From *Very Weak* to *Very Strong*: Analyzing Password-Strength Meters

Xavier de Carné de Carnavalet and Mohammad Mannan
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada
{x_decarn, mmannan}@ciise.concordia.ca

***Abstract*—Millions of users are exposed to password-strength meters/checkers at highly popular web services that use user-chosen passwords for authentication. 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 to be a black box. We analyze password meters deployed in selected popular websites, by measuring the strength labels assigned to common passwords from several password dictionaries. From this empirical analysis with millions of passwords, we report prominent characteristics of meters as deployed at popular websites. We shed light on how the server-end of some 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 *very strong*. 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.**

I. Introduction

Proactive password checkers have been around for decades; for some earlier references, see e.g., Morris and Thompson [20], Spafford [28], and Bishop and Klein [3]. Recently, password checkers are being deployed as password-strength meters on many websites to encourage users to choose strong passwords. Password meters are generally represented as a colored bar, indicating e.g., a weak password by a short red bar or a strong password by a long green bar. They are also often accompanied by a word qualifying password strength (e.g., weak, medium, strong), or sometimes the qualifying word is found alone. We use the terms password-strength meters, checkers, and meters interchangeably in this paper.

The presence of a password meter during password creation has been shown to lead ordinary users towards more secure passwords [29], [11]. However, strengths and weaknesses of widely-deployed password meters have been scarcely studied so far. Furnell [12] 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., [8], [34]. 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 [7], and its weaknesses [34]. Better password checkers have been proposed (e.g., [8], [30], [25], [15]), 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 guiding the password choice of millions of users.

We evaluate the password meters of 11 prominent web service providers, ranging from financial, email, cloud storage to messaging services. Our target meters include: Apple, Dropbox, Drupal, eBay, FedEx, Google, Microsoft, PayPal, Skype, Twitter and Yahoo!. First, to understand these checkers, we extract and analyze JavaScript code (with some obfuscated sections) for eight services involving local/in-browser processing. We also reverse-engineer, to some extent, the six services involving server-side processing, which appear as black-boxes. Then, for each meter, we take the relevant parts from the source code (when available) and plug them into a custom dictionary-attack algorithm written in JavaScript and/or PHP. We then analyze how the meter behaves when presented with passwords from publicly available dictionaries that are more likely to be used by attackers and users alike. Some dictionaries come from historical real-life passwords leaks. For each meter, we test nearly four million passwords from 11 dictionaries (including a special leet dictionary we created). We also optimize our large-scale automated tests in a server-friendly way to avoid unnecessary connections, and repeated evaluation of the same password. At the end, we provide a close-approximation of each meter's scoring algorithm, weaknesses and strengths of the algorithm, and a summary of scores as received by our test dictionaries against the meter.

To measure the quality of a given password, checkers usually employ one of the two methods: they either enforce strong requirements, mostly regarding the length and character-set complexity, or they try to detect weak patterns such as common words, repetitions and easy keyboard sequences. Some checkers are implemented at client-end only, some at server-end only and the rest are hybrid, i.e., include measurements

NDSS '14, 23-26 February 2014, San Diego, CA, USA
 ISBN 1-891562-35-5
http://dx.doi.org/doi-info-to-be-provided-later

both at the server- and client-ends. We also analyze strengths and limitations of these approaches.

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

Contributions.

1) METER CHARACTERIZATION. We systematically characterize 13 password checkers from 11 widely-used web services to understand their behaviors. For Microsoft, we test three versions of their checker, two of which are not used anymore. 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).
2) EMPIRICAL EVALUATION OF METERS. For each of the 13 meters, we used nearly four million unique passwords from several password dictionaries (a total of 53 million test instances, approximately). This is the largest such study on password meters to the best of our knowledge.
3) METER WEAKNESSES. Weaknesses exposed by our tests include: (a) Several meters label many common passwords as of decent quality—varying their strengths from *medium* to *very strong*; (b) Strength outcomes widely differ between meters, e.g., a password labeled as *weak* by one meter, may be labeled as *perfect* by another meter; and (c) Many passwords that are labeled as weak can be trivially modified to bypass password requirements, and even to achieve *perfect* scores. These weaknesses may cause confusion and mislead users about the true strength of their passwords. Compared to past studies, our analysis reveals the extent of these weaknesses.
4) TEST TOOLS. We have implemented a web-based tool to check results from different vendors for a given password. In addition to making the inconsistencies of different meters instantly evident, this tool can also help users choose a password that may be rated as strong or better by all the sites (from our test set), and thus increasing the possibility of that password being effectively strong. Test tools and password dictionaries as used in our evaluation are also available for further studies.

We first explain some common requirements and features of the studied password checkers in Section II. In Section III, we discuss issues related to our automated testing of a large number of passwords against these meters. In Section IV, we detail the dictionaries used in this study, including their origin and characteristics, and some common modifications we performed on them. We present the tested web services and their respective results in Section V. In Section VI, we further analyze our results and list some insights as gained from this study. Section VII discusses more general concerns related to our analysis. A few related studies are discussed in Section VIII. Section IX concludes.

II. PASSWORD METERS OVERVIEW

Password-strength meters are usually embedded in a registration or password update page. During password creation, the meters instantly evaluate changes made to the password field, and output the strength of a given password. Below we discuss different aspects of these meters as found in our test websites. Some common requirements and features of different meters are also summarized in Table I.

(a) Charset and length requirements. By default, some checkers classify a given password as invalid or too short, until a minimum length requirement is met; most meters also enforce a maximum length. Some checkers require certain character sets (charsets) to be included. Commonly distinguished charsets include: lowercase letters, uppercase letters, digits, and symbols (also called special characters). Although symbols are not always considered in the same way by all checkers (e.g., only selected symbols are checked), we define symbols as being any printable characters other than the first three charsets. One particular symbol, the space character, may be disallowed altogether, allowed as external characters (at the start or end of a password), or as internal characters. Some checkers also disallow identical consecutive characters (e.g., 3 or 4 characters for Apple and FedEx respectively).

(b) Strength scales and labels. Strength scales and labels used by different checkers also vary. For example, both Skype and PayPal have only 3 possible qualifications for the strength of a password (Weak-Fair-Strong and Poor-Medium-Good respectively), while Twitter has 6 (Too short-Obvious-Not secure enough-Could be more secure-Okay-Perfect).

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

(d) Types. Based on where the evaluation is performed, we distinguish three main types of password checkers as follows. *Client-side*: the checker is fully loaded when the website is visited and checking is done locally only (e.g., Dropbox, Drupal, FedEx, Microsoft, Twitter and Yahoo!); *server-side*: the checker is fully implemented on server-side (e.g., eBay, Google and Skype); and *hybrid*: a combination of both (e.g.,

TABLE I. PASSWORD REQUIREMENTS AND CHARACTERISTICS OF DIFFERENT WEB SERVICES; SEE SECTION II FOR DETAILS. NOTATION USED UNDER "MONOTONICITY": REPRESENTS WHETHER ANY ADDITIONAL CHARACTER LEADS TO BETTER SCORING (NOT ACCOUNTING FOR USER INFORMATION CHECK). "USER INFO": ○ (NO USER INFORMATION IS USED FOR STRENGTH CHECK), AND ◐ (SOME USER INFORMATION IS USED OR ALL BUT NOT FULLY TAKEN INTO ACCOUNT). UNDER "CHARSET REQUIRED", WE USE 1+ TO DENOTE "ONE OR MORE" CHARACTERS OF A GIVEN TYPE. THE "ENFORCEMENT" COLUMN REPRESENTS THE MINIMUM STRENGTH REQUIRED BY EACH CHECKER FOR REGISTRATION COMPLETION: ∅ (NO ENFORCEMENT); AND OTHER LABELS AS DEFINED UNDER "STRENGTH SCALE".

Service	Type	Strength scale	Length limits		Charset required	Monotonicity	User info	Space acceptance		Enforcement
			Min	Max				External	Internal	
Dropbox	Client-side	Very weak, Weak, So-so, Good, Great	6	72	∅	No	◐	✓	✓	∅
Drupal		Weak, Fair, Good, Strong	6	128	∅	Yes	○[1]	×	✓	∅
FedEx		Very weak, Weak, Medium, Strong, Very strong	8	35	1+ lower, 1+ upper, 1+ digit	Yes	○	×	×	Medium
Microsoft		Weak, Medium, Strong, Best	1	–	∅	Yes	○	✓	✓	∅
Twitter		Invalid/Too short, Obvious, Not secure enough (NSE), Could be more secure (CMS), Okay, Perfect	6	>1000	∅	No	◐	✓	✓	CMS
Yahoo!		Weak, Strong, Very strong	6	32	∅	Yes	◐	✓	✓	Weak
eBay	Server-side	Invalid, Weak, Medium, Strong	6	20	any 2 charsets	Yes	◐	×	✓	∅
Google		Weak, Fair, Good, Strong	8	100	∅	No	○	×	✓	Fair
Skype		Poor, Medium, Good	6	20	2 charsets or upper only	Yes	○	×	×	Medium
Apple	Hybrid	Weak, Moderate, Strong	8	32	1+ lower, 1+ upper, 1+ digit	No	◐	×	×	Medium
PayPal		Weak, Fair, Strong	8	20	any 2 charsets[2]	No	○	×	×	Fair

[1] Partially covered in the latest beta version (ver 8), as of Nov. 28, 2013
[2] PayPal counts uppercase and lowercase letters as a single charset

Apple and PayPal). This distinction leads us to different approaches to automate our testing, as explained in Section III.

(e) Diversity. None of the 11 web services we evaluated use a common meter. Instead, each service provides their own meter, without any explanation of how the meter works, or how the strength parameters are assigned. For client-side 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* [35]. Dropbox also provides details of this meter and open-sourced it to encourage further development.

(f) Entropy estimation and blacklists. Every checker's implicit goal is to determine whether a given password can be easily found by an attacker. To this end, most employ a custom "entropy" calculator, either explicitly or not, based on the perceived complexity and length of the password. As discussed in Section VI, 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.

III. TEST AUTOMATION

For our evaluation, we tested nearly four million of passwords against each of the 13 checkers. In this section, we discuss how we performed such large-scale automated tests.

Client-side checkers. 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 (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 10min for a 2-million dictionary against the most complex meter (Dropbox).

Server-side checkers. 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[1] for handling HTTPS requests to a server-side checker. 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()`[2] call before its execution. This 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 4 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, 10 against eBay, 64 against Google (2.5 against PayPal and 8 against

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

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, we did not face any blocking mechanisms during our tests.

Hybrid checkers. 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.[3] To allow client-side cross-origin requests, the cross-origin resource sharing (CORS [36]) 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` [36], 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.

IV. Tested dictionaries

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

Overview and notes. Table II lists the 11 dictionaries we used, including their sizes, and maximum, average and standard deviation of their password length. Dictionary sources include: password cracking tools (John the Ripper and Cain & Abel), a list of 500 most commonly used passwords (Top500), an embedded dictionary in the Conficker worm, and leaked databases of plaintext or hashed passwords (RockYou and phpBB). We mostly chose simple and well-known dictionaries (as opposed to more complex ones, see e.g., [1], [2]), to evaluate checkers against passwords that are reportedly used by many users. We expected passwords from these non-targeted dictionaries would be mostly rejected (or rated as weak) by the meters.

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

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

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

As a side note, many passwords in the source dictionaries are related to insults, love and sex. We avoid mentioning such words as example passwords. We also noticed poor internationalization of dictionary passwords in general, where most of them originate from English. One exception is the RockYou dictionary, which contains some Spanish words (possibly due to some RockYou users being originated from Spanish-speaking countries). Finally, some leaked passwords contained UTF-8-encoded words that were generally not handled properly by the checkers (also some are malformed UTF-8 strings) [4]. 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 Top500, Cfkr and JtR among leaked passwords, e.g., 494 passwords of Top500 are included in RY5; see Table III.

A. Cracking tool dictionaries

Top500. This dictionary was released in 2005 as the "Top 500 Worst Passwords of All Time" [5], and later revised as Top 10000 [6] passwords in 2011. We use the 500-word version as a very basic dictionary. Passwords such as *123456*, *password*, *qwerty* and *master* can be found in it. Actually, a "0" is duplicated in this list, making it have only 499 unique passwords. Password composition: 91% lowercase letters only, 7% digits only, and 2% lowercase letters and digits.

Cfkr. The dictionary embedded in Conficker worm was used to try to access other machines in the local network and spread the infection. Simple words and numeric sequences are mostly used; example passwords include: *computer*, *123123*, and *mypassword*.[4] Password composition: 52.5% lowercase letters only, 29.8% digits only, 15.5% lowercase letters and digits, and 2.2% mixed-case letters.

JtR. John the Ripper [22] is a very common password cracker that comes with a dictionary of 3,546 passwords, from which we removed an empty one. Simple words can be found in this dictionary too; however, they are little more complex than those in Top500, e.g., *trustno1*. Password composition: 82.79% lowercase letters only, 8.12% lowercase letters and digits, 4.4% mixed-case letters, and few other charset combinations.

[3] http://www.w3.org/Security/wiki/Same_Origin_Policy.

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

TABLE III. DICTIONARY OVERLAPS (PERCENTAGE RELATIVE TO THE SIZE OF THE DICTIONARY FROM THE LEFT-MOST COLUMN)

	Top500	Cfkr	JtR	C&A	RY5	phpBB	Top500+M	Cfkr+M	JtR+M	RY5+M	Leet
Top500	-	6.61	84.37	89.18	99	95.99	0	0	2.2	0	0
Cfkr	18.23	-	45.86	46.41	93.92	86.74	2.76	0	3.87	1.66	0
JtR	11.88	2.34	-	73.34	95.99	85.08	6.21	0.59	0	0.82	0
C&A	0.15	0.03	0.85	-	8.59	4.03	0.06	~0[1]	0.18	0.23	0
RY5	0.09	0.03	0.6	4.68	-	7.73	1.22	0.11	4.86	0	~0
phpBB	0.26	0.09	1.64	6.7	23.61	-	0.74	0.11	2.07	2.19	~0
Top500+M	0	0.02	0.98	0.75	30.44	6.09	-	4.4	83.84	19.25	0
Cfkr+M	0	0	0.45	0.32	12.73	4.43	21.08	-	43.19	15.82	0
JtR+M	0.01	~0	0	0.38	18.75	2.61	12.95	1.39	-	17.99	0.01
RY5+M	0	~0	~0	0.03	0	0.19	0.2	0.03	1.21	-	0.01
Leet	0	0	0	0	~0	~0	0	0	~0	0.03	-

[1] ~0 means less than 0.01%

C&A. Another password cracking tool, Cain & Abel [23] comes with a 306,706-word dictionary that primarily consists of long lowercase words (e.g., *constantness*, *psychotechnological*). The composition of passwords is quite unique: 99.84% lowercase letters only. The rest is shared among lowercase letters and symbols (0.09%), lowercase letters and digits (0.05%), few digits only, symbols only, and lowercase letters with digits and symbols.

B. Real password database leaks

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

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

C. Mangling

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 [8], [18], [7]; for details on this wide-spread behavior, see e.g., Weir [32]. 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: (a) lowercase passwords that are not; (b) capitalize; (c) pluralize; (d) suffix with "1"; (e) combine (a) and (d); (f) duplicate short words (6 characters or less); (g) reverse the word; (h) prefix with "1"; (i) uppercase alphanumerical passwords; and (j) 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 [32]). 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.

D. 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 [25]. Leet transformations are not covered in our main dictionaries, apart from few exceptions; thus, we built our own leet transformed dictionary to test the effect of such transformations.

Our Leet dictionary is based on the passwords from Top500, Cfkr, JtR, C&A, phpBB, the full RockYou dictionary, along with the Top 10,000 dictionary, and a 37,141-word version of the leaked MySpace password dictionary,[5] 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

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

(e.g., *here4you*), so as to keep letter-only passwords. Passwords that are less than 6-character long are also dropped, while 6-character long ones are suffixed with a digit and a symbol chosen at random, and 7-character passwords are only suffixed with either a digit or a symbol. At this point, all passwords are at least 8-character long, allowing us to pass all minimum length requirements. Those longer than 20 characters are also discarded. The dictionary was reduced to 648,116 words.

We then apply leet transformations starting from the end of each password to mimic known user behaviors of selecting digits and symbols towards the end of a password (see e.g., [32], [13]). For these transformations, we also use a translation map that combines leet rules from Dropbox and Microsoft checkers. Password characters are transformed using this map (if possible), by choosing at random when multiple variations exist for the same character, and up to three transformations per password. Thus, one leet password is generated from each password, and only single character equivalents are considered (e.g., we do not consider more complex transformations such as V becoming double slashes: \/). The resulting Leet dictionary is composed of 77.56% 4-charset passwords, 18.66% mixed-case letters and digits, 3.72% mixed-case letters and symbols, and the rest of mixed-case letters only. Arguably, this dictionary is not exhaustive; however, our goal is to check how meters react against simple leet transformations. The near-zero overlap between this dictionary and the leaked ones (as in Table III) 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 [13], 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.

V. Meters Evaluation

For each password-strength meter evaluated, we present their general behavior, analyze the way they operate and discuss their strengths and weaknesses. Most of our tests have been performed between the months of June and July, 2013. Only Dropbox, Google, eBay, Apple and FedEx are presented in this section. Summary results of other meters are provided in Appendix A; for details, see [9].

A. Dropbox

Dropbox has developed a client-side password strength checker called *zxcvbn* [35], and open-sourced it to encourage others to use and improve the checker. Fig. 1 summarizes our results for Dropbox.

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

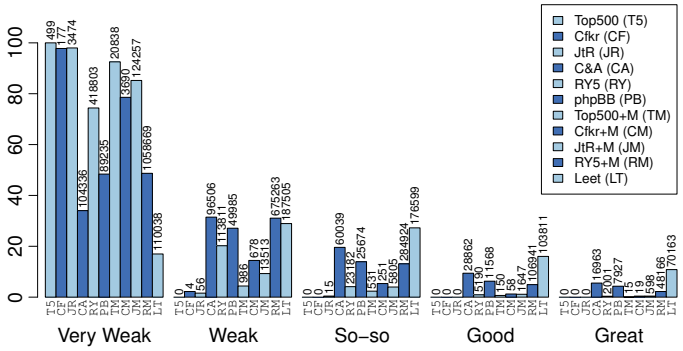

Fig. 1. Dropbox checker password strength distribution

patterns are considered weak altogether, and given a low entropy count.

To restrict common passwords, a candidate password is checked against five embedded, frequency-ordered dictionaries: 7140 passwords from the Top 10000 password dictionary [6]; 32544 English words from the Wikitionary project;[6] 1003 male, 3814 female names and 40582 surnames from the 2000 US Census. The presence of a subpart of the password in a dictionary is not forbidden or directly penalized in contrast to other password checkers involving a blacklist in our test. Such a subpart is assigned an entropy value based on the average number of trials that an attacker would have to perform, considering the rank of the subpart in its dictionary.

If no pattern is found for a given subpart of the password, it is considered a random string. The entropy of such string is computed based on a simple brute-force attack. For example, *MySuperP4$$w0Rd* is decomposed as *my* (5.3 bits of entropy based on a dictionary attack), *super* (9.4 bits), and a transformed *password* (about 5 bits; 1.58 for the leet transformation and 3.45 for the uppercase), which are all found quickly in the dictionaries; hence this password gets a "very weak" score (entropy of 19.8 bits). On the contrary, the partially randomly generated password *P4$$w0RdTBuK9Ye6MZkdyx* decomposes as a transformed *password* and something that does not match a dictionary word or pattern; hence, the latter part is considered to be found only by a brute-force attack (92 bits of entropy), granting the password a score labeled "great".

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

(b) User information. The first and last names and the email address of a registering user are added to a new dictionary in the algorithm to weaken the strength of a password containing these user-specific registration items. The part of the password that matches any registration item, even if transformed, is assigned a very low entropy. It is given 1 bit if it matches the first name, 1.58 for the last name, and 2 for the email address (because of the rank in the dictionary). Additional bits are assigned for transformations and uppercased letters. Overall, a password reusing a registration item will be significantly

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

weakened. However, there is a programming error that allows most users to bypass this filter, even unknowingly. If an item contains an uppercase letter (a probable case for the names), it will not be detected as a pattern, and thus a password with such items will not be weakened accordingly.

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

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

As zxcvbn promotes the use of passwords consisting of a combination of common words such as *correcthorsebatterystaple*,[7] many words from C&A are considered as good (9.4%), or even great (5.5%), as they mostly consist of long words (often a combination of simple words). Zxcvbn, however, fails for some trivial passwords that are not listed in its internal dictionary, e.g. from RY5, *evanescence*, *SEPULTURA* (an American rock band and a Brazilian heavy metal band, respectively) and *dolce&gabana* (an Italian luxury industry fashion house) are considered great. This highlights an important limitation of fixed embedded dictionaries. Finally, it is interesting to note that even though Dropbox made an effort to build a better-than-average password-strength meter, they do not enforce a minimum strength during the registration process, letting users register with a possibly very weak password.

B. Google

Google employs a server-side checker. Once the minimum requirement of 8 characters is met, an AJAX query to the server-side checker is made. The response provides the password's strength. Fig. 2 summarizes our results for Google.

(a) Algorithm. Google's checker is difficult to reverse-engineer because of its inconsistent output at times. Consider the following examples: *testtest* is weak and *testtest0* is strong, but *testtest1* is fair, *testtest2* is good and *testtest3* is strong again. It is no surprise that a simple repetitive string, like *testtest*, followed by a digit is considered weak, but generating such a variety of scores for so minor changes is difficult to comprehend. In addition, we found that *test1234* and *Test1234* are weak, while *TesT1234* is fair, *TeSt1234* is good, and *TEst1234* and *tEst1234* are strong. From these results, one can

[7] https://www.xkcd.com/936/

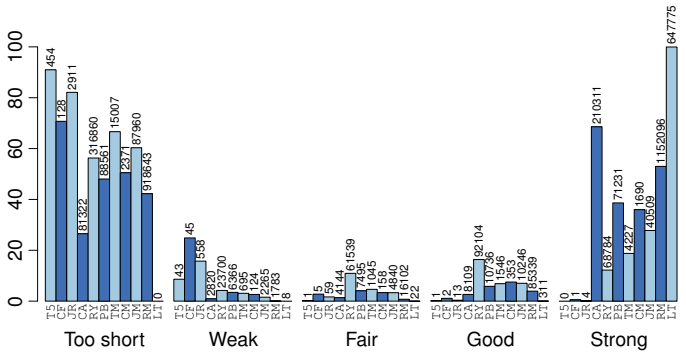

Fig. 2. Google checker password strength distribution

approximate that a first letter uppercase is not as rewarding as another letter being uppercased, and that a pattern of first and last letter uppercase is labeled as intermediate. If this approximation is true, then many commonly used patterns would be penalized. However, Google's checker is ranked first and third in terms of yielding "strong" passwords from our base and mangled dictionaries, respectively (see Section VI for more comparison).

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

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

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

(b) Weaknesses. Simple dictionary passwords can easily reach a good or strong strength. Examples include: *access14* in

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

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

C. eBay

eBay also uses a fully server-side checker (similar to Google). Unlike Google, the username and email address are also sent to the checker. The response provides the password's strength along with different messages for the user. Fig. 3 summarizes our results for eBay.

(a) Algorithm. A password is considered as invalid and is not sent to the server-side checker before it matches the length requirement. (In July 2013, eBay introduced a "Too short" feedback to replace the term invalid for short passwords.) The checker requires passwords to cover any two charsets; thus, many words from Top500, C&A and JtR are considered invalid. As a side-effect, concatenation of simple words always leads to weak passwords (in contrast to Dropbox).

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

(b) User information. eBay prevents a user from choosing a password similar to her user ID, taking into account case changes and few leet transformations as found in our test. For example, having *leettest* as the user ID, the passwords *LeetTest L3ettest* and *LEETTEST* are invalid; *L33ttest* and *lEEtt3st* are medium; and *LeetT3st+* is strong. In Table I, we rate eBay's checker as partially taking registration items into account, due to the exclusion of a user's real name from the check.

(c) Weaknesses. eBay does not consider any password as strong from our test dictionaries except few passwords from phpBB and RY5, in which we can find few relatively simple

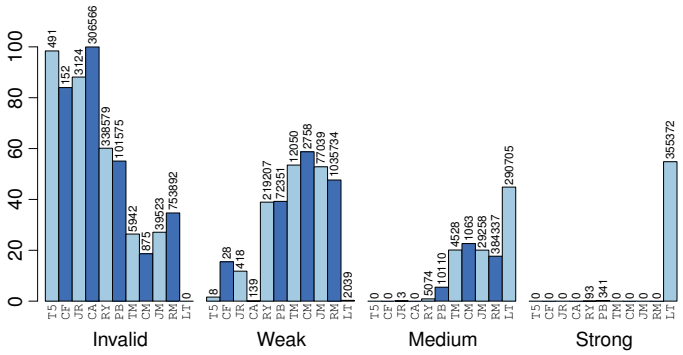

Fig. 3. eBay checker password strength distribution

passwords being labeled as strong, e.g., *P@ssw0rd*, *0b1W@n*, *sp3ciaL***, *phpBB2!* and *p0pm@iL* (found in the phpBB dictionary). These are simple leet transformations of common words with possibly leading special characters. Also, several common passwords are categorized as of medium strength, including, *Lucky1*, *Abc123* and *Password1*. Simply covering all four charsets is also enough by design to get a strong score, e.g., *AAaa1+*, resulting high scores for our leet dictionary.

D. Apple

The client-side/in-browser part of Apple's hybrid meter first checks if a password meets the policy requirements: 8-character long, has one lowercase, one uppercase, one digit, and does not include more than two identical consecutive characters. The password is then checked against a server-side blacklist. Blacklisted passwords are disallowed, irrespective of their strength as measured by the client-side checker. Fig. 4 summarizes our results for Apple. Note that we have created an extra category to group blacklisted passwords, even if they are categorized as moderate or strong.

(a) Client-side algorithm. The client-side part of the algorithm is based on an increasing-only score, adjusted by some rules with associated weights. The score is higher if a password contains more characters (by ranges of 6-7, 8-15 and 16+ characters). Having the following features also cumulatively contributes to higher scores: at least an uppercase letter; one or two digits; three digits or more; at least a symbol; symbols separated by other characters (thus, encouraging symbols inside a password rather than at the ends); lower and uppercase letters; alpha-numerical characters; and alpha-numerical-symbol characters. This algorithm clearly encourages charset complexity. Similar to FedEx and eBay's checkers, Apple's considers only `!@#$%^&*?_,~` as symbols (remaining special characters do not influence the strength but are counted in the password length). Once a password passes all requirements, it is at least tagged as "moderate", hence we report this strength in the enforcement column for Apple in Table I (even though a password can be rated as "strong" while not passing all requirements).

(b) Blacklist check. Apple does not provide any information about the blacklist checker. However, it is apparent from our results in Fig. 4 that the blacklist contains Top500, JtR and C&A, albeit modified versions, which may explain the small percentage of dictionary words that still passes the blacklist check. This is also supported by some example passwords from

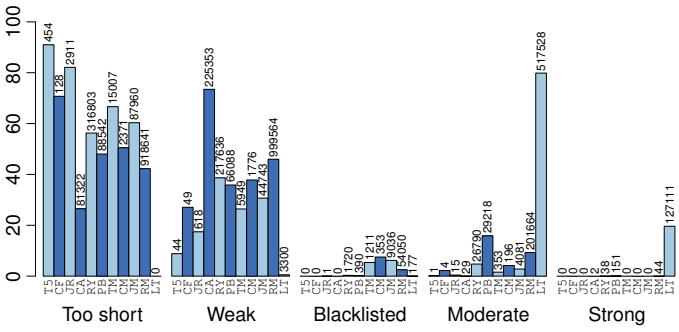

Fig. 4. Apple checker password strength distribution

these dictionaries; e.g., a rather unusual password, *Protransubstantiation1*, taken and mangled from C&A, is blacklisted. It appears that digits are removed from the dictionaries and only core words are kept in the blacklist. Passwords that do not pass the requirements during the client-side check, can still reach a moderate or strong strength; such passwords when modified to comply with the requirements, may then be blacklisted. For example, *Password1* and *A1b2c3d4* pass the client-side check, as they are long enough and satisfy charsets requirements, but fail the blacklist test (although they are labeled as moderate).

Many simple mangled passwords with the first letter uppercase and a terminal digit are caught by the server-side checker. For a given password, e.g., *Franklin123*, apparently the following steps are performed: the password is stripped from trailing digits, giving *Franklin*, and is then checked against a blacklist, disregarding its letter-case. For some base words like *Franklin*, up to three trailing digits are stripped during the blacklist check, but only one digit is removed for words such as *Adorable*. This behavior seems to originate from non-trivial rules that we cannot explain from our tests. However, adding extra terminating digits, and/or starting digits, easily bypasses the blacklist check.

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

(d) Weaknesses. Apple's policy requires that passwords should "not contain identical consecutive characters."[8] However, in practice, this restriction is too weak to catch repetitive patterns in some cases; e.g., *P4ssw0rd* is blacklisted but *P4ssw0rdP4ssw0rd* is labeled as strong. We also noticed that the blacklist check response times vary significantly—ranging from less than a second up to more than a minute. Very few dictionary passwords are considered strong, even when mangled. However, examples of strong passwords also include: *P@ssw0rd!*, *P@55w0rd* and *Robot123!* (mangled versions of simple passwords). Finally, a strong password can be blacklisted as it is the case for *Pa$$w0rd*, which is a unique state among other checkers.

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

E. FedEx

FedEx allows its users to register an online account to facilitate simple services such as shipment handling and tracking. Interestingly, FedEx's password checker is quite stringent even though FedEx hosts arguably far less sensitive information compared to other services in our evaluation; see Fig. 5 for summary results.

(a) Algorithm. The client-side checker is a simple 130-line JavaScript program with an embedded 566-word dictionary. Passwords matching any dictionary word are labeled as weak. The checker is aware of leet transformations, and normalizes a given password by reversing the transformation (if found). The password is also converted to lowercase before comparing it against the dictionary. These rules significantly weaken many passwords such as *P@$5W0rD*. Each strength level has specific rules to be matched. A password is very weak until it passes the basic requirements: 8 characters in length, the use of characters from three charsets (lowercase, uppercase and digit), and no three identical consecutive characters. The password can reach a medium strength if it has at least 4 unique characters and is not in the dictionary. The password becomes strong when it has 9 characters or more, in addition to at least 6 unique characters. Very strong passwords must be at least 10 characters long and have at least 6 unique characters; or 9 characters long, 6 unique characters and have at least one symbol. Only the characters `!@#$%^&*?,~` are considered in the symbols set (remaining other characters do not influence the strength but are counted in the password length). In contrast to other checkers tested, the password strength is not shown as a text label but only as a colored meter.

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

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

Also, a password not meeting the basic requirements (e.g., including characters from multiple charsets) cannot be better

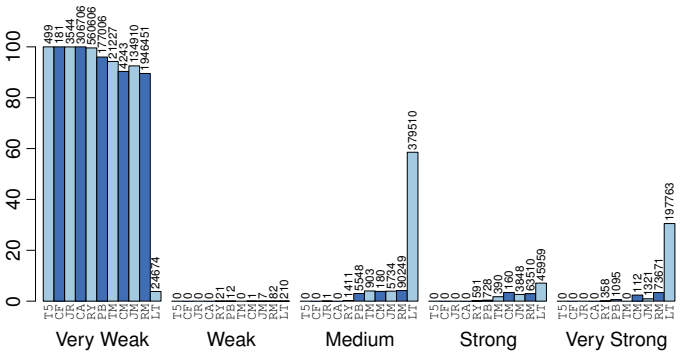

Fig. 5. FedEx checker password strength distribution

than very weak, implying that long memorable passwords (single-case passphrases) are denied in favor of more complex ones. This effectively bans all non-mangled cracking dictionaries. Among the mangled ones, although there is one rule that includes a first letter uppercase and a leading digit, few passwords are long enough to be considered strong or very strong.

VI. Results Analysis

Below, we further analyze our results from Section V, list several common features and weaknesses, and compare the meters. We also discuss a web-based password multi-checker tool that we implemented for testing any password across all the meters we evaluated. Beyond the generic dictionaries as discussed in Section IV, we also report the results of using a more targeted dictionary against the FedEx checker.

A. Meters heterogeneity and inconsistencies

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

Some inconsistencies are particularly evident when a password passes the minimum requirements. For example, *password$1* is correctly assigned very-weak by FedEx, but the score jumps to very-strong when the first letter is uppercased. Such modifications are normally considered very early in a cracking algorithm; hence such a jump is unlikely to match reality. Similarly, *qwerty* is tagged as weak by Yahoo!, while *qwerty1* jumps to strong; *password0* as weak and *password0+*

Password Multi-Checker

password$1

Services	Strength scores	
Apple	Moderate	2/3
Dropbox	Very Weak	1/5
Drupal	Strong	4/4
eBay	Medium	4/5
FedEx	Very Weak	1/5
Google	Fair	3/5
Microsoft (v1)	Strong	3/4
Microsoft (v2)	Medium	2/4
Microsoft (v3)	Medium	2/4
PayPal	Weak	2/4
Skype	Poor	1/3
Twitter	Perfect	6/6
Yahoo!	Very Strong	4/4

Fig. 6. Password multi-checker output for *password$1*. A strength score of "2/3" denotes the relative position (2) in a given strength scale (3).

as strong by Google. Finally, as expected, a random password *+^v16#5{](* is rated as strong by most checkers (or at least medium by Microsoft v3 and eBay); surprisingly, FedEx considers it as very-weak. These problems can be mostly attributed to the stringent minimum requirements.

One possible consequence of these heterogeneous behaviors is the confusion of users with regard to the security of their passwords. When opposite strength values are given for the same password by different services, users may not understand the reason behind it, which may decrease their trust and willingness to comply with password policies. It may also encourage them to search for easy tricks to bypass stringent restrictions rather than reconsidering their password. Also, permissive meters may drive users to falsely assume their weak password as strong, and provide only a false sense of security (cf. Heijningen [13]), which in turn may encourage users to reuse such weak passwords for other more sensitive accounts. However, the real effects of wrong/incoherent meter outcomes on users may be demonstrated only by a large-scale user study.

Web-based password multi-checker. We combine the functionalities of all tested checkers in a web-based password multichecker that computes the strength of a given password against all checkers in realtime; see Fig. 6 for a screenshot of the tool. Inconsistencies (if exist) in password rating become instantly evident from the output of a given password. This tool can also be used to choose a password that is considered strong by all meters, increasing the chances of that password being effectively strong. Finding such passwords may also help users select few strong passwords to reuse, but note that although password reuse is a common practice, it is generally considered bad for security. The tool is available at: https://madiba.encs.concordia.ca/software/passwordchecker/index.php.

B. Targeted dictionary for FedEx

From the results in Fig. 5, it may appear that FedEx's meter is nearly optimal in detecting the most common passwords.

However, such results may be misleading as our dictionaries are selected to uncover only the general weaknesses of widely-used meters. A more targeted dictionary may reveal specific weaknesses of a given meter. To evaluate this hypothesis, we built a combined dictionary using words from Top500, JtR and Cfkr. We then applied slightly more refined mangling rules that are consistent with [32], [13], namely: (a) capitalize and append a digit and a symbol; (b) capitalize and append a symbol and a digit; (c) capitalize and append a symbol and two digits; and (d) capitalize, append a symbol and a digit and prefix with a digit. We then removed the passwords below 8 characters, resulting in a dictionary of 121,792 words (only 4 symbols and 4 digits are covered for simplicity). 60.9% of this dictionary is now very-strong, 9.0% is strong, 29.7% is medium, and the rest is very-weak (due to repetitions of the same character). Thus, the FedEx checker is particularly prone to qualify easy-to-crack mangled passwords as of decent strength, as it cannot detect the core word anymore. One evident weak point in the algorithm is the way a password is searched in the dictionary, which checks for equality with the entire password, rather than searching for dictionary words as a substring of the password.

C. Comparison

In Section V, 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 I), we simplified the scales for our comparison. Fig. 7 and 8 show the percentage of the dictionaries that are tagged with an *above-average* score by the different web services, sorted by decreasing cumulative percentages. To be conservative, we choose to count only the scores labeled at least "Good", "Strong" or "Perfect". Clearly, such scores should not be given to most of our test set (possible exceptions could be the complex passwords from leaked dictionaries).

In reality, Google, Drupal and Yahoo! assign decent scores to passwords from our base dictionaries; see Fig. 7. Significant percentages of Top500, Cfkr and JtR are qualified as good by Drupal and Yahoo!, which are about 1.6%, 15.5%, and 12% respectively for both checkers. Also, roughly 40% of RY5 and 45% of phpBB passwords are tagged as good by both Drupal and Yahoo!. This similarity in the results possibly originates from the simple design of their meters, which perform similar checks. Google assigns good scores to 71.2% of C&A, 28.6% of RY5 and 44.5% of phpBB. Other checkers mostly categorize our base dictionaries as weak.

The mangled and leet dictionaries trigger more distinctive behaviors. Drupal, Yahoo! and Google still provide high scores with a minimum of 63% given to RY5+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, Dropbox, Skype, 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).

D. International characters

We have not tested passwords with international characters due to the lack of dictionaries with a considerable number of such passwords. International characters are also usually not properly handled for web passwords (see e.g., Bonneau and Xu [4]). We briefly discuss how such characters are taken into account for strength calculation by different checkers.

International characters, when allowed, are generally considered as part of the symbols charset (or "others" by Microsoft v2). However, this charset is limited to specific symbols for Apple, eBay, FedEx, Google, Microsoft, PayPal, Skype, Twitter, and Yahoo! (that is, all except Dropbox and Drupal). 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 [4]. Passwords are also correctly escaped with percentage as part of the URI encoding by Apple, eBay, Google and Skype. However, PayPal shows an interesting behavior in our tests: it sends the HTTP Content-Type header `application/x-www-form-urlencoded; charset=UTF-8`, meaning that a properly URI encoded string is expected as POST data. However, no encoding is performed and characters that require escaping are sent in a raw format, e.g., `search_str=myspace1&PQne)!(4`, where the password is *myspace1&PQne)!(4*. The character `&` interferes with the parsing of `search_str` argument and the remaining of the password (`PQne)!(4`) is dropped from the check. Then, because *myspace1* is blacklisted, the entire password is blacklisted. However, removing the ampersand makes the entire password being evaluated, which in turn is not blacklisted, and even tagged as strong. UTF-8 characters are also sent in a raw format (the proper Content-Type should be `multipart/formdata` in this case [4]). To get the same output as the PayPal website, we carefully implemented this buggy behavior in our tests and multichecker tool.

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

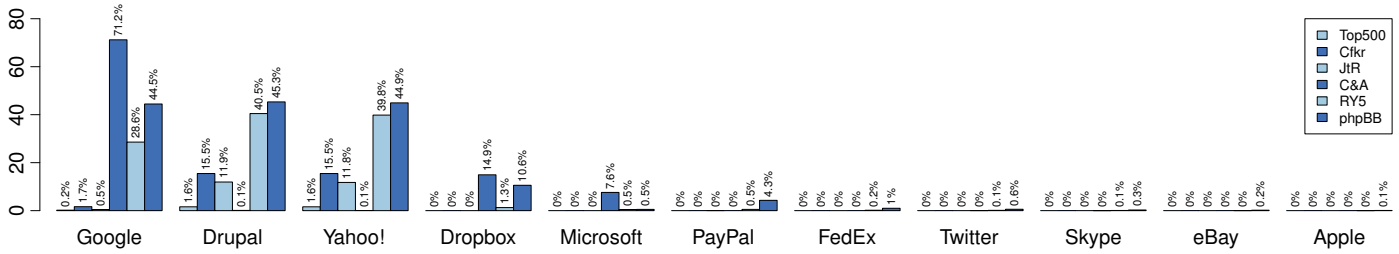

Fig. 7. Comparison between services assigning decent scores to our base dictionaries (Microsoft is represented by its latest checker)

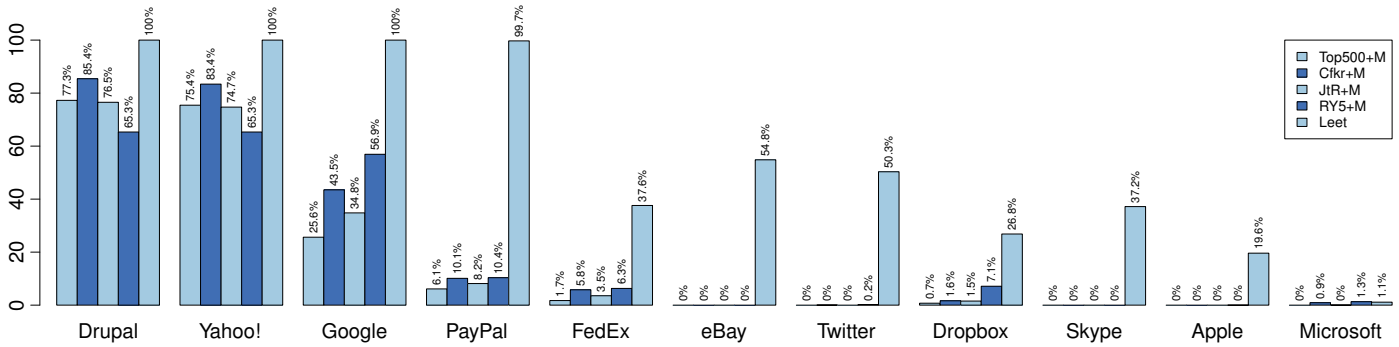

Fig. 8. Comparison between services assigning decent scores to our mangled and leet dictionaries (Microsoft is represented by its latest checker)

F. Stringency bypass

Users may adopt simple mangling rules to bypass password requirements and improve their password strength score [27]. However, all checkers (except Apple and Dropbox), 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. 8.

G. Google checker hypothesis

Based on our test results, it is difficult to model Google's server-side checker. We explained discrepancies in Section V-B by providing strange examples, where for a given charset structure, a leading digit not only has an importance for the charset diversity, but the value of the digit itself also appears to be significant. We showed that *testtest0* is strong, while *testtest1* is only fair. We also noticed that strength scores fluctuate in time for no apparent reason. We speculate that Google's scoring algorithm might be based on a dynamic mechanism (cf. password popularity [25]), which may explain the change of strength for some passwords with time, and why the particular value of digits is of importance in a password.

H. Password policies

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

VII. Discussion

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

(a) Implications of online vs. offline attacks. The strength of a password should represent the amount of effort an adversary must employ to break the password (see e.g., [8]). When mapping entropy or guessability to a strength score, in effect, we estimate the time needed by an attacker for guessing a particular password. However, such an estimate will be very different depending on the type of password guessing attack considered, i.e., online vs. offline (see e.g., [16]). Few passwords per second may be guessed in an online attack (before being rate-limited and/or facing CAPTCHA challenges), while billions per second may be tested in an offline attack. While it may be reasonable for web services to consider only online attacks, history proved us time and

again that (hashed) password databases leak more frequently than we may anticipate, and millions of hashed passwords may be subjected to offline cracking. Thus, the large difference in efficiency between online and offline attacks complicates assigning password strengths. Web services should at least explain to users what the assigned strength of a given password may mean.

(b) Password leaks. Real-world password leaks[9] complicate the design of a reliable meter. A strong leaked password used by a significant proportion of users, will most likely be integrated into a general attack dictionary, and thus should be disallowed, or at least be assigned a lower score. For checker designers, tracking and incorporating all leaked password databases may be infeasible in practice. Users also may be confused to discover that their *perfect* password not even being allowed after a while. Thus, considering password leaks, and the increasing number of users creating new passwords, adaptive and time-variant checkers (e.g., [8], [15]) may provide more reliable strength outcome.

(c) Passphrases. Passphrases can offer decent entropy while being easier to memorize for users [19], [26], as long as they do not follow simple grammatical structures [24]. However, most checkers, except Dropbox (and Twitter to some extent), would rank passphrases at the lower-end in the strength scale. Checkers basing their score on charset complexity (Drupal, FedEx, Microsoft v1, Yahoo!, Apple, and PayPal) always assign low scores to passphrases. Other checkers (Apple, Microsoft v2 and v3) do not grant their best scores unless more charsets are used.

(d) Relative performance of our dictionaries. As discussed in Section IV-C and IV-D, the mangling rules and leet transformations employed in our tests are not carefully optimized or targeted as they would be by a determined attacker (cf. [1], [2]). A better designed targeted dictionary may prove significantly more effective against the meters, as evident from our test against FedEx (see Section VI-B). We believe that even if our base/mangled dictionaries are disallowed or assigned reduced scores, users will try to creatively bypass such restrictions with other simple patterns (cf. [18], [32]). In this regard, our analysis is an underestimate as for how real user-chosen passwords are evaluated by meters.

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

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

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 services that wish to implement a meter, should not start the development of yet another algorithm, but rather consider using or extending *zxcvbn* [35].

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. Meters can easily improve their detection of weak passwords by leveraging known cracking techniques and common password dictionaries. In contrast, labeling user-chosen passwords as perfect may often lead to errors (seemingly random passwords, e.g., *qeadzcwrsfxv1331* or *Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl fhtagn1* may not be as strong as they may appear [1], [2]).

VIII. Related Work

Below we discuss related work mostly from password meter and password cracking areas.

In a recent user study, Ur et al. [29] tested the effects of 14 visually-different password meters on user-chosen password creation. They found that meters indeed positively influence user behavior and lead to better password quality in general. 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.

In another user study, Egelman et al. [11] also reported positive influence of password meters. This study also considered context-dependent variations in the effectiveness of meters, and found that passwords for unimportant accounts are not much influenced by the presence of a meter. 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 [12] 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-

[9]For an example collection of leaked password databases, see: http://thepasswordproject.com/leaked_password_lists_and_dictionaries

deployed password meters against millions of passwords, and uncovering several previously unknown weaknesses.

Castelluccia et al. [8] 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 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).

Kelley et al. [17] studied password composition rules by analyzing 12,000 user-chosen passwords under seven different policies, and reported that imposing only a longer length requirement, yields better entropy than enforcing charset complexity on smaller passwords. Memorability and usability effects of composition policies are discussed in a separate study [18]. Studied users tend to end a password with several digits and a possible symbol, correlating with the observations from past studies (e.g., Burr et al. [7] and Weir [32]). Such known user behaviors validate the use of mangling rules in password cracking.

In its community-enhanced version, John the Ripper [22] 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" [22]. In fact, this mode is an implementation of a 2005 proposal from Narayanan and Shmatikov [21], 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.

Dürmuth et al. [10] extend the work of Narayanan and Shmatikov [21] to improve cracking performance. They also explore the inclusion of users' personal information (e.g., username, birthday, list of friends, education, work, siblings, first name, last name, and location) in the cracking algorithm, and found that such personal attributes further enhanced the cracker's performance.

Weir et al. [33] propose an algorithm for extracting password structures from a given training dictionary (e.g., of leaked passwords), so as to infer which ones are the most common. Base structures are in the form of a sequence of charsets with associated lengths, e.g., L_3D_1 for 3 letters followed by 1 digit. In their proposed cracking algorithm, letter-only parts are filled by searching for a word of the required size in a dictionary, and other sets are filled in a decreasing order of probability based on the derived statistics from training dictionaries.

Concurrent to our work, Veras et al. [30] leverage Natural Language Processing (NLP) algorithms to analyze semantic patterns in leaked passwords. They found that most passwords in the RockYou dataset are semantically meaningful, containing terminologies related to love, sex, profanity, animals, alcohol and money. Their semantic-aware cracking technique shows significantly better results than existing techniques, and may also be used as a password-strength checker.

IX. 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. [14]). Users may be forced to choose stronger passwords by imposing stringent policies, at the risk of user resentment. An apparent better approach is to provide appropriate feedback to users on the quality of their chosen passwords, with the hope that such feedback will influence choosing a better password, *willingly*. For this approach, password-strength meters play a key role in providing feedback and should do so in a consistent manner to avoid possible user confusion. In our large-scale empirical analysis, it is evident that the commonly-used meters are highly inconsistent, fail to provide coherent feedback on user choices, and sometimes provide strength measurements that are blatantly misleading.

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

Acknowledgment

We are grateful to anonymous NDSS2014 reviewers and Jeremy Clark for their insightful suggestions and advice. We also thank the members of Concordia's Computer Security Lab for their enthusiastic discussion on this topic. The second author is supported in part by an NSERC Discovery Grant and Concordia University Start-up Program.

References

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

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

[3] M. Bishop and D. Klein, "Improving system security via proactive password checking," *Computers & Security*, vol. 14, no. 3, pp. 233–249, May/June 1995.

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

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

[6] ——, "10,000 top passwords," June 2011, https://xato.net/passwords/more-top-worst-passwords/.

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

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

[9] X. de Carné de Carnavalet and M. Mannan, "From very weak to very strong: Analyzing password-strength meters," Concordia University, Tech. Rep. 978049, Dec. 2013, http://spectrum.library.concordia.ca/978049/.

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

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

[12] S. Furnell, "Assessing password guidance and enforcement on leading websites," *Computer Fraud & Security*, vol. 2011, no. 12, pp. 10–18, Dec. 2011.

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

[14] C. Herley and P. Van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Security & Privacy*, vol. 10, no. 1, pp. 28–36, 2012.

[15] S. Houshmand Yazdi, "Analyzing password strength and efficient password cracking," Master's thesis, Florida State University, June 2011.

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

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

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

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

[20] R. Morris and K. Thompson, "Password security: A case history," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979.

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

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

[23] oxid.it, "Cain & Abel," http://www.oxid.it/cain.html.

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

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

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

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

[28] E. H. Spafford, "OPUS: Preventing weak password choices," *Computers & Security*, vol. 11, no. 3, pp. 273–278, May 1992.

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

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

[31] W3Techs.com, "Market share trends for content management systems for websites," online report. http://w3techs.com/technologies/history_overview/content_management.

[32] C. M. Weir, "Using probabilistic techniques to aid in password cracking attacks," Ph.D. dissertation, Florida State University, Mar. 2010.

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

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

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

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

Appendix A
Evaluation of Additional Meters

(a) Drupal. Drupal is an open-source framework for building content management systems (CMS). It is the third mostly used CMS,[10] behind Joomla and WordPress [31] (as of Dec. 6, 2013). Drupal version 7.x uses a simple client-side checker based on a decreasing scoring system (Joomla and WordPress currently do not use a checker). Fig. 9 summarizes our results.

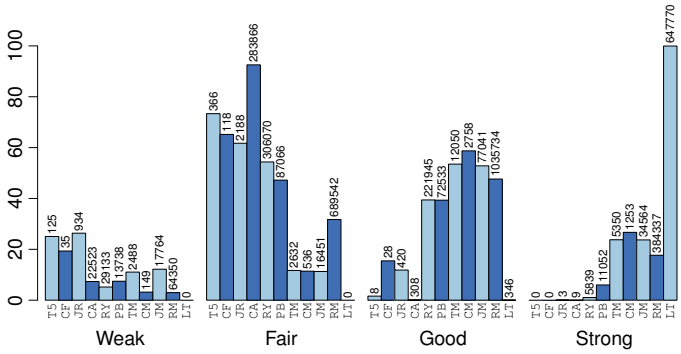

Fig. 9. Drupal checker password strength distribution

(b) Microsoft. Microsoft's password checker is available as a separate webpage for users to evaluate any password. The JavaScript source of the client-side checker shows a commented old version along with the current version. During our study, the checker has changed to a newer algorithm. We

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

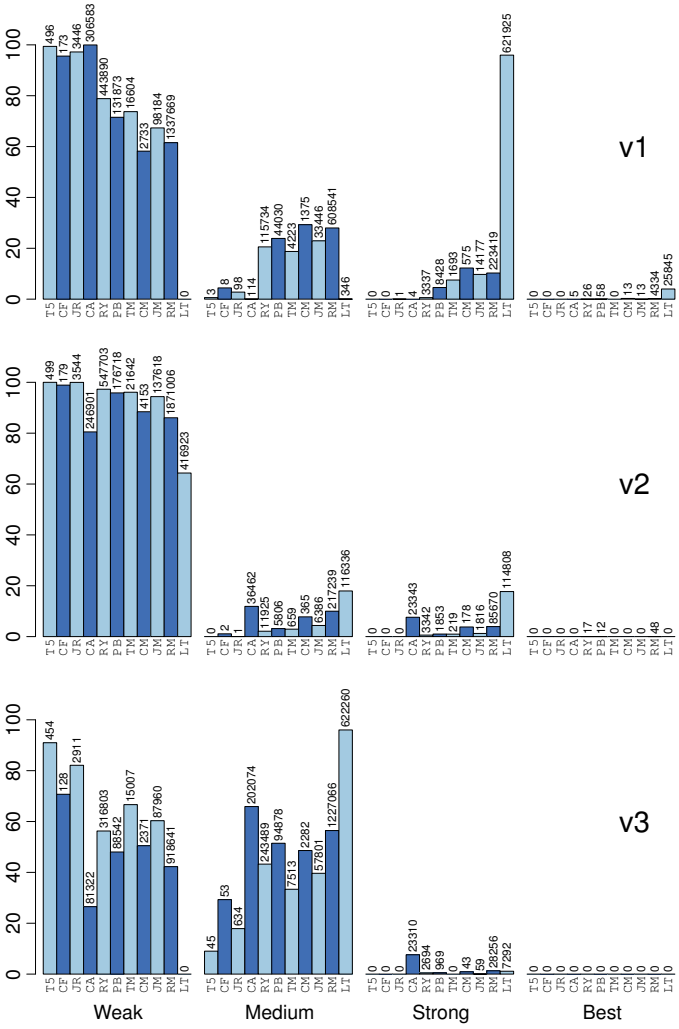

Fig. 10. Microsoft checkers (v1, v2 and v3) password strength distribution

evaluate all three versions here. All algorithms are based on predefined strict rules as explained below. Fig. 10 summarizes our results for the three versions of Microsoft's checker.

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

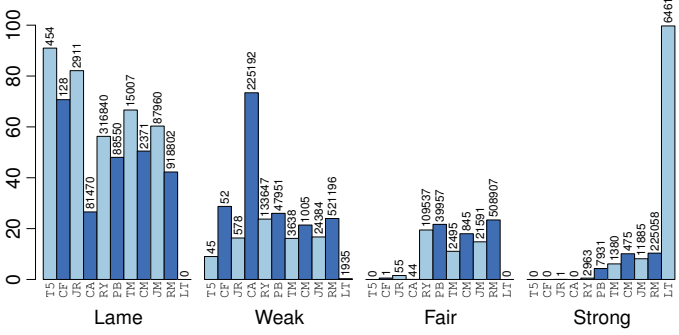

Fig. 11. PayPal checker password strength distribution

(d) Skype. Skype is now a Microsoft-owned VoIP service provider with more than half a billion users. Skype's password checker relies solely on a server-side validator, independent of the Microsoft checker. Like Google, no user information are sent to the server during password evaluation, allowing a user to register with a password close to her username.

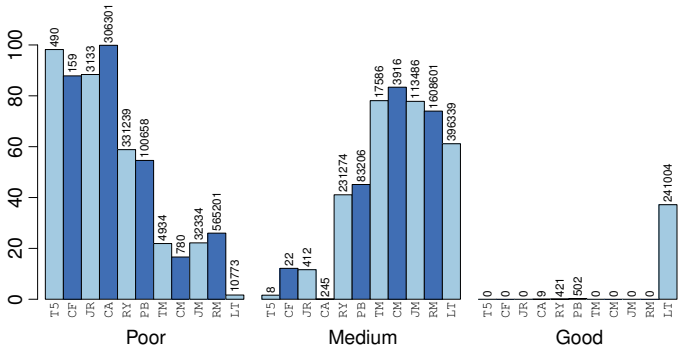

Fig. 12. Skype checker password strength distribution

(e) Twitter. Twitter is a microblogging service gathering over half a billion users. It has the most diverse strength scale with 6 categories and is also fully client-side.

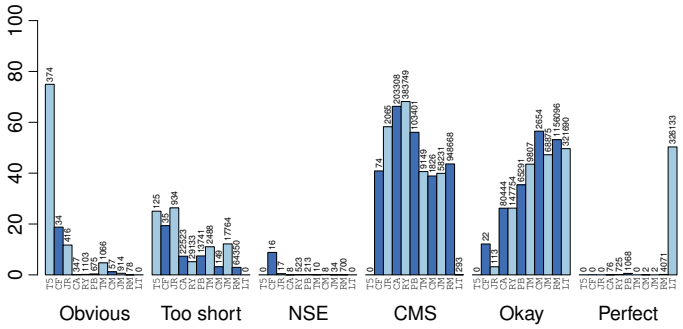

Fig. 13. Twitter checker password strength distribution

(f) Yahoo!. Yahoo! relies on a client-side checker with 4 main strength categories. The additional invalid category is used for passwords matching the provided user information, or if the password is *password*. We omit this category in Fig. 14, which summarizes our results for Yahoo!.

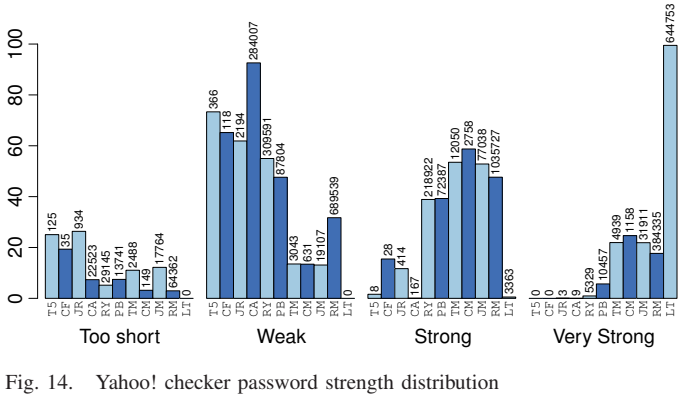

Fig. 14. Yahoo! checker password strength distribution