article2/0,2817,2407168,00.asp.

Stuart Schechter, Cormac Herley, and Michael Mitzenmacher. 2010. 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.

Richard Shay, Saranga Komanduri, Patrick Gage Kelley, Pedro Giovanni Leon, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2010. Encountering stronger password requirements: user attitudes and behaviors. In *Symposium On Usable Privacy and Security (SOUPS'10)*. Redmond, WA, USA.

Sophos.com. 2009. Passwords used by the Conficker worm. (2009). Blog article (Jan. 16, 2009). http://nakedsecurity.sophos.com/2009/01/16/passwords-conficker-worm/.

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

TheNextWeb.com. 2014. This could be the iCloud flaw that led to celebrity photos being leaked. (2014). News article (Sep. 1, 2014). http://thenextweb.com/apple/2014/09/01/this-could-be-the-apple-icloud-flaw-that-led-to-celebrity-photos-being-leaked/.

Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. 2012. How does your password measure up? The effect of strength meters on password creation. In *USENIX Security Symposium*. Bellevue, WA, USA.

Rafael Veras, Christopher Collins, and Julie Thorpe. 2014. On the Semantic Patterns of Passwords and their Security Impact. In *Network and Distributed System Security Symposium (NDSS'14)*. San Diego, CA, USA.

Matthew Weir. 2010. *Using probabilistic techniques to aid in password cracking attacks*. Ph.D. Dissertation. Florida State University.

Matt Weir, Sudhir Aggarwal, Michael Collins, and Henry Stern. 2010. 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.

Dan Wheeler. 2012. zxcvbn: realistic password strength estimation. (10 April 2012). Dropbox blog article. https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/.

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

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

Appendix

ALGORITHM 1: 1Password: main password checking algorithm (ver: 1.0.9.340 for Windows)

Input: Candidate password
Output: Score in percentage and label related to the strength of the password
if *master password* **then**
    $length \leftarrow$ number of bytes taken by the password;
else
    $length \leftarrow$ number of characters in the password;
end
$charset \leftarrow 0$;
$symbols \leftarrow$ `!"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~`;
forall the *matching patterns* $\in$ {*password contains lowercase letters, uppercase letters, digits,* $symbols$} **do**
    $charset \leftarrow charset + 1$;
end
if *password contains spaces and* $charset = 0$ *and* $length > 0$ **then**
    $charset \leftarrow 1$;
end
$score \leftarrow \left\| 3.6 \cdot length \cdot 1.2^{charset-1} \right\|$;
if $score > 100$ **then**
    $score \leftarrow 100$;
end
$label \leftarrow getLabelFromScore(score)$;
`//returns: Terrible (0-10), Weak (11-20), Fair (21-40), Good (41-60),`
`//Excellent (61-90), or Fantastic (91-100).`
return $score$, $label$;

ALGORITHM 2: KeePass password checker pattern detection algorithm

Input: Password
Output: Password decomposed into overlapping patterns with their corresponding entropy
Assign each characters a pattern type from the 6 sets of characters (L, U, D, S, H, X);
forall the *repetitive patterns in password* **do**
    Consider identified substring as pattern of type R;
    Assign repetitive occurrences with entropy of $\log_2$(pattern offset $\cdot$ pattern length);
end
forall the *numbers of 3 digits or more* **do**
    Consider identified substring as pattern of type N;
    Assign $number$ an entropy of $\log_2(number)$;
    if $number$ *has leading zeros* **then**
        Add $\log_2$(number of leading zeros $+ 1$) to the entropy assigned;
    end
end
forall the *sequences of 3 characters or more whose code points are distanced by a constant* **do**
    Consider identified substring as pattern of type C;
    Assign pattern an entropy of $\log_2$(charset size matching the first character $\cdot$ (sequence length $- 1$));
end
forall the *substring of password* **do**
    $size \leftarrow$ number of words of same length in dictionary;
    if *substring included in lowercase in dictionary* **then**
        Consider identified substring as pattern of type W;
        $distance \leftarrow$ number of differences between substring and word in dictionary;
        Assign substring an entropy of $\log_2 \left(size \cdot \binom{n}{distance} \right)$;
    else if *unleeted substring included in lowercase in dictionary* **then**
        Consider identified substring as pattern of type W;
        $distance \leftarrow$ number of differences between substring and word in dictionary;
        Assign substring an entropy of $1.5 \cdot distance + \log_2 \left(size \cdot \binom{n}{distance} \right)$;
    end
end
return password with assigned patterns;