StopGuessing: Using Guessed Passwords to Thwart Online Guessing

Yuan Tian
Carnegie Mellon University Pittsburgh, PA
yuan.tian@sv.cmu.edu

Cormac Herley and Stuart Schechter
Microsoft Research, Redmond, WA, USA
cormac@microsoft.com, stuart.schechter@microsoft.com

Abstract

Practitioners who seek to defend password-protected resources from online guessing attacks will find a shortage of tooling and techniques to help them. Little research suggests anything beyond blocking or throttling traffic from IP addresses sending suspicious traffic; counting failed authentication requests, or some variant, is often the sole feature used to determine suspicion. In this paper we show that several other features can greatly help distinguishing benign and attack traffic. First, we increase the penalties for clients responsible for fail events involving passwords frequently-guessed by attackers. Second, we reduce the threshold (and thus protect better) for accounts with weak passwords. Third, we detect, and are more forgiving of, login failures caused by users mistyping their passwords. Most importantly, we achieve all of these goals *without* needing any marker that indicates weak accounts, changing the format in which passwords are stored (i.e. we do not store passwords plaintext or in any recoverable form), or storing any information that might be harmful if leaked.

We present an open-source implementation of this system and a guessing-attack simulator and demonstrate how researchers and practitioners can use it to examine different attack and defense strategies. We examine the performance under several experimental simulations. While particular to the conditions simulated our experiments suggest if practitioners can only implement one of the techniques described in this paper, they should protect users who rely on frequently-guessed passwords more aggressively than others.

I. Introduction

In *online* password guessing, attackers send login requests to a service to test if a guessed password is correct. Limited to a fixed number of guesses, attackers compromise the maximum number of accounts by using the most popular password as a guess against all accounts, then using the second most popular password, and so on. While attackers may not know exactly which passwords are most popular at a particular service, they can estimate using data from past breaches of other services.

The problem is significant. GitHub [18], Twitter [6], and Apple [38] have revealed that user passwords had been compromised in online guessing attacks. The information security arm of the UK's GCHQ recommends [17] "defending against automated guessing attacks by either using account lockout, throttling, or protective monitoring." Yet, while large websites presumably implement some techniques to block online password guessing, none detail precisely how. While there is a rich literature on strengthening user-chosen passwords (mostly to protect against offline attacks), there is a scarcity of research on tools to detect and block online guessing attacks. This leaves operators of password-protected online services to develop their proprietary defenses on their own.

The techniques most often referenced are a "three strikes" type policy for locking individual accounts, or some strategy to block traffic from an IP address, e.g., if it exceeds a threshold number of failed attempts. Contrary to the popular view, "three strikes" policies appear very uncommon at large providers [21]. This may be due the Denial of Service (DoS) attack that they open, or the fact that poorly-configured clients can generate many repeat fails with a cached incorrect credential (see Section VI). IP address blocking also potentially suffers from this difficulty: repeated fails from a benign client can easily block a legitimate users's access. Fixed thresholds will also imply very different false positive to false negative tradeoffs as the ratio of attack to benign traffic rises or falls.

A further reason for reluctance to enforce "three strikes" may be that the obvious measure of penalizing only *distinct* failed guesses (thus not locking out clients that resend cached credentials) is not entirely trivial in a distributed environment. That is, if authentication requests are assigned in a round-robin fashion among several authentication servers we can recognize whether fails are distinct or not only if we store some information on failed requests. This information must be accessible to all servers, and (since successive requests for the same username will not in general be handled by the same server) cannot simply reside in short-term memory. Web services that have user populations that run to hundreds of millions may have thousands of servers to handle the load.

Effectively IP address blocking is a binary classifier making a benign/malicious decision. If the only input feature is the number of failed attempts then it represents a very crude defensive weapon. The tradeoff between false negatives (attacker gains access) and false positives (legitimate user is blocked) will obviously depend on how we threshold. It's difficult to imagine a threshold that blocks all attackers while never inconveniencing ordinary users, especially since numerous fail events may also be generated by legitimate users. In this paper we show how additional features can greatly assist the task of deciding whether an authentication request should be blocked. We exploit the fact that optimal failed-guess traffic from an attacker necessarily looks very different from failed attempts from legitimate users. We sketch the insight behind these features next, and then tackle some of the difficulties in a naive implementation.

First, to maximize per-guess success probability an attacker should concentrate on the most common passwords. This means that fail events with common passwords are strong indicators that an attacker is involved. Second, a user account with a weak password is much more vulnerable than one with a strong password; it would make sense to protect those accounts more. Third, users mis-remember and mis-type their passwords often; we expect user-generated typos to be close in typing-distance to the actual password. It would make sense to penalize fail events that are close to the actual password much less than, e.g., fail events with common passwords.

These are all rather obvious examples of features that might improve a binary block/no-block decision. Equally obvious, however, is that there are reasons that they are not currently used: each of them appears to require information, which cannot be stored openly without great risk (in the event of an attacker gaining access to the server). We cannot store all failed guesses as we would reveal information (e.g. typos that are close to the actual password). We certainly cannot store any information which reveals which accounts have weaker passwords (since this would give an attacker a guide as to where to invest effort). Finally, since passwords are stored as salted-hashes we cannot determine whether a failed attempt was close (and thus likely to be a typo) since all we can tell by comparing hashes is whether the guess matches the actual password or not. Thus, we have a frustrating situation. We can identify features which appear very valuable in distinguishing malicious from benign fail-event traffic, yet we cannot use those features since they appear to require information that cannot be safely stored without introducing new risk.

Our contributions in this paper is to offer techniques to overcome these difficulties. We show how each of the features identified can be used without introducing new risk. First, we need to be able to identify frequently-submitted incorrect passwords without storing information that might compromise the incorrect passwords submitted when users make benign errors. We do this using a probabilistic data structure known as a *binomial ladder filter* [37] that identifies "heavy hitters" without revealing information on less-frequent failed guesses. Second, we show how to enhance protection for accounts with weak passwords without storing any information about the password strength. We do this by observing that block decisions only need to be made when the submitted password is correct. We calculate a strength-dependent threshold speculatively assuming the submitted password is correct. This yields a low threshold when the submitted password is correct and weak, a high threshold when it is correct and strong, and is irrelevant when the submitted password is incorrect (since login fails anyway). Third, we introduce a scheme to identify login attempts that fail due to typos of passwords. We do this *without* storing users' passwords in plaintext or storing any other information that might put users' passwords at greater risk than they will already face should the account database be compromised. This is accomplished by making the key observation that all decisions on whether a given fail is a typo can be deferred until the correct password is presented in an authentication request (and thus is available without any additional risk). Since block decisions only have to be made when the correct password is submitted we lose nothing by suspending judgement on whether fail events are typos.

We have designed, implemented, and tested a system motivated by these observations to attempt to identify guesses before attackers are granted access to accounts and in so doing prevent them from learning when their guesses are correct. Our approach which is open source and publicly available at https://github.com/Microsoft/StopGuessing, adheres to Kerckhoffs's principle that the security of a system should not rely on the secrecy of the underlying algorithms—we assume that attackers know every detail of our system's design.

Since different online services have different user bases, employ different security policies, implement different password-composition rules, and experience different attack traffic, the optimal blocking strategy for one deployment may be unacceptable for another. For example, at a small site there might be no overlap in the IP address pools used by attackers and legitimate users; at a web-scale provider there will likely be significant overlap and much traffic may be aggregated and arrive via proxies. No single study can establish a universal "best" configuration. Thus, our final contribution is a simulator that allows services to test different configuration parameters under various usage and attack scenarios. We use this simulation framework to illustrate the potentially dramatic differences in the effectiveness of different attack strategies and the effectiveness of different detection features—including those that rely on examining incorrect passwords.

We use these simulations to illustrate the effectiveness of optimal attacks, i.e. those that focus on guessing the most popular passwords. We exploit the fact that authentication failures generated by efficient attacks are much more likely to contain a frequently-guessed passwords than legitimate user errors. Thus, while an attacker can take evasive action, this necessarily involves sending far-from-optimal attack traffic. Indeed, we see in simulation that attacks that avoid detection by underutilizing common passwords are even less effective than those that do not attempt to evade detection. Thus, informing IP blocking with information about frequently-guessed passwords achieves the desired goal of decreasing attacker success rates even when attackers have a detailed understanding of the schemes and take evasive action.

We also discover that, for our simulated scenarios, password-based blocking complements the passive defense of preventing users from choosing common passwords—yielding the best results when these two techniques are combined. While these conclusions are based on observations under a wide range of simulation settings (e.g. assumptions about how much IP address pools used by attackers and users overlap) we stress that they need not be universal. Our simulator is also open-source, so readers are invited to stress test under different scenarios.

II. Blocking algorithm

We present a number of simple techniques to improve IP-based blocking of password-guessing attacks which, when combined, may appear complex. We begin with today's state of the art, used by a number of tools for protecting remote terminals from guessing attacks [16], [11], [42], which blocks attempts from those IP addresses that have exceeded a threshold T of recent login failures. This existing rule can be summarized with the simple blocking criterion:

$$I > T,$$

where I is a count of login attempts with invalid credentials.

While each technique we will introduce is in itself relatively simple, the result of combining them all to create a new blocking criterion may appear complex. To assist the reader, we start with the equation above and highlight the changes needed to accommodate each new technique.

A. *Penalize frequently-guessed passwords*

Blocking IP addresses that issue too many incorrect guesses will limit the number of guesses an attacker can issue. Facing a fixed budget of guesses, an attacker can maximize the number of accounts compromised by guessing only those passwords they expect to be most popular—choosing a small set of passwords to guess. On the other hand, when legitimate users make mistakes they rarely submit passwords that are among those frequently-guessed in the past..

We do this by examining each failure to estimate how frequently the provided password $\mathbf{g}$ has occurred among prior failures (how often it is being used to guess). We introduce a penalty function, $\phi(\mathbf{g}_i)$, which increases when the password submitted in the failed attempt is among the most frequently occurring incorrect passwords submitted in the past. In place of an aggregate failure counter, I, we use the summation of penalties for each failed login attempt $\sum_i \phi(\mathbf{g}_i)$.

$$\sum_i \phi(\mathbf{g}_i) > T$$

Next we will explain how to identify frequently-occurring incorrect passwords, which typically result from attacks, while minimizing the risk of storing information about less-frequently occuring incorrect passwords, which may result from user errors.

Identifying frequently-guess passwords: Storing even incorrect password guesses has risk, as these include nearly-correct mistyped passwords as well as correct passwords for users who provided an incorrect account name. An attacker who breached the server might easily guess the correct password by observing typos that are very close in edit distance, for example. The risk is not uniformly distributed across all fail events: the passwords guessed most often by attackers will be the most common failures, and there is no risk in storing these. Thus, we would like to store accurate information on the "heavy hitters" while storing no information at all on the very infrequent fail events. A number of data structures allow calculation and storage of "heavy hitters" [30]. We use a binomial ladder frequency filter [37], since it gives strong privacy guarantees. This is a privacy-preserving structure that allows us to reliably identify frequently-occurring elements without storing information that would allow identification of infrequent ones. An example of the idealized performance of such a filter is shown in Figure 1. Frequent elements (e.g. those more common than 1 in 10^6 in Figure 1) are detected, while infrequent elements (e.g. those less common than 1 in $5 \cdot 10^7$ in Figure 1) are rejected with very high probability. One can think of the filter as an function with elements as input and frequencies as output; it returns an accurate estimate when the frequency is higher than one threshold and no information when it is lower than a second threshold.

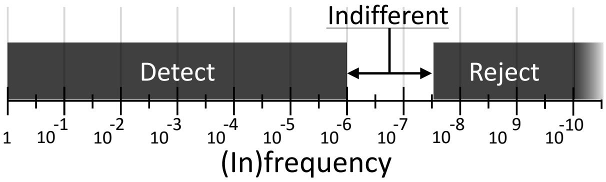


Fig. 1: A frequency filter is used to examine a sequence of values to identify frequently-occurring values while filtering out infrequent ones. The filter should detect values that arrive at a rate that exceeds the detection threshold (one in a million, or 10^{-6}, in this illustration) and should reject values that arrive at a rate below the rejection threshold (one in fifty million, or $2 \cdot 10^{-8}$). It may detect or reject values that arrive at rates between the two thresholds—a segment of the frequency range called the *indifference* region. [This Figure is copied with permission from [37].]

The reason for using a binomial ladder, as opposed to maintaining deterministic counts of each hash or using a more sensitive probabilistic data structure (*e.g.,* a count-min sketch) is the privacy guarantees it provides [37]. Maintaining the filter is simple. Like a Bloom filter, a binomial ladder filter pairs an array of N bits with a family of H hash functions used to index into the array. When an element is added to the filter, one of the associated bits in the array is set to one, while a bit from the array not associated with the element is cleared to zero. Elements are treated as frequent if the number of one bits they are associated with exceeds a threshold T.

B. *Defend weakest accounts more*

Accounts that employ the most frequently-guessed passwords are the most likely to be compromised by guessing attacks, and so the threshold at which an IP address should be blocked should be significantly lower for these accounts. Conversely, accounts with rarely-guessed passwords are the least vulnerable, and so may allow higher (less stringent) blocking thresholds to reduce false positives; there is no need to be particularly aggressive when the account is protected by a password strong enough to withstand online guessing.

For our algorithm, we denote the correct password for an account by $\mathbf{c}$. We replace the fixed threshold T with a threshold function $T(\mathbf{c})$ which takes as input this correct password. The threshold it returns should decrease with the expected number of attempts needed to guess the password:

$$\sum_i \phi(\mathbf{g}_i) > T(\mathbf{c}).$$

Since our blocking algorithm will only influence the outcome of a login attempt when the entered password is also the correct password for the account (i.e. $\mathbf{g} = \mathbf{c}$), we can substitute the former for the latter:

$$\sum_i \phi(\mathbf{g}_i) > T(\mathbf{g}_i). \tag{1}$$

To see this, note that for each login attempt the password provided by the client is either (a) correct or (b) incorrect. We don't need to make any decision about blocking in case (b). Thus the calculation above, speculatively assuming that $\mathbf{g} = \mathbf{c}$, will be correct in case (a) and irrelevant in case (b) since login fails anyway. This allows us to give stronger protection (i.e., lower threshold) to user accounts with weaker passwords without storing any information whatever that distinguishes the weak accounts from the strong.

While one could implement a threshold function $T(\mathbf{g}_i)$ by tracking the frequency of users' actual passwords, this presumes an attacker who has perfect knowledge of the frequency of all user passwords. The frequencies used to compute $T(\mathbf{g}_i)$ using real users' passwords would be helpful to attackers if compromised. Instead, we will compute $T(\mathbf{g}_i)$ using our estimate of the most frequently guessed *incorrect* passwords observed in previous login attempts. This frequency more closely reflects the actual likelihood that a password is one attackers are likely to guess rather than what they should be guessing if they already knew the distribution of users' passwords. Further, if the most frequently guessed incorrect passwords are those being guessed by attackers, this information reflects what attackers already know (what they're *already* guessing) and would be of little value if compromised.

C. Penalize typos less than other failures

Mistyping a password will often result in a submitted password $\mathbf{g}$ that is a short edit distance away from the correct password $\mathbf{c}$. While a few popular passwords are a short edit distance away from each other, the probability that an attacker's incorrect guess will be a short edit distance from the correct password is still quite small.

We incorporate this observation by adjusting the penalty for submitting an incorrect password by a function, $\beta(\mathbf{g}_i, \mathbf{c})$, that calculates the edit distance between the password given in the login attempt ($\mathbf{g}_i$) and the correct password ($\mathbf{c}$), and returns a penalty that is lower if the edit distance is small:

$$\sum_i \beta(\mathbf{g}_i, \mathbf{c}) \cdot \phi(\mathbf{g}_i) > T(\mathbf{g}_i)$$

Computing the edit distance requires the plaintext of both the incorrect and the correct password. Storing a user's password in a format that allows for plaintext recovery would put the password at greater risk, as could storing an incorrect password that closely resembled the correct password. We describe how we can identify login attempts that failed as the result of a typo *without* putting users' passwords at greater risk in Section III.

D. Ignore repeat account/password pairs

Attackers do not benefit from attempting to login to an account with a password they have already learned to be incorrect. There is thus little need to penalize login attempts that repeat the same user identifier and password reported as incorrect in a previous attempt. Conversely, the legitimate owner of an account may repeatedly mistype a password she thinks is correct but is not. Further, she may run client software (*e.g.,* an IMAP mail client, or a password manager) that repeatedly attempts to login with a password that has expired or was entered incorrectly in the first place. Since counting a failure with a previously-seen user identifier and password will penalize benign users, but not thwart attackers, we can simply ignore these failures.

Thus, we want to ignore failures with an account-password pair when we have already counted a failure for that pair. To achieve this, for each account we maintain a fixed-size LRU cache of recent failed password hashes. To detect pairs that include a non-existent account identifier, we also maintain a sketch (essentially an aging Bloom filter) of user identifier and password pairs.

E. Treat invalid accounts differently

Depending on the deployment environment, an incorrect account name may be stronger or weaker evidence of a guessing attack than an incorrect password. For systems with account identifiers that are effectively public (*e.g.,* systems with account IDs drawn from a dense space) attackers will rarely guess an incorrect account identifier, whereas legitimate users may mistype them. For systems with very private account identifiers (*e.g.,* email addresses for sites with a small private user base) attackers may be much more likely to provide an invalid account identifier.

To support different penalties for these different types of failures, we separate failures into two subsets: the invalid-account failures A and the invalid-password failures P. Invalid-account failures can then be assigned a penalty, represented as α, to parallel the β penalty for invalid-passwords (unlike the β function, α is a constant since we do not identify typos in account names).

$$\alpha \sum_{i \in A} \phi(\mathbf{g}_i) + \sum_{i \in P} \beta(\mathbf{g}_i, \mathbf{c}) \cdot \phi(\mathbf{g}_i) > T(\mathbf{g}_i)$$

F. Account for prior successful logins

While someone who shares or obtains a user's device might try to guess that one account's password, such attacks do not scale beyond that one user. Thus, we can assume that if a client has previously logged into an account in the past, it is not likely attempting to do so as part of a large-scale guessing attack. To detect repeat logins by the same client, we can configure web clients with unique random cookies drawn from a large space that act as client-identification keys. We will reduce the blocking penalty by κ if a login attempt contains a cookie proving that this client has authenticated successfully in the past. To do this, we set w is 1 when a client's cookie was used in a previous login and 0 otherwise.

$$\alpha \sum_{i \in A} \phi(\mathbf{g}_i) + \sum_{i \in P} \beta(\mathbf{g}_i, \mathbf{c}) \cdot \phi(\mathbf{g}_i) - \kappa \cdot w > T(\mathbf{g}_i)$$

G. Offset failures with successes

Proxies or NATs may aggregate under a single IP address a cluster of users within an organizational network, users on a home network, or users who otherwise share a common network access infrastructure. The expected number of benign failures originating from an IP address will grow linearly with the number of users who share that address.

We accommodate high-traffic IPs by offsetting the accumulated penalties with credit γ for a *carefully-selected subset* of successful login attempts, s, yielding the final criteria used by our blocking algorithm:

$$\alpha \sum_{i \in A} \phi(\mathbf{g}_i) + \sum_{i \in P} \beta(\mathbf{g}_i, \mathbf{c}) \phi(\mathbf{g}_i) - \kappa \cdot w - \gamma \cdot s > T(\mathbf{g}_i). \quad (2)$$

The set of successful login attempts must be selected carefully to limit attackers' ability to offset the penalties accumulated against their IP addresses. For example, attackers might interleave successful login attempts to accounts they control between login attempts used to guess other accounts' passwords. We allow each account to provide only one credit per IP and only a fixed number of credits per time period (*e.g.,* three per day). Services for which account creation is expensive, such as enterprise accounts for employees and (to a lesser extent) subscription services, can be generous in how they count successful login attempts and how much offset credit (γ) they provide. Services that have no defenses against mass-account-creation, and no other way to estimate which accounts are likely legitimate and which are suspicious for the purpose of determining which may provide offset credits, may need to set γ close to 0.

We have put all of the features together in a weighted-sum (2). This form is convenient for use with a logistic regression learning algorithm [15] or a log likelihood ratio test [43].

H. Additional Details

An IP that has been used in a guessing attack may become associated with a benign user, such as when an ISP moves a dynamic IP between users or when an attacker is kicked out of a machine it has compromised. As such, IP-based schemes require a means to restore the reputations of IP addresses that have previously engaged in guessing. Common methods to do so include expiring scores after a fixed period. We address this problem by applying exponential decay to blocking scores (half life: 12 hours), which requires us to store the time a score was last updated along with its value at that time.

To avoid revealing which accounts are valid, password-based authentication systems typically provide the same response message for *every* failure, regardless of whether a failure was caused by an invalid account or an incorrect password. We assume the same response is also used when blocking a suspected guess, even if the password is correct, to avoid similar information leaks. This includes the timing of the response; a different response time, for example, when the username is invalid or when the guess is close to the actual password would convey valuable information to an attacker. Bortz et al [9] show how in-use usernames can be distinguished from non-existent ones at many web services. The just-in-time typo detection that we describe in Section III involves generation of public-keys and decryption, which are obviously slower than simple hash comparisons. However, since best-practice is to use an iterated or slow hash (to deliberately slow an offline attacker), this burden is easily accommodated and a constant response time given for all requests.

Some services already evaluate the reputation of IP addresses based on their geography or their history of other malicious behaviors (*e.g.,* spamming or malware hosting). As these approaches are complementary, one can either integrate existing IP reputation scores into our blocking algorithm by adding another another term and coefficient to the blocking equation, or integrate the techniques described in this section into the service's existing mechanisms.

First, however, we must reveal how it is possible to identify mistyped passwords while minimizing the risks associated with the information required to do so.

III. Just-In-Time Typo Detection

To solve the challenge of identifying which incorrect passwords are typos *without* keeping correct passwords around for comparison, we make two observations.

First, we don't need to identify typos immediately. When an incorrect password is submitted, we will reject the login attempt regardless of whether the client submitted a typo or a completely different password. Thus, the only time we will actually need to make a decision about whether to block a login attempt or not is when the client submits the correct password for an account. In the common case, a user who submits a number of typos followed by her correct password will do so from the same IP address. Rather than identify typos immediately, we can identify a user's past typos when she next submits her correct password, and evaluate whether her login attempt should be blocked using an analysis of whether those past failures were the result of typos. In other words, we can perform just-in-time detection of typos if we can keep a record of the incorrect passwords a user has submitted in previous login attempts.

Our second observation is that storing the incorrect passwords submitted to a user's account need not increase risk, even if the user's account records become compromised in a breach. So long as decrypting the failed passwords is no easier than cracking the stored hash of the user's actual password, the additional information does not increase risk.

We associate with each user account a public/secret key pair (p, s) used to encrypt incorrect passwords submitted in failed logins. These keys are for the server's use only; users and their clients have no need or ability to access them. Rather, the public key p is stored in the user's account record and incorrect passwords are logged encrypted with this public key. The private key is itself encrypted and given the same protection as the user's (correct) password.

As is common practice to protect against offline dictionary attacks, we protect users' passwords in the authentication database by not storing them directly, but instead storing the result of an expensive (typically iterated) hash function with a salt. We call this the *expensive hash*. We diverge from common practice by not storing the raw expensive hash, but by hashing it one more time with a fast, inexpensive one-way hash to create the *stored hash*. The *stored* hash is the hash that goes into the authentication database and can be used later to verify that a provided password is correct. Because the *expensive hash* is not stored in the authentication database, we can use it to encrypt the secret key s and store it in the authentication database along with the stored hash of the user's password. Whether testing whether a candidate password is the correct password for a user account, or testing whether the candidate password can decrypt the secret key s, both tests require the same work to perform; both require a computation of the *expensive hash*.

Cracking the secret key used to store incorrect passwords is no easier than cracking the password: both require guessing the correct password and each guess requires computation of the expensive hash function.

When a client attempts to login, we use the submitted password to calculate the expensive hash and use the fast hash to test if the password was correct. If it was, we use the already-computed expensive hash to decrypt the secret key, use that key to decrypt the passwords used in recently-failed login attempts, perform our edit distance comparisons, and adjust penalties for failures that were due to typos. Once the edit distances have been calculated, we can erase our records of the failed password attempts from long-term storage and erase the plaintext password submitted by the client from short-term memory (DRAM).

If a user resets her password without providing the old password, she is assigned a new key pair and the record of her previous incorrect passwords is lost.

The additional cryptographic operations required to track typos will have little impact on performance, so long as passwords are being protected by a reasonably expensive hash function. Each login attempt with an incorrect password causes at most two cryptographic operations: one to record the incorrect password and one future operation to decrypt it so that it can be compared to the correct password. In comparison, the expensive hash function will typically be configured to iterate enough to consume between 1 and 100 milliseconds of a CPU core—lest it be too easy for attackers who might compromise the hashed passwords to crack them. Thus, the cost of public key encryption for typo detection, as with all of the costs of our algorithms, are dwarfed by the cost an executing an expensive hash function once for every login attempt.

In our implementation, the choice of hash function and the number of iterations are both configurable, allowing such options as PBKDF2 [22] and scrypt [33], and backwards compatibility with any function that a caller might choose to provide.

IV. System, Configuration and Simulator

We built an an open-source reference implementation of our system which is available at https://github.com/Microsoft/StopGuessing. The weights used in (2) are configurable in `PostSimulationAnalysis. Condition. StopGuessingCondition`.

A. Configuration

At every login attempt, successful or not, we observe a collection of features $\boldsymbol{x} = (x_0, x_1, \cdots, x_{M-1})$. The likelihood ratio test says that we decide a login attempt is malicious (mal) rather than benign (ben) when:

$$\frac{P(\text{mal} \mid \boldsymbol{x})}{P(\text{ben} \mid \boldsymbol{x})} = \frac{P(\boldsymbol{x} \mid \text{mal})}{P(\boldsymbol{x} \mid \text{ben})} \cdot \frac{P(\text{mal})}{P(\text{ben})} > 1. \tag{3}$$

Making the common assumption that the features are independent, i.e.

$$P(\boldsymbol{x} \mid \text{ben}) = \prod_{i=0}^{M-1} P(x_i \mid \text{ben})$$

(and similarly for $P(\boldsymbol{x} \mid \text{mal})$) and taking the log we can test the log likelihood instead of (3). That is, decide the request is malicious when

$$\sum_{i=0}^{M-1} \ln \left\{ \frac{P(x_i \mid \text{mal})}{P(x_i \mid \text{ben})} \right\} + \ln \left\{ \frac{P(\text{mal})}{P(\text{ben})} \right\} > 0. \tag{4}$$

Observe that the log likelihood ratio test has the same form as the blocking algorithm (2). That is by associating the x_i with the features introduced in Section II we have a way of calculating the weights α, β and so on. For example, the weight for the binary feature indicating whether a cookie is present or not would be

$$\kappa = -\ln \left\{ \frac{P(\text{cookie present} \mid \text{mal})}{P(\text{cookie present} \mid \text{ben})} \right\}.$$

Thus, the weights for the blocking algorithm depend on the ratio of how probable an observation is attack traffic to how probable it is in benign traffic. The test also depends on $P(\text{mal})/P(\text{ben})$, the ratio of attack to benign traffic.

This should make clear that no fixed set of weights will be appropriate in all settings. E.g., the rates at which any observation occurs among the benign and malicious requests probably varies from service to service. Usernames are more-or-less public at sites like `facebook`, `twitter` etc, but not at banks: so malicious fails using non-existent usernames are probably far more common at the latter. The rate at which typos occur in the password probably depends on the fraction of users who access using an app that caches the password, as opposed to typing it at a web-page. Thus, even if we measured how common a certain observation is at one site, we cannot assume this value would also apply to others. Mostly obviously, the amount and nature of attack traffic that a site receives likely depends on the nature of the site, and might fluctuate with time. Sites with the largest populations probably receive attack traffic that is the aggregation of traffic from many different attackers; smaller sites might receive attack traffic from a single (or even at times no) attackers.

The importance of estimating these weights on a per-service basis is easily illustrated. Suppose that at one service 50% of the traffic is malicious, while it is only 10% at another. If we used the same fixed weights at both then we see, from (3), that for a given false negative rate one site will have a false positive rate 5× higher than the other. Rather than fix weights which are unlikely to match the needs of any service we provide a configurable simulator, which allows exploration in any desired settings.

B. Simulator

We built a simulator to allow evaluation of features introduced in Section II. Simulations can be configured to match the profile of the sire. Like the rest of our system, our simulator is open source and publicly accessible. Most of the parameters of interest are set in `Simulator.ExperimentalConfiguration`. This allows variation of parameters such as the fraction of traffic that comes from attackers, the amount of overlap between attacker and benign IP address pools, the rate at which users forget and commit typos, etc.

C. Simulating legitimate users

The simulator begins by creating accounts for legitimate users, and so the first parameter required for simulation is the number of user accounts to provision. We assign each account's password by selecting at random from a weighted distribution of passwords provided to the simulator. By default, we provided the simulator with the large publicly-available distribution of plaintext passwords that was made public as the result of a breach of the RockYou website [19], weighted by the frequency of each password in the data set. Since RockYou was a low-value account website for which users may have chosen unusually weak passwords, and since one of the best ways to protect a website is to ban the most common passwords, we remove the 100 most common passwords from the distribution. This accords with recent advice that very common choices should be blocked [17] and the practice of some large sites, e.g. Twitter and Microsoft [13].

We associate one IP address with each account at the start of the simulation. Users' IP addresses are either chosen randomly from the IPv4 space or from a proxy. During IP-assignment, we assign users to the current proxy until it reaches a limit of users and then create a new one. We also associate each account with a device cookie (secret) that the user has previously employed to login to the service.

Not all users login at the same frequency. We compute the frequency with which a user logs in (the relative probability that user will be randomly selected for the next login) as the inverse of a log normal distribution.

When a user attempts to login, we either randomly choose a client IP address from one of the prior IP addresses employed by that user or we generate a new one using the approach above (15%). Similarly, with each login we either randomly associate the client with a cookie the user has used before (90%) or assign the user a new client cookie. We set a maximum limit on the number of cookies and IPs per user account.

When generating user's login attempts we randomly introduce user errors. For a fraction (by default one in 50) we introduce typos, by making small changes to the password. Since the confusion or typing difficulty that causes a typo makes another typo more likely, we assign a probability (by default 67%) that each typo is followed by a subsequent typo shortly after (by default 7 seconds). We follow the last typo in the resulting chain by the a login attempt with correct password shortly after. All of these parameters are of course configurable in the simulator.

For another fraction of attempts (by default one in 50), we simulate a user typing an entirely-incorrect password, as might occur if the user accidentally types a password used for another service. We replace the correct password with one chosen at random using the same probability distribution used to assign passwords. As with typos, each such error is followed by a subsequent occurrence of the same error with a greater probability and then the correct password is submitted.

For yet another fraction of attempts, we simulate users who mistakenly enter the wrong account name, replacing their account name with another user's at random. As with typos, users have an increased chance (20%) repeating this mistake shortly after until eventually sending the correct credentials.

For a very small fraction of attempts, we simulate an automated client that tries to login repeatedly with an old password. We first change the user's password, then create a chain of login attempts with the old passwords (by default one every five minutes for 24 hours, for a total of 288), followed by a chain of correct logins (one every five minutes for 24 hours) once the client has been given the correct password.

D. Simulating attackers

The simulator begins by assuming attackers control a specified (configurable) number of IP addresses. We assign a fraction of attackers' IP addresses from the set of IP addresses in use by legitimate users, as attackers may control machines also in use by legitimate users or behind the same proxy. These IP collisions can make it impossible to differentiate attackers correct guesses and users' valid logins coming from the same IP address. If the IP pools used by attackers and legitimate users are entirely disjoint the task becomes much easier.

We simulate three different types of attacks. The first is a *descending-popularity* attack (often called a statistical guessing attack), in which the attacker guesses the most-popular password against all known user accounts, then the next most-popular, and so on. To simulate the strongest attacker possible, we provide the attacker the actual distribution of passwords assigned to accounts. We also simulate a *weighted attack*, in which for each login attempt the attacker chooses a password at random, with candidates weighted by their popularity among users. Finally, we simulate a *detection-avoidance* attack, a variant of the descending-popularity attack in which the attacker abandons using a password after a fixed number of attempts (25) so as to avoid it being detected as frequently-guessed by the binomial ladder filter.

The simulator assigns a fraction of login attempts to be performed during the simulation by attackers (e.g., 50%). Since attackers may not know which account names are legitimate, we simulate guesses targeting non-existent account names by replacing valid account names with invalid ones (e.g., 10%).

V. Evaluation & Discussion

The efficacy of any defense against password-guessing attacks will depend on the attacker's strategy in choice and ordering of password/account pairs for each IP address the attacker controls. In Figure 2a we illustrate the efficacy of the three attacks introduced in Section IV-D: descending-popularity, weighted, and detection-avoidance. The graph shows the number of accounts compromised versus accounts experiencing false positives for the baseline fixed-failure threshold algorithm as T varies from 0 (block everything) to ∞ (no blocking). The Y axis intercept represents the number of accounts that will be compromised if no guesses are blocked (i.e. $T \rightarrow \infty$). The X axis intercept represents the number of false positives if everything is blocked (in this case all 5 million accounts in the simulation). Decreasing the threshold moves the line down, and to the right, as more guesses are blocked and more users have their legitimate logins fail.

The nearly-horizontal lines on the left side of each curve have two causes. First, we assume an attacker who limits the rate of testing from each IP to one attempt roughly every ten minutes (1,000 attempts over 7 days). The automated clients we simulate, which sometimes cache users' old passwords, attempt to login once every five minutes. Thus, as we have observed in our experience managing real-world systems, blocking algorithms that count repeat username/password pairs as distinct failures (rather than ignoring subsequent occurrences of these pairs) will lock out these clients first. So false positives increase at first without any decrease in false negatives. The second cause is the set of IP addresses used by both legitimate users and attackers, which represent machines, or proxied networks, that have been infiltrated by attackers. When attackers guess the most common passwords first, most of attackers' *correct* guesses are distributed toward the the start of the simulation period. On the other hand, legitimate users' correct logins are equally spaced over the entire simulation period and we count false positives if a user is blocked at any time during the period.

Figure 2a shows that, should attackers adopt the detection-avoidance attack to confound those blocking techniques that rely on identifying frequently-guessed password, the number of passwords they can expect to guess correctly falls by more than three orders of magnitude! In other words, forcing attackers to adopt a detection-avoidance strategy would be an effective blocking strategy in and of itself. Even a shift to a weighted attack would cause attackers more than an order of magnitude drop as compared to the descending-popularity attack.

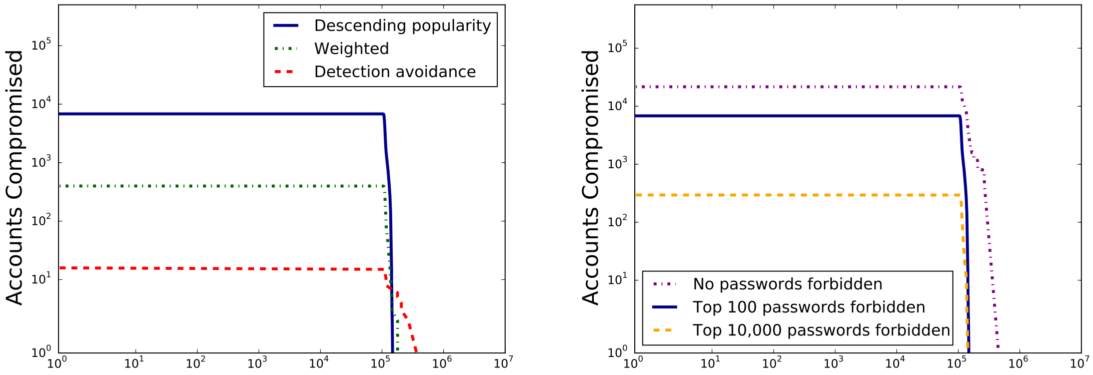
In Figure 2b, we examine the effectiveness of the descending-popularity attack had the site not banned any common passwords, or if it had banned the 10,000 most popular passwords. Here we see that using a popular-password prevention policy alone can reduce the efficacy of guessing attacks by two orders of magnitude. While we focus on the mid-point of 100 banned passwords for the rest of the analysis in this paper, those interested in how different blocking techniques perform in these other scenarios will answers in Figure 6.

In Figure 3, we show the performance of blocking against the descending-popularity (3a) and weighted (3b) attacks, zooming into the critical regions. (Readers who prefer ROC curves will find them as Figures 7a and 7b.) The thick monochromatic lines show the baseline of a fixed-failure threshold (gray) and of blocking using all the techniques in Section II (black). To illustrate the marginal contribution of each technique in Section II, we draw thin lines which show the impact of removing that technique while leaving all other techniques in place.

The most effective techniques appear to be to ignore repeat account/password pairs (Section II-D), and to adjust the blocking threshold if the account's password is among those frequently guessed (Section II-B). Indeed, practitioners who are looking to get the greatest improvement in detection for the least amount of effort would be well served to implement these two approaches first.

Penalizing IP addresses that guess frequently-guessed passwords (Section II-A) can actually reduce efficacy in certain circumstances, but is typically helpful—especially in the critical region for defending against descending-popularity attacks (the zoomed in region of Figure 3a). The reason that adjusting the blocking threshold is so much more effective than penalizing IP addresses appears to be that many false positives are caused when attackers and defenders share an IP (proxy or host). In this case both attacker and defender will be punished if we penalize the IP but the great majority of users (whose accounts do not use frequently-guessed passwords) will be unharmed if instead we adjust the threshold.

There is a significant benefit to allowing users to login if they have a cookie proving that they have successfully logged into that account before (Section II-F), though the benefit – roughly a factor of two – may not appear significant on a log graph. Adjusting scores for typos (Section II-C) has a



(a) Different attacks

(b) Different password policies (descending-popularity attack)

Fig. 2: The effectiveness of the baseline threshold-based IP blocking: we graph the number of accounts compromised vs. the number of accounts that experience login failures due to false positives as the threshold is increased. The leftmost point of each line illustrates the number of accounts that will be compromised if no blocking is performed. Subfigure (a) illustrates that descending popularity attacks are the most powerful by an order of magnitude, even though the attacks are simulated for a site that prevents users from choosing among the 100 most common passwords. For that attack, subfigure (b) illustrates the differences in the effectiveness of the descending-popularity attack against sites that prevent users from choosing the top 0, 100, or 10,000 most common passwords.

smaller impact but one that may still be significant in the critical region. There is also a small impact for penalizing invalid account names more than invalid passwords (Section II-E), but far smaller than we would expect than if we had simulated a system on which account names are hard to guess.

In Figure 4 we see that forbidding users from having common passwords and using frequently-guessed passwords for blocking are complementary techniques. Each line color and style shows detection rates for the baseline algorithm and for the algorithm of Section II. Combining the new algorithm with a ban on the top 10,000 passwords can bring the number of compromised accounts down to less than 20 while keeping the the number of accounts experiencing false positives down to double digits—even when a large number of IPs are shared between users and attackers.

A. *Limitations of our simulations*

Our main contributions are the introduction of new features to tell benign and attack traffic apart, a system that exploits those features without exposing passwords to new risk, and a simulator that allows exploration of performance in particular settings. We offer example simulations only as a guide.

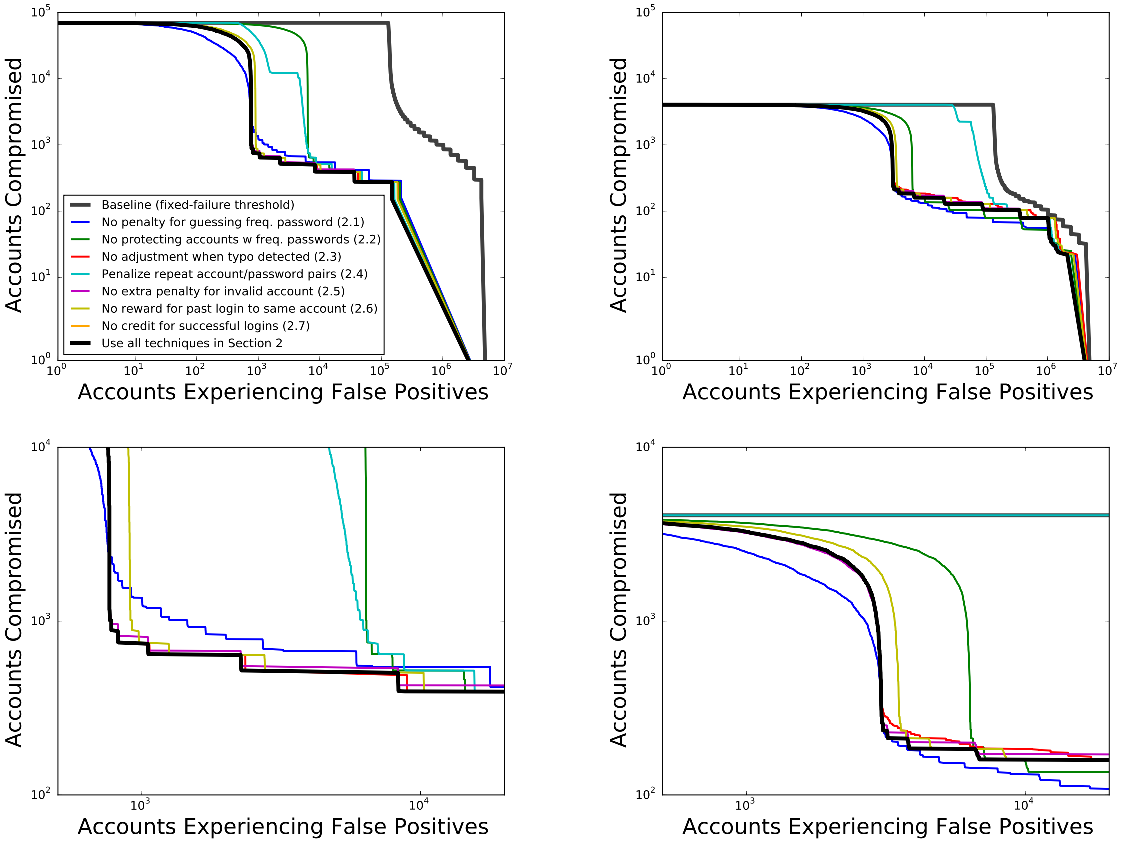
A limitation of our work is that we simulate both user and attack traffic in our evaluation section. This necessarily involves making certain assumptions and choosing default settings for the simulator. For example, in Section IV-C we assumed the rates at which users commit typos, misremembered passwords, have server-issued cookies etc. The actual rates might be higher or lower, and for some parameters may depend on the nature of the service. For attack traffic we simulated three different types of attack; we assumed a certain IP pool available to the attacker, and an overlap rate between legitimate and attacker IP addresses.

While it would be natural to prefer data from real-world user and attacker traffic there are a number of issues that make such data hard to obtain and publicize safely. First, such a data set would need to include real-world login attempts with the submitted passwords in plaintext and with labels differentiating guesses from user errors (ground truth). Such data are unlikely to become available and could not be safely shared if they were, yielding a result that could not be reproduced. Second, for several of these parameters there is no correct answer that applies to all services. For a web service with fewer than a million users there might be no overlap at all between the IP address pools used by attackers and users, while for Facebook, Gmail or Twitter it might be significant. Equally, the rate at which an attacker guesses against a non-existent username almost certainly depends on the service and the attacker; a measurement of this parameter at one site would not generalize to another.

Rather than leave the problem unaddressed for lack of perfect data, or consider unrealistic traffic by simplifying the traffic models, we provide an open-source simulator that others can use and modify to reproduce our results or to test different scenarios.

VI. Related Work

Several large sites have seen compromises due to online guessing attacks. In 2013, popular code-repository site GitHub was hit by a large scale online guessing attack from over $40,000$ unique IP addresses [18]. As a result, GitHub was forced to reset many accounts' passwords and introduced a ban on a list of weak passwords. Another example is iCloud,



(a) Descending-popularity attack

(b) Weighted attack

Fig. 3: We measure the effectiveness of baseline fixed-failure threshold blocking algorithm (the thick gray dotted line) and our algorithm (the thick black solid line) by comparing the number of accounts compromised (Y axis) with the number of accounts that experience a false block (X axis). To illustrate the role of each technique we have introduced in this paper, the thin lines illustrate the effect of removing one technique. We present two graphs in each Subfigure, with the top graph showing the full error detection curve and the bottom graph zoomed into the region that balances the number of compromised accounts with the number inconvenienced by false positives. All four graphs use the legend in the top left graph.

Apple's cloud storage service, which suffered an attack on celebrity accounts with consequent leakage of many personal photos. Since iCloud had no limit on failed login attempts for the "find my iPhone" functionality [38], the attacker was able to brute force the targeted celebrity accounts (which were protected by weak passwords).

Online guessing attacks are by no means a problem only for large sites. Seifert [39] documents online guessing against a honeypot SSH server. He reports that common passwords such as "123456" and "Password" dominated the guessing and a small number of IP addresses sent many of the guesses. Owens [31] reports attacks on three different SSH servers located on a small business, a residential DSL and a university campus networks. He finds great similarity between the attacks and concludes that "many brute-force attacks are based on pre-compiled lists of usernames and passwords, which are widely shared."

Whereas there is a dearth of research on online blocking, there is a wealth of research on preventing offline attacks, which require a compromised account database. Prior work includes Manber's introduction of iterated hashing [26], followed by the proposal of Provos and Mazieres [35] to make the number of iterations grow as advances in hardware decrease computation time. Recognizing that attackers can build custom hardware to speed computation, Percival [33], and others since, have designed password hashing functions designed to require both computation and large amounts of memory.

There has also been a wealth of recent research focused on helping users choose passwords that are harder to guess. Several large commercial [2] [4] [1] and government [5] sites offer advice with rules to help users choose passwords. Myriad papers have examined the usability and security of various rules [28], [25], [23]. Notably, Weir *et al.*.[27] examine various password creation policies and conclude that most do a poor job of ensuring that passwords will withstand a well-resourced offline attack.

More promising are recently-proposed password guidance based on data rather than composition rules. In 2009 Twitter

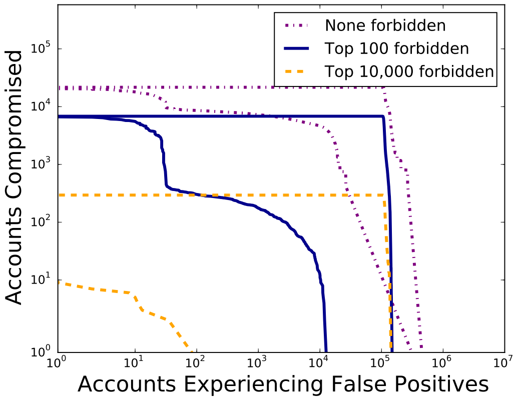


Fig. 4: An illustration of how blocking guesses and prohibiting users from choosing common passwords complement each other. Line styles represent three password policies: one allowing any password, one blocking the 100 most common passwords, and one blocking the 10,000 most common passwords. For each style we draw two lines: one for the baseline scheme of blocking after a threshold of failed attempts (the upper right of the two lines) and one integrating the techniques described in Section II. The line at the bottom left shows the benefits of combining a prohibition on popular passwords with techniques to block guessing.

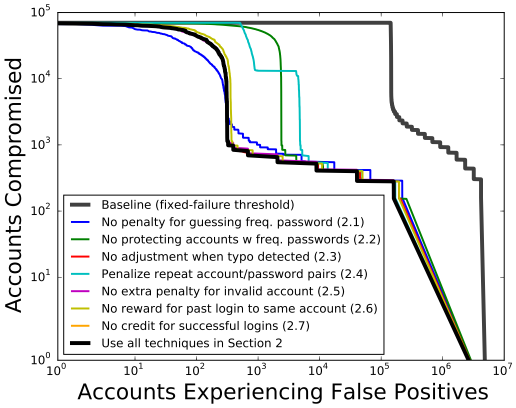


Fig. 5: Even when we don't intentionally create overlap between the IP addresses used by attackers and those used by defenders, the overlap occurs naturally at the scale of our tests due to the birthday paradox on IPv4 addresses.

banned a list of 370 known common passwords [6] and in 2011 Microsoft followed suit by banning its own list of common passwords for its online services [29]. Florêncio *et al.*.[13] suggest that website accounts should be able to withstand on the order of one million guesses and suggest banning several thousand of the most common choices.

In 2010, Schechter *et al.*. [41] proposed using a count-min sketch to identify and ban popular passwords while minimizing the risk of revealing uncommon passwords should the data structure be compromised. While asserting that a count-min sketch *can* be configured to ensure privacy, they do not provide such a configuration algorithm or any quantifiable guarantees about the information revealed each time a password is recorded. Further, their count-min sketch has to be replaced periodically to prevent false positives from growing too high. In contrast, with the binomial ladder that we use [37], we provide quantifiable measurements of probability reduction for each recorded password, a data structure that is trivial to initialize, and one that does not require replacement due to aging. Indeed, the binomial ladder would be a superior mechanism for banning popular passwords while minimizing the risk should attackers obtain a copy of the data structure.

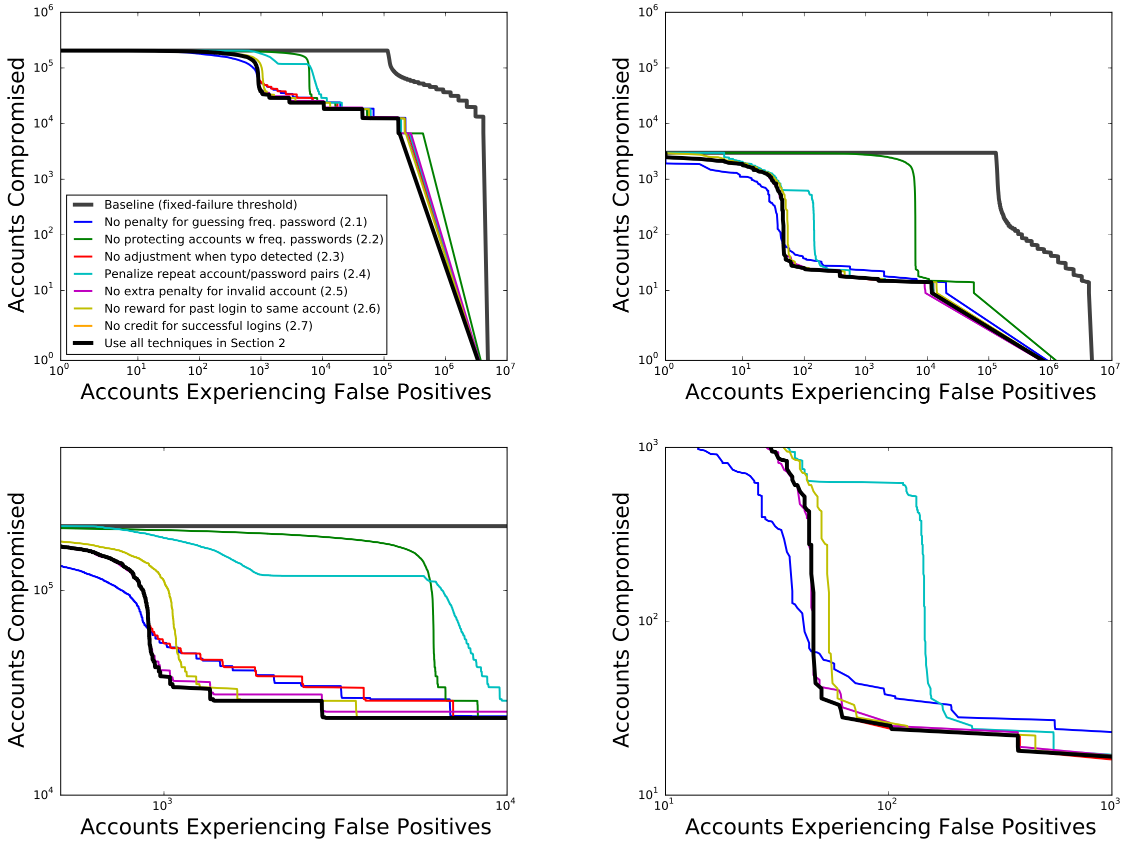
Wheeler's `zxcvbn` [44], introduced in 2014 and used by DropBox, scores passwords poorly if they contain words on a banned-password list. Komanduri *et al.*. describe a password meter that shows users predicted completions of the keys they type, thereby communicating the guessability of common choices [24]. Both approaches have been effective in user studies [24]. As both rely on lists of common passwords, both would benefit from a mechanism to identify passwords that become popular even when previously-popular passwords are banned—such as the binomial ladder filter.

Fixed per-account failure thresholds are a commonly-recommended approach to limit guessing, despite the downfall when applied to online services (as opposed to devices) which face descending-popularity attacks this policy may lock all users out of a service. Florêncio *et al.*.[12] observe that a 6 digit random PIN can withstand an online attack if the account is locked for 24 hours after three unsuccessful attempts. Brostoff and Sasse studied the 10-week login histories of 386 undergraduates users at a university where no lockout or throttling policy was in place. They observed an average of 34.5 attempts, 31 successes and 3.3 failures per user (i.e. a 9.6% failure rate). They observe that a "three strikes" policy would be unnecessarily restrictive and suggest a "ten strikes" lockout policy would reduce the number of password reset requests for negligible security impact [36]. Bonneau and Preibusch studied lockout policies at large web-sites allowing free sign-up [21]. Of 150 sites they found that 126 permitted a login even after 100 failed attempts. Thus, contrary to popular assumption, lockout policies appear quite rare.

Blocking IP addresses exceeding a threshold of failed logins has been used to protect low-utilization remote terminal (*e.g.*, `ssh`) servers [16], [20], [11], [42], but among its problems is indiscriminate blocking of large proxies.

Duermuth etal [10] describe a scheme for determining if a login event (with the correct password) comes from the legitimate user or an attacker. Their scheme uses the history of features such as IP address and browser useragent to detect compromised accounts. Their scheme is complementary to ours, in that it is agnostic to how the password was obtained and does not attempt to detect password guessing.

A number of organizations give guidance on protecting authentication servers but often this is vague or dated. OWASP



(a) No forbidden passwords

(b) 10,000 forbidden passwords

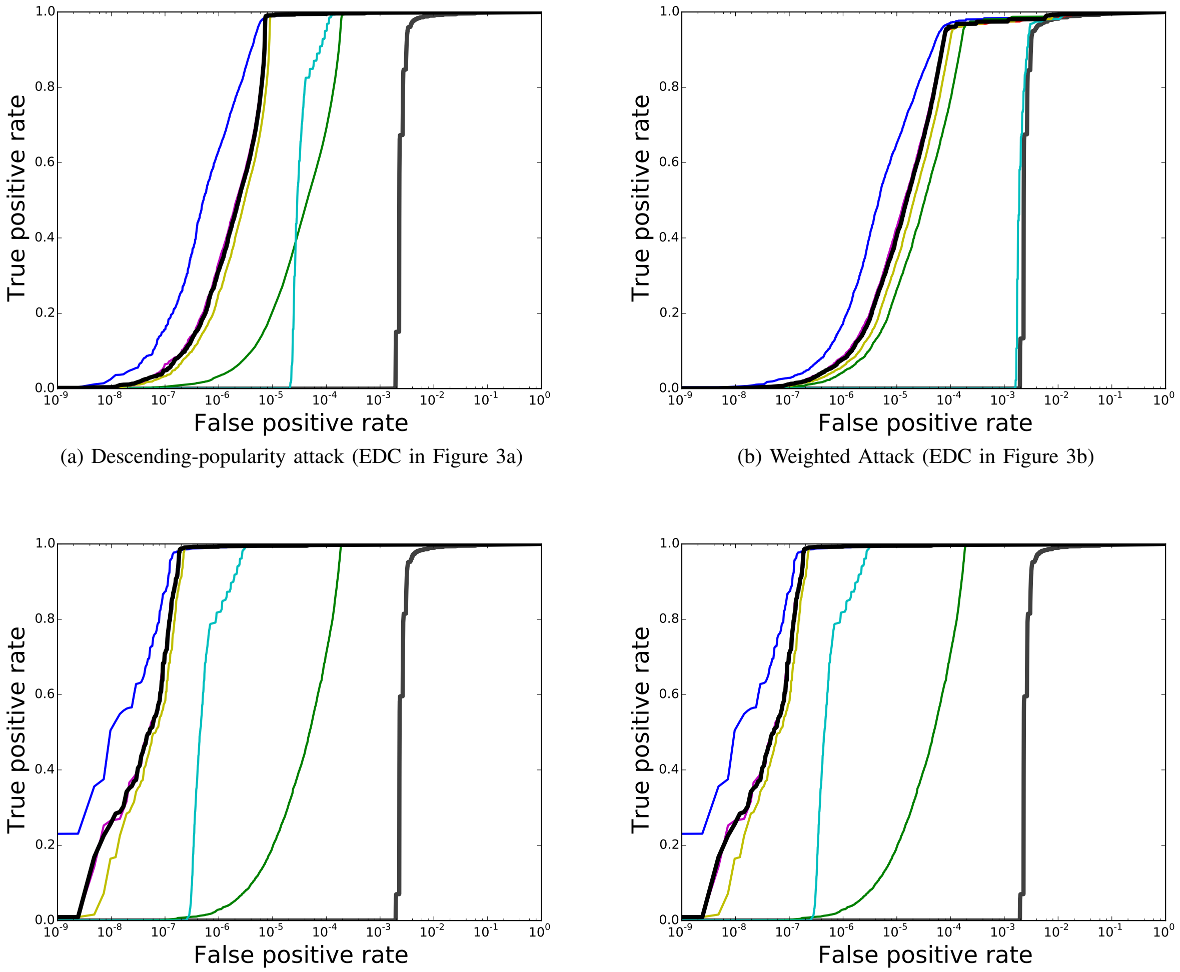
Fig. 6: Descending popularity attack results for sites with 0 and 10,000 forbidden passwords. The legend in the top left graph applies to all four graphs. We refer readers who prefer ROC curves to Figures 7c and 7d.

[40] recommends to "lockout accounts for a period of time (*e.g.,* 20 minutes)" concluding that "this significantly slows down attackers, while allowing the accounts to reopen automatically for legitimate users." While they list several ways to block attacks, such as locking accounts, device cookies, showing unpredictable behaviors for failed passwords, and using CAPTCHAS, there is little implementation detail.

The authentication libraries for platforms such as PHP, ASP.NET and Ruby on Rails often include some rudimentary mitigations. In an examination of these libraries we observe that the most common defense is to hard-code a default number of fails after which an account will be locked, or an IP address blocked. For example, an open source library called Brute Force Detection (BFD) [16] blocks an IP address if more than a certain number of failures is observed from an IP address. The observation duration, number of failed account logins or from an IP address are configurable, but there is no guidance on how they might be chosen. Microsoft's identity framework (which is likely to be used by anyone creating a site using Microsoft tools) supports account lockout [3] but not IP blocking.

Pinkas and Sander [34] suggest differentiating between human user and automated guessing scripts by presenting a CAPTCHA [7]. Van Oorschot and Stubblebine [32] and Alsaleh *et al.*. [8] enhance this approach and greatly reduce the circumstances under which users will be forced to respond to a challenge. CAPTCHAs can be used to complement blocking, though since users know when a CAPTCHA appears, the algorithm for determining when to use a CAPTCHA must not use a threshold that could leak information about the strength of a user's password (as our approach uses for blocking triggers). A shortcoming of approaches that rely on challenges to distinguish humans from computers is that they are less well suited for web services that may serve many automated clients (*e.g.,* email services).

Freeman et al [14] give an interesting approach to identifying hijack attempts among successful logins. Their approach focuses on identifying features that look anomalous; e.g. a user logging in from a very unlikely IP address, or with a previously unseen useragent. Their method is complementary to ours, in the sense that they do not explicitly attempt to identify guessing. We suggest several features that they do not use. On the other hand, their method has the potential to detect account compromise that does not come from guessing,



(a) Descending-popularity attack (EDC in Figure 3a)

(b) Weighted Attack (EDC in Figure 3b)

(c) No forbidden passwords (EDC in Figure 6a)

(d) 10,000 forbidden passwords (EDC in Figure 6b)

Fig. 7: ROC curves matching earlier error-detection curves. Whereas the error-detection curves count the number of users experiencing false positives, these curves count every failed login as a separate false positive. The legend for the error-detection curves also describes the colors for these curves.)

while ours does not.

VII. Conclusion

We have built a system that protects services from online guessing attacks. Among our contributions are features to make better benign/attack traffic classification decisions. These include mechanisms to penalize fail events with commonly guessed passwords more, to protect accounts with weak passwords better, and to distinguish user-generated typos from other fails. We do all of these without storing any information which exposes accounts to new risk in the event of a server breach. These innovations allow much greater flexibility over the previous state-of-the-art, which treats all failures equally. The system is available to any password-based service exposed to online guessing attack, and welcomes further contributions from researchers and developers.

Based on simulations, we recommend that if practitioners can only implement a subset of the techniques in this paper, the choose the following two. First, only count account/password pairs as failures once, and do not punish IPs or clients that repeatedly submit the wrong password for an account. Second, protect users who have chosen passwords that are among those frequently guessed by attackers, either by immediately forcing those users to reset their passwords or by significantly reducing the IP blocking threshold for login attempts on these accounts.

References

[1] Amazon: Choose a strong password. http://www.amazon.com/gp/help/customer/display.html.

[2] Google: Secure Your Passwords. http://www.google.com/goodtoknow/online-safety/passwords/.

[3] Microsoft asp.net identity framework lockout options. https://github.com/aspnet/Identity/blob/8796f7e78a2355d322c0380ffa6b02107e05f20b/src/Microsoft.AspNet.Identity/LockoutOptions.cs.

[4] Microsoft Security and Safety Center. http://www.microsoft.com/security/online-privacy/passwords-create.aspx.

[5] Stop.Think.Connect. http://www.stopthinkconnect.org/.

[6] Wired: Weak Password Brings 'Happiness' to Twitter Hacker. http://blog.wired.com/27bstroke6/2009/01/professed-twitt.html.

[7] AHN, L., BLUM, M., HOPPER, N., AND LANGFORD, J. Captcha: Using hard ai problems for security. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques* (2003), Springer-Verlag, pp. 294–311.

[8] ALSALEH, M., MANNAN, M., AND VAN OORSCHOT, P. C. Revisiting defenses against large-scale online password guessing attacks. *Dependable and Secure Computing, IEEE Transactions on 9*, 1 (2012), 128–141.

[9] BORTZ, A., BONEH, D., AND NANDY., P. Exposing Private Information by Timing Web Applications. *WWW 2007, Banff*.

[10] DÜRMUTH, M., FREEMAN, D. M., JAIN, S., BIGGIO, B., AND GIACINTO, G. Who are you? a statistical approach to measuring user authenticity.

[11] Fail2ban. http://www.fail2ban.org/.

[12] FLORÊNCIO, D., HERLEY, C., AND COSKUN, B. Do Strong Web Passwords Accomplish Anything? *Proc. Usenix Hot Topics in Security* (2007).

[13] FLORÊNCIO, D., HERLEY, C., AND VAN OORSCHOT, P. C. An administrator's guide to internet password research. In *USENIX LISA* (2014).

[14] FREEMAN, D. M., JAIN, S., DÜRMUTH, M., BIGGIO, B., AND GIACINTO, G. Who are you? a statistical approach to measuring user authenticity. In *23rd Annual Network & Distributed System Security Symposium (NDSS). The Internet Society* (2016).

[15] FRIEDMAN, J., HASTIE, T., AND TIBSHIRANI, R. *The elements of statistical learning*, vol. 1. Springer series in statistics Springer, Berlin, 2001.

[16] FX NETWORKS, R. Brute force detection. https://www.rfxn.com/projects/brute-force-detection/.

[17] GCHQ. Password guidance simplifying your approach. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/458857/Password_guidance_-_simplifying_your_approach.pdf.

[18] GITHUB BLOG. Weak passwords brute forced. https://github.com/blog/1698-weak-passwords-brute-forced.

[19] IMPERVA. Consumer Password Worst Practices.

[20] INC, L. W. What is brute force detection (bfd)? http://www.liquidweb.com/kb/what-is-brute-force-detection-bfd/.

[21] J. BONNEAU AND S. PREIBUSCH. The Password Thicket: technical and Market Failures in Human Authentication on the Web. *WEIS* (2010).

[22] KALISKI, B. Pkcs #5: Password-based cryptography specification version 2.0. IETF RFC 2898.

[23] KELLEY, P. G., KOMANDURI, S., MAZUREK, M. L., SHAY, R., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND LOPEZ, J. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 523–537.

[24] KOMANDURI, S., SHAY, R., CRANOR, L. F., HERLEY, C., AND SCHECHTER, S. Telepathwords: Preventing weak passwords by reading users minds. In *23rd USENIX Security Symposium (USENIX Security 14). San Diego, CA: USENIX Association* (2014), pp. 591–606.

[25] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), ACM, pp. 2595–2604.

[26] MANBER, U. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security 15*, 2 (1996), 171–176.

[27] MATT WEIR, SUDHIR AGGARWAL, MICHAEL COLLINS, HENRY STERN. Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords. In *Proc CCS* (2010).

[28] MAZUREK, M. L., KOMANDURI, S., VIDAS, T., BAUER, L., CHRISTIN, N., CRANOR, L. F., KELLEY, P. G., SHAY, R., AND UR, B. Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 173–186.

[29] MCDOUGALL, P. Hotmail bans guessable passwords, like 'password'. http://www.informationweek.com/applications/hotmail-bans-guessable-passwords-like-password/d/d-id/1099007 month=jul, year=2011,.

[30] METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory* (2005), Springer, pp. 398–412.

[31] OWENS JR, J. P. *A study of passwords and methods used in brute-force SSH attacks*. PhD thesis, Clarkson University, 2008.

[32] P. C. VAN OORSCHOT, S. STUBBLEBINE. On Countering Online Dictionary Attacks with Login Histories and Humans-in-the-Loop. *ACM TISSEC vol.9 issue 3* (2006).

[33] PERCIVAL, C. Stronger key derivation via sequential memory-hard functions. In *BSDCan* (May 2009).

[34] PINKAS, B., AND SANDER, T. Securing Passwords Against Dictionary Attacks. *ACM CCS* (2002).

[35] PROVOS, N., AND MAZIERES, D. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 81–91.

[36] S. BROSTOFF AND M. A. SASSE. "Ten Strikes and You're Out": Increasing the Number of Login Attempts Can Improve Password Usability. *CHI Workshop* (2003).

[37] S. SCHECTER AND C. HERLEY. The Binomial Ladder Frequency Filter and its Applications to Shared Secrets. *MSR-TR-2018-18* (2016).

[38] SEAN GALLAGHER. Apple confirms celebrities accounts breached. http://arstechnica.com/tech-policy/2014/09/apple-confirms-celebrities-accounts-breached-in-highly-targeted-attack/.

[39] SEIFERT, C. Analyzing malicious SSH login attempts. *Security Focus, Infocus 1876* (2006).

[40] STEVEN, J., AND MANICO, J. *Password Storage Cheat Sheet (OWASP)*. OWASP. Apr.7, 2014, https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet.

[41] STUART SCHECHTER AND CORMAC HERLEY AND MICHAEL MITZENMACHER. Popularity is Everything: a new approach to protecting passwords from statistical-guessing attacks. *Proc. HotSec, 2010.*

[42] Team password manager. http://teampasswordmanager.com/docs/ip-address-blocking.

[43] VAN TREES, H. L. *Detection, Estimation and Modulation Theory: Part I*. Wiley, 1968.

[44] WHEELER, D. zxcvbn: realistic password strength estimation. Dropbox Tech Blog. https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/, Apr. 2012.