Honeywords: making password-cracking detectable

Honeywords: Making Password-Cracking Detectable

Ari Juels
RSA Labs
Cambridge, MA 02142
ajuels@rsa.com

Ronald L. Rivest
MIT CSAIL
Cambridge, MA 02139
rivest@mit.edu

May 2, 2013
Version 2.0

Abstract

We suggest a simple method for improving the security of hashed passwords: the maintenance of additional "honeywords" (false passwords) associated with each user's account. An adversary who steals a file of hashed passwords and inverts the hash function cannot tell if he has found the password or a honeyword. The attempted use of a honeyword for login sets off an alarm. An auxiliary server (the "honeychecker") can distinguish the user password from honeywords for the login routine, and will set off an alarm if a honeyword is submitted.

Keywords: passwords, password hashes, password cracking, honeywords, chaffing, login, authentication.

1 Introduction

Passwords are a notoriously weak authentication mechanism. Users frequently choose poor passwords. An adversary who has stolen a file of hashed passwords can often use brute-force search to find a password p whose hash value $H(p)$ is equal to the hash value stored for a given user's password, thus allowing the adversary to impersonate the user.

A recent report by Mandiant[1] illustrates the significance of cracking hashed passwords in the current threat environment. Password cracking was instrumental, for instance, in a recent cyberespionage campaign against the *New York Times* [32]. The past year has also seen numerous high-profile thefts of files containing consumers' passwords; the hashed passwords of Evernote's 50 million users were exposed [20] as were those of users at Yahoo, LinkedIn, and eHarmony, among others [19].

One approach to improving the situation is to make password hashing more complex and time-consuming. This is the idea behind the "Password Hashing Competition."[2] This approach can help, but also slows down the authentication process for legitimate users, and doesn't make successful password cracking easier to detect.

Sometimes administrators set up fake user accounts ("honeypot accounts"), so that an alarm can be raised when an adversary who has solved for a password for such an account by inverting a hash from a stolen password file then attempts to login. Since there is really no such legitimate user, the adversary's attempt is reliably detected when this occurs. However, the adversary may be able to distinguish real usernames from fake usernames, and thus avoid being caught.

Our suggested approach can be viewed as extending this basic idea to *all* users (i.e., including the legitimate accounts), by having *multiple possible passwords* for each account, only one of which is genuine. The others we call "*honeywords*." The attempted use of a honeyword to login sets off an alarm, as an adversarial attack has been reliably detected.

This approach is not terribly deep, but it should be quite effective, as it puts the adversary at risk of being detected with *every* attempted login using a password obtained by brute-force solving a hashed password.

Thus, honeywords can provide a very useful layer of defense.

[1] http://intelreport.mandiant.com/

[2] https://password-hashing.net/index.html

We are not sure whether this approach is new; we have not seen any literature describing precisely the same proposal. (Let us know if you have!) Perhaps the closest related work we're aware of is the Kamouflage system of Bojinov et al. [6]. See Section 8 for further details. To the best of our belief, the term "honeyword" first appeared in that work. Also closely related to our proposal is the anecdotally reported practice of placing whole, bogus password files ("honeyfiles") on systems and watching for submission of any password they contain as signalling an intrusion.

In any case, our hope is that this note will help to encourage the use of honeywords.

2 Technical Description

2.1 Context

We assume a computer system with n users $u_1, u_2, \ldots, u_n$; here u_i is the username for the ith user. By "computer system" (or just "system" for short) we mean any system that allows a user to "log in" after she has provided a username and a password; this includes multi-user computer systems, web sites, smart phones, applications, etc.

We let p_i denote the password for user u_i. This is the correct, legitimate, password; it is what user u_i uses to log in to the system.

In current practice, the system uses a cryptographic hash function H and stores hashes of passwords rather than raw passwords. That is, the system maintains a file F listing username / password-hash pairs of the form

$$(u_i, H(p_i))$$

for $i = 1, 2, \ldots, n$. On Unix systems the file F might be `/etc/passwd` or `/etc/shadow`.

The system stores password hashes rather than raw passwords so that an adversary with access to F does not find out the passwords directly; he must invert the hash function (compute p_i from $H(p_i)$) to find out the password for user u_i (see Evans et al. [1] and Purdy [33]).

The computation of the hash function H may (should!) involve the use of system-specific or user-specific parameters ("salts"); these details don't matter to us here. When a user attempts to log in, the file F is checked for the presence of the hash of the proffered password in the user's entry (see Morris and Thompson [26]).

2.2 Attack scenarios

There are many attack scenarios relating to passwords, including the following six:

- **Stolen files of password hashes:** An adversary is somehow able to steal the file of password hashes, and solve for many passwords using offline brute-force computation. He may more generally be able to steal the password hash files on many systems, or on one system at various times.
- **Easily guessable passwords:** A substantial fraction of users choose passwords so poorly that an adversary can successfully impersonate at least some users of a system by attempting logins with common passwords. (See Bonneau [7, 8].) Schecter et al. [36] suggest fighting this threat by requiring users to use uncommon passwords.
- **Visible passwords:** A user's password is compromised when an adversary views it being entered (shoulder-surfing), or an adversary sees it on a yellow stickie on a monitor. A *one-time password generator*[3] such as RSA's SecurID token provides good protection against this threat.
- **Same password for many systems or services:** A user may use the same password on many systems, so that if his password is broken on one system, it is also thereby broken on others.
- **Passwords stolen from users:** An adversary may learn user passwords by compromising endpoint devices, such as phones or laptops, using malware or by perpetrating phishing attacks against users.
- **Password change compromised:** The mechanism for allowing users to change or recover their passwords is defective or compromised, so an adversary can learn a user's password, or set it to a known value.

We focus on the first attack scenario where an adversary has obtained a copy of the file F of usernames and associated hashed passwords, and has obtained the values of the salt or other parameters required to compute the hash function H.

In this scenario, the adversary can perform a brute-force search over short or likely passwords, hashing each one (with salting if necessary) until the adversary determines the passwords for one or more users. (See for example Weir et al. [40].) Assuming that passwords are the

[3] `http://en.wikipedia.org/wiki/One-time_password`

only authentication mechanism in place, the adversary can then log in to the accounts of those users in a reliable and undetected manner.

In this paper, we assume that the adversary can invert most or many of the password hashes in F.

We assume that the adversary does not compromise the system on a persistent basis, directly observing and capturing newly created passwords and honeywords. (Certainly, the adversary risks detection the *first* time he tries logging in using a cracked password, since he may be using a honeyword; after that, the ability of the system to detect further attempts to login using cracked passwords may be compromised if the adversary is able to modify the login routine and its checks, or the password-change routine.)

Although our methods are directed to the first attack scenario, some of our approaches (e.g., the take-a-tail method) also have beneficial effects on password strength, thus helping to defeat the other attacks as well.

2.3 Honeychecker

We assume that the system may utilize an auxiliary secure server called the "*honeychecker*" to assist with the use of the honeywords.

Since we are assuming that the computer system is vulnerable to having the file F of password hashes stolen, one must also assume that salts and other hashing parameters can also be stolen. Thus, there is likely no place on the computer system where one can safely store additional secret information with which to defeat the adversary.

The honeychecker is thus a separate hardened computer system where such secret information can be stored. We assume that the computer system can communicate with the honeychecker when a login attempt is made on the computer system, or when a user changes her password. We assume that this communication is over dedicated lines and/or encrypted and authenticated. The honeychecker should have extensive instrumentation to detect anomalies of various sorts.

We also assume that the honeychecker is capable of raising an alarm when an irregularity is detected. The alarm signal may be sent to an administrator or other party different than the computer system itself.

Depending on the policy chosen, the honeychecker may or may not reply to the computer system when a login is attempted. When it detects that something is amiss with the login attempt, it could signal to the computer system that login should be denied. On the other hand it may merely signal a "silent alarm" to an administrator, and let the login on the computer system proceed. In the latter case, we could perhaps call the honeychecker a "login monitor" rather than a "honeychecker."

Our honeychecker maintains a single database value $c(i)$ for each user u_i; the values are small integers in the range 1 to k, for some small integer parameter k (e.g. $k = 20$). The honeychecker accepts commands of exactly two types:

- **Set:** i, j
  Sets $c(i)$ to have value j.
- **Check:** i, j
  Checks that $c(i) = j$. May return result of check to requesting computer system. May raise an alarm if check fails.

Design principles. The computer system and honeychecker together provide a basic form of distributed security. A distributed security system aims to protect secrets even when an adversary compromises some of its systems or software. Diversifying the resources in the system—for example, placing the computer system and honeychecker in separate administrative domains or running their software on different operating systems—makes it harder to compromise the system as a whole.

We have designed the protocol so that compromise of the honeychecker database by itself does not allow an adversary to impersonate a user. In fact, the honeychecker only stores randomly selected integers (the index c(i) for each u_i).

Indeed, one of our design principles is that *compromise (i.e. disclosure) of the honeychecker database at worst only reduces security to the level it was at before the introduction of honeywords and the honeychecker.* Disclosure of the file F then means that an adversary will now no longer be fooled by the presence of the honeywords; he will just need to crack the users' actual passwords, since he now knows which hash values are for the real passwords, and which hash values are for the honeywords.

As we discuss in Section 8, other distributed approaches to password protection are possible. Distributed cryptographic protocols for instance can prevent disclosure of passwords and even password hashes completely against compromise of the computer system. Unlike such schemes, though, honeywords can be incorporated into existing password systems with few system changes and little overhead in computation and communication.

We also design the honeychecker interface to be extremely simple, so that building a hardened honeychecker should be realistic. Importantly, the honeychecker need

not interact with the computer system. If configured to send "silent alarms" to administrators or trigger defenses such as we consider in Section 2.5, it need not even provide input to user authentication decisions.

The honeychecker could be a product, using a standardized interface. The honeychecker could serve a number of computer systems.

2.4 Approach – Setup

This subsection describes how honeywords work, in simplest form.

For each user u_i, a list W_i of distinct words (called "potential passwords" or more briefly, "sweetwords") is represented:

$$W_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,k}) \ .$$

Here k is a small integer, such as our recommended value of $k = 20$. For simplicity we assume that k is a fixed system-wide parameter, although k could be chosen in a per-user manner—for example, $k_i = 200$ for the system administrator(s). Values as low as $k = 2$ or as large as $k = 1000$ might be reasonable choices in some circumstances.

Exactly one of these sweetwords $w_{i,j}$ is equal to the password p_i known to user u_i. Let $c(i)$ denote the "correct" index of user u_i's password in the list W_i, so that

$$w_{i,c(i)} = p_i.$$

Although we call the $w_{i,j}$ entries "potential passwords," they could be phrases or other strings; a potential password could be a "potential passphrase" or a "sweetphrase."

The correct password is also called the "sugarword."

The other $(k-1)$ words $w_{i,j}$ are called "honeywords," "chaff," "decoys," or just "incorrect passwords."

The list W_i of sweetwords thus contains one sugarword (the password) and $(k-1)$ honeywords (the chaff).

We also allow a sweetword to be what we call a "*tough nut*"—that is, a very strong password whose hash the adversary is unable to invert. We represent a tough nut by the symbol '$\boxed{?}$'. A honeyword, or the password itself, may be a tough nut.

The definition of the file F is changed so that it now contains an extended entry for each user u_i, of the form:

$$(u_i, H_i) \ ,$$

where

$$v_{i,j} = H(w_{i,j})$$

is the value of the hash of the user's jth sweetword $w_{i,j}$, and

$$H_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,k})$$

is the list of all these hash values.

The file F is now larger by a factor of roughly k. Given the rapidly decreasing cost of storage this expansion should not cause any problems on a typical computer system, even for k as large as our recommended value of $k = 20$. Many systems already store hashes of ten or more old passwords per user to limit password reuse [17].

The user only needs (as usual) to remember her password p_i; she does not need to know the values of the honeywords or even know about their existence.

We let $\mathbf{Gen}(k)$ denote the procedure used to generate both a list W_i of length k of sweetwords for user u_i and an index $c(i)$ of the correct password p_i within W_i:

$$(W_i, c(i)) = \mathbf{Gen}(k)$$

Here **Gen** is typically randomized and must involve interaction with the user (otherwise the user cannot create or know the password). We may represent this user interaction in some cases by allowing an additional argument in the form of a user-supplied password p_i to **Gen**, so that $\mathbf{Gen}(k; p_i)$ ensures that p_i is the password in W_i; that is, $p_i = w_{i,c(i)}$.

The table c is maintained in a secure manner; in the proposal of this note it is stored on the honeychecker.

Salt. Again, we omit discussion of per-system or per-user salts or other parameters that may be included in the hash computation. We do, however, strongly urge the use of per-user salts.

Additionally, we recommend that hashing of $w_{i,j}$ now also take j as an additional parameter. Such distinct per-sweetword salting prevents an adversary from hashing a password guess once (with the per-user salt) and then checking the result simultaneously against all of the user's hashed sweetwords.

2.5 Approach – Login

The system login routine needs to determine whether a proffered password g is equal to the user's password or not. (Here mnemonics for g are "given" or "guess.") If g is not the user's password, the login routine needs to determine whether g is a honeyword or not.

If the user—or perhaps the adversary—has entered the user's correct password, then login proceeds successfully as usual.

If the adversary has entered one of the user's honeywords, obtained for example by brute-forcing the password file F, then an appropriate action takes place (determined by policy), such as

- setting off an alarm or notifying a system administrator,
- letting login proceed as usual,
- letting the login proceed, but on a honeypot system,
- tracing the source of the login carefully,
- turning on additional logging of the user's activities,
- shutting down that user's account until the user establishes a new password (e.g. by meeting with the sysadmin),
- shutting down the computer system and requiring all users to establish new passwords.

How does the login routine determine whether $g = p_i$ (that is, that the given word is the password)?

If the hash $H(g)$ of g does not occur in the file F in the user u_i's entry H_i, then word g is neither the user's password nor one of the user's honeywords, so login is denied.

Otherwise the login routine needs to determine whether g is the user's password, or it is merely one of the user's honeywords. The login routine can determine the index j such that $H(g) = v_{i,j}$, but the login routine doesn't know whether $j = c(i)$, in which case g is indeed the password, or not, in which case g is just a honeyword.

The table c is maintained securely in the separate secure "*honeychecker*" server described in Section 2.3. The computer system sends the honeychecker a message of the form:

Check: i, j

meaning: "Someone has requested to login as user u_i and has supplied sweetword j (that is, $w_{i,j}$) in response to the login password prompt. Please determine if $j = c(i)$, and take the appropriate action according to policy."

The honeychecker determines whether $j = c(i)$; if not, an alarm is raised and other actions may be taken. The honeychecker may (or may not, depending on policy) then respond with a (signed) message indicating whether login should be allowed.

It may be desirable for a "**Check**" message to be sent to the honeychecker, even when the proffered password g is not on the list W_i of sweetwords; in this case the check message could specify $j = 0$. In this variant the honeychecker is notified of *every* login attempt, and can observe when a password guessing attack is in progress.

It might also be desirable for a "**Check**" message to include additional information that might be forensically useful, such as the IP address of the user who attempted to log in. We don't pursue such ideas further here.

Many systems suspend an account if (say) five or more unsuccessful login attempts are made. With our approach, this limit is likely to be reached even if the adversary has access to W_i: The chance that the user's password does not appear in the first five elements of a random ordering of a list W_i of length 20 containing the user's password is exactly 75%. However, when failed attempts are made with honeywords rather than arbitrary non-sweetwords, a reduced limit may be appropriate before lockout occurs and/or additional investigations are undertaken.

2.6 Approach – Change of password

When user u_i changes her password, or sets it up when her account is first initialized, the system needs to:

- use procedure **Gen**(k) to obtain a new list W_i of k sweetwords, the list H_i of their hashes, and the value $c(i)$ of the index of the correct password p_i in W_i.
- securely notify the honeychecker of the new value of $c(i)$, and
- update the user's entry in the file F to (u_i, H_i).

We emphasize that the honeychecker does not learn the new password or any of the new honeywords. All it learns is the position $c(i)$ of the hash $v_{i,c(i)}$ of user u_i's new password in the user's list H_i in F. To accomplish this, the computer system sends the honeychecker a message of the form:

Set: i, j

meaning: "User u_i has changed or initialized her password; the new value of $c(i)$ is now j." (This message should of course be authenticated by the system to the honeychecker.)

3 Security definitions

We define the security of a honeyword generation algorithm **Gen**, using an *adversarial game*, an algorithm or thought experiment that models the capabilities of the adversary.

For simplicity, we consider a honeyword generation scheme of the form $\mathbf{Gen}(k; p_i)$, with user input p_i. (The definition may be adapted to other forms of **Gen**.)

The game proceeds as follows:

- $\mathbf{Gen}(k, p_i)$ is run, using a user-provided input consisting of a proposed password p_i chosen according to a probability distribution U over passwords that meet a specified password-composition policy.

  The output of $\mathbf{Gen}(k; p_i)$ is a list W_i of sweetwords and an index $c(i)$ such that the password p_i is in the $c(i)$-th position of W_i.

- The adversary is given W_i, with the exception that some randomly chosen honeywords output by $\mathbf{Gen}(k; p_i)$ may be "tough nuts"; for those honeywords the adversary only sees the symbol $\boxed{?}$ and not the underlying (hard) honeyword.

- The adversary must now either "pass" (refuse to play further) or else submit a guess $j \in \{1, 2, \ldots, k\}$ for the index $c(i)$.

The outcome is three-way:

- The adversary "wins" the game (or "succeeds") if he guesses and his guess is correct ($j = c(i)$).
- The adversary is "caught" if he guesses but his guess is a honeyword.
- The adversary "passes" if he doesn't play.

Flatness. Let z denote the adversary's expected probability of winning the game, given that the adversary does not pass. This probability is taken over the user's choice of password p_i, the generation procedure $\mathbf{Gen}(k; p_i)$, and any randomization used by the adversary to produce its guess j. Observe that $z \geq 1/k$, since an adversary can win with probability $1/k$ merely by guessing j at random.

We say a honeyword generation method is "ϵ-flat" ("epsilon-flat") for a parameter ϵ if the maximum value *over all adversaries* of the adversary's winning probability z is ϵ.

If the generation procedure is as flat as possible (i.e., $1/k$-flat), we say it is "perfectly flat" (for a given distribution U). If it is ϵ-flat for ϵ not much greater than $1/k$, we say that it is "approximately flat."

Our recommended value of $k = 20$ means that an adversary who has compromised F and inverted H successfully k times to obtain all 20 sweetwords has a chance of at most 5% of picking the correct password p_i from this list, if **Gen** is perfectly flat. In this ideal case, $\epsilon = 1/20$.

Note that if honeyword-generation is ϵ-flat, then an adversary who plays has at least a $(1-\epsilon)$ chance of picking a honeyword, being caught, and setting off an alarm. So a method that is perfectly flat ensures that an adversary who plays has a chance of least $(k-1)/k$ of being caught.

In some cases, even a modest chance of catching an adversary, e.g., $1-\epsilon = 1/4$ (a 25% chance of detecting sweetword guessing), would be sufficient to detect systemic exploitation by the adversary of a compromised password file—and perhaps even deter the adversary from attacking the system altogether. So while a flat **Gen** is ideal, **Gen** *may be effective even if not flat.*

4 Honeyword Generation

This section proposes several flat (or approximately flat) generation procedures **Gen** for constructing a list W_i of sweetwords and for choosing an index $c(i)$ of the actual password within this list.

The procedures split according to whether there is an impact on the user interface (UI) for password change. (The login procedure is always unchanged.) We distinguish the two cases:

- With *legacy-UI* procedures, the password-change UI is unchanged. This is arguably the more important case. We propose two legacy-UI procedures: *chaffing-by-tweaking* (which includes *chaffing-by-tail-tweaking* and *chaffing-by-tweaking-digits* as special cases), and *chaffing-with-a-password-model*.
- With *modified-UI* procedures, the password-change UI is modified to allow for better password/honeyword generation. We propose a modified-UI procedure called *take-a-tail*. With take-a-tail, the UI change is really very simple: the user's new password is modified to end with a given randomly-chosen three-digit value. Otherwise take-a-tail is the same as chaffing-by-tail-tweaking.

We explain the legacy-UI scenario and associated methods in Section 4.1, and the modified-UI scenario and the take-a-tail method in Section 4.2.

Many other approaches are possible, and we consider it an interesting problem to devise other practical methods under various assumptions about the knowledge of the adversary and the password-selection behavior of users.

4.1 Legacy-UI password changes

With a legacy-UI method, the password-change procedure asks the user for the new password (and perhaps

asks her to type it again, for confirmation). The UI does not tell the user about the use of honeywords, nor interact with her to influence her password choice.

A nice aspect of legacy-UI methods is that one can change the honeyword generation procedure without needing to notify anyone or to change the UI.

We start with a password p_i supplied by user u_i.

The system then generates a set of $k-1$ honeywords "similar in style" to the password p_i, or at least plausible as legitimate passwords, so that an adversary will have difficulty in identifying p_i in the list W_i of all sweetwords for user u_i.

- **Chaffing:** The password p_i is picked, and then the honeyword generation procedure $\mathbf{Gen}(k; p_i)$ or "chaff procedure" generates a set of $k-1$ additional distinct honeywords ("chaff"). Note that the honeywords may depend upon the password p_i. The password and the honeywords are placed into a list W_i, in random order. The value $c(i)$ is set equal to the index of p_i in this list.

The success of chaffing depends on the quality of the chaff generator; the method fails if an adversary can easily distinguish the password from the honeywords.

We propose two basic methods for chaffing (and one method for embellishing chaff). They are somewhat heuristic; the chaffing approach in general offers no provable guarantee that the honeyword generation procedure **Gen** is flat—particularly if the user chooses her password in a recognizable manner.

We note (but do not discuss further) the obvious fact that if there are syntax or other restrictions on what is allowed as a password (see Section 6.1), then honeywords should also satisfy the same restrictions.

4.1.1 Chaffing by tweaking

Our first method is to "tweak" selected character positions of the password to obtain the honeywords. Let t denote the desired number of positions to tweak (such as $t = 2$ or $t = 3$). For example, with "*chaffing-by-tail-tweaking*" the *last* t positions of the password are chosen.

The honeywords are then obtained by tweaking the characters in the selected t positions: each character in a selected position is replaced by a randomly-chosen character of the same type: digits are replaced by digits, letters by letters, and special characters (anything other than a letter of a digit) by special characters.

For example, if the user-supplied password is "`BG+7y45`", then the list W_i might be (for tail-tweaking with $t = 3$ and $k = 4$):

`BG+7q03`, `BG+7m55`, `BG+7y45`, `BG+7o92`.

We call the password-tail the "sugar", while the honeyword tails we call "honey" (of course).

Other tweaking patterns could also be used, such as choosing the last digit position and the last special-character position. With "*chaffing-by-tweaking-digits*" the last t positions containing digits are chosen. (If there are less than t digits in the password, then positions with non-digits could also be selected as needed.) Here is an example of chaffing-by-tweaking-digits for $t = 2$:

`42*flavors`, `57*flavors`, `18*flavors` (1)

Chaffing-by-tweaking-digits is typical of the "tweaks" users often use to derive new passwords from old ones; see Zhang et al. [42] for extended discussion of password tweaks and a model of tweaks used by users in practice to change their passwords.

The positions to be tweaked should be chosen solely on the pattern of character types in the password, and not on the specific characters in the password, otherwise the password might be easily determined as the only sweetword capable of giving rise to the given list W_i.

For chaffing-by-tail-tweaking we consider each word w_{ij} to consist of a *head* h_{ij} followed by a t-character *tail* t_{ij}. For example, "`Hungry3741`" has head "`Hungry3`" and tail "`741`". The value of t need not be the same for all users. There does not need to be any separating character between the head and the tail; the parsing of a password into password-head and password-tail need not be obvious.

If the user picks the last three characters of her password randomly, then tail-tweaking is impossible to reverse-engineer—an adversary cannot tell the password from its tweaked versions, as all tails are random. Otherwise, an adversary may be able to tell the password from the honeywords: in the following list, which is the likely password?

`57*flavors`, `57*flavrbn`, `57*flavctz` (2)

The *take-a-tail* method of the next section fixes the problem of poorly-chosen password tails by requiring new passwords to have system-chosen random password tails.

Chaffing-by-tweaking works pretty well as a legacy-UI honeyword generation method: it doesn't require users to follow any password syntax requirements, other than having enough characters in the password.

We now give another view of tweaking. Let $T(p)$ denote the class of sweetwords obtainable by tweaking p for the selected character positions.

Clearly p is in $T(p)$, since the randomly-chosen characters may be the same as the originals—in general p is always in $T(p)$. Note that $T()$ should have the property that tweaks don't change the class of a password: For any $p^* \in T(p)$, $T(p^*) = T(p)$. The classes $T(p)$ are seen to form a partition of the set of all passwords into disjoint sets of "similar" passwords.

We then define **Gen** to start with the singleton set $\{p\}$, and to repeatedly add elements chosen randomly from $T(p)$ to it until k distinct elements have been chosen. (Of course, the class $T(p)$ needs to contain at least k elements for this to work.)

The list W is then just a randomly chosen ordering of the elements in this set, and $c(i)$ is the position of p in this list.

Tweaking is perfectly flat in the case that each word in $T(p)$ was equally likely to be chosen as the password by the user—that is, $U(p^*)$ is constant for each word p^* in $T(p)$. In practice, users tend to favor some character strings more than others. (For instance, it's observed in [40] that for one-digit numbers in passwords, users choose '1' about half the time.) So perfect flatness may be difficult to achieve, and it may be helpful to bias selection of honeywords from $T(p)$ toward those most likely to be chosen by users.

4.1.2 Chaffing-with-a-password-model

Our second method generates honeywords using a probabilistic model of real passwords; this model might be based on a given list L of thousands or millions of passwords and perhaps some other parameters. (Note that generating honeywords solely from a *published* list L as honeywords is not in general a good idea: such a list may also be available to the adversary, who could use it to help identify honeywords.) Unlike the previous chaffing methods, this method does not necessarily need the password in order to generate the honeywords, and it can generate honeywords of widely varying strength.

Here is a list of 19 honeywords generated by one simple model (see Appendix for details):

```
kebrton1          02123dia
a71ger            forlinux
1erapc            sbgo864959
aiwkme523         aj1aob12
9,50PEe]KV.0?RIOtc&L-:IJ"b+Wol<*[!NWT/pb
xyqi3tbato        a3915
#NDYRODD_!!       venlorhan
pizzhemix01       dfdhusZ2
sveniresly        'Sb123
mobopy            WORFmgthness
```

Note the presence of one "tough nut," a very hard (length 40) password that the adversary will be unable to crack.

See Weir et al. [40] for a presentation of an interesting alternative model for passwords, based on probabilistic context-free grammars.

Modeling syntax. Bojinov et al. [6] propose an interesting approach (based on [40]) to chaffing-with-a-password-model in which honeywords are generated using the same syntax as the password. (Note that with this method, unlike the one above, honeywords *do* depend on the password.) In their scheme, the password is parsed into a sequence of "tokens," each representing a distinct syntactic element—a word, number, or set of special characters. For example, the password

`mice3blind`

might be decomposed into the token sequence $\mathbf{W}_4 \,|\, \mathbf{D}_1 \,|\, \mathbf{W}_5$, meaning a 4-letter word followed by a 1-digit number and then a 5-letter word. Honeywords are then generated by replacing tokens with randomly selected values that match the tokens. For example, the choice $\mathbf{W}_4 \leftarrow$ "gold," $\mathbf{D}_1 \leftarrow$ '5′, $\mathbf{W}_5 \leftarrow$ "rings" would yield the honeyword

`gold5rings`.

Replacements for word tokens are selected from a dictionary provided as input to the generation algorithm. Further details are given in [6]

4.1.3 Chaffing with "tough nuts"

One might also like to have some honeywords that are *much harder* to crack than the average—so much so that they would probably never be cracked by an adversary. (These "honeywords" might not even be passwords; these "honeyword hashes" might just be long, e.g., 256-bit, random bitstrings.) So, the adversary would not then (as we have been assuming) be faced with a completely broken list of sweetwords, but rather only a partial list. There may possibly be some uncracked hashes (represented by '$\boxed{?}$' here) still to work on, with the correct password

possibly among them. For example, what should the adversary do with the following list?

`gt79, tom@yahoo,` ? `, g*7rn45, rabid/30frogs!,` ?

Having some "tough nuts" among the honeywords might give the adversary additional reason to pause before diving in and trying to log in with one of the cracked ones. To ensure that the adversary can not tell whether the password itself lies among the set of "tough nuts," both the *positions* and the *number* of "tough nuts" added as honeywords should be random.

4.2 Modified-UI password changes

We now propose another method, "*take-a-tail*," which utilizes a modified UI for password-changes; the password-change UI is just a slight variant of the standard one.

The take-a-tail method is identical to the chaffing-by-tail-tweaking method, except that the tail of the new password is now *randomly chosen by the system*, and required in the user-entered new password.

That is, the password-change UI is changed from:

```
Enter a new password:
```

to something like:

```
Propose a password: •••••••
Append '413' to make your new password.
Enter your new password: ••••••••••
```

Thus, if the user proposes "`RedEye2`," his new password is "`RedEye2413`." This is a *very* simple change to the UI, and shouldn't require any user training. The required tail ("`413`" in this example) is randomly and freshly generated for each password change session. It could even be by chance the same as the tail of the user's previous password. (The login routine will normally prevent the user from trying to use his old password as his new password; the system might generate a new, different tail in this case.)

Once the password has been determined, the system can generate honeywords in same manner as chaffing-by-tail-tweaking.

With take-a-tail the head of the password is chosen by the user, thus increasing memorability, while the password tail is picked randomly to ensure that the password and honeyword generation procedure is perfectly flat.

A user might try to reset her password repeatedly to obtain a preferred tail, undermining the property of flatness. Most systems, however, prohibit frequent password changes (to prevent users from cycling through passwords and thus bypassing policies on old-password reuse).

4.3 Comparison of methods

Our proposed methods for generating honeywords have various benefits and drawbacks, as shown in Table 4.3. The hybrid method is described in Section 5.5.

5 Variations and Extensions

We now consider a few other ways of generating honeywords and some practical deployment considerations.

5.1 "Random pick" honeyword generation

We now present a modified-UI procedure that is perfectly flat. At a high level, a good way of generating a password and honeywords is to first generate the list W_i of k distinct sweetwords in some arbitrary manner (which may involve interaction with the user) and then pick an element of this list uniformly at random to be the new password; the other elements become honeywords. As an example of user involvement, we might just ask the user for k potential passwords. The value $c(i)$ is set equal to the index of (randomly chosen) password p_i in this list.

For example, the user may supply $k = 6$ sweetwords:

```
4Tniners    all41&14all   i8apickle
sin(pi/2)   \{1,2,3\}     AB12:YZ90
```

and the system could then inform the user that password $c(i) = 6$ (the last one) is her password.

The random pick method is perfectly flat, no matter how the list W_i of sweetwords was generated, since the given procedure is equivalent to choosing $c(i)$ uniformly at random from $\{1, 2, \ldots, k\}$ independent of the actual sweetwords; there is thus no information in W_i that can aid in determining $c(i)$.

It is probably a bad idea, however, to ask the user for k sweetwords. Not only is this burdensome on the user, but the user may remember and mistakenly enter a sweetword supplied by her and used by the system as a honeyword.

The random pick approach is probably better applied to a set of k sweetwords generated by an algorithmic password generator.

5.2 Typo-safety

We would also like it to be rare for a legitimate user to set off an alarm by accidentally entering a honeyword.

Honeyword method	**Flatness** (§3)	**DoS resistance** (§7.5)	**Storage cost (# of hashes)** (§5.4)	**Legacy -UI?** (§4)	**Multiple-system protection** (§7.6)
Tweaking (§4.1.1)	$(1/k)$ if U constant over $T(p)$	weak	1	yes	no
Password-model (§4.1.2)	$(1/k)$ if $U \approx G$	strong	k	yes	no
Tough nuts* (§4.1.3)	N/A	strong	k	yes	no
Take-a-tail (§4.2)	$(1/k)$ unconditionally	weak	k	no	yes
Hybrid (§5.5)	$(1/k)$ if $U \approx G$ and U constant over $T(p)$	strong	$\sqrt{k}$	yes	no

Table 1: Comparison of honeyword-generation methods. All methods can achieve excellent $(1/k)$-flatness under some conditions. By "weak" DoS (denial of service) resistance, we mean that an adversary can with non-negligible probability submit a honeyword given knowledge of the password; by "strong" DoS resistance we mean that such attack is improbable. Multiple-system protection is the property that compromise of the same user's account in different systems will not immediately reveal p_i. Finally, G denotes the probability distribution of honeywords generated by chaffing-with-a-password-model. (Thus $U \approx G$ means that these honeywords are distributed like user passwords in the view of the adversary.) The * means "tough nuts" are not useful on their own; they are best used together with other methods. The storage costs assume generation of $k-1$ honeywords. For further details, see the indicated sections.

Typos are one possible cause of such accidents, especially for tweaking methods.

With tail-tweaking, it could thus be helpful if the password tail were quite different from the honeyword tails, so a typing error won't turn the password into a honeyword. (The honeyword tails should also be quite different from each other, so the password doesn't stand out as the sweetword that is the "most different" from the others.)

One can use an error-detection code to detect typos (as for ISBN book codes). Let q denote a small prime greater than 10, such as $q = 13$. Suppose tails are three-digit numbers; let t_{ij} denote the tail of w_{ij}. Then we just require that the difference $(t_{ij} - t_{ij'})$ between any two sweetword tails is a multiple of q. This property is easy to arrange, even if (at most) one of the words is chosen arbitrarily by the user. This property allows detection of a substitution of any single digit for another, or of a transposition of two adjacent digits, in the tail.

5.3 Managing old passwords

Many password systems, particularly for government and industry users, store hashes of users' old passwords—usually the last 10, as stipulated in, e.g., [17]. When a user changes her password, she is prohibited in such systems from reusing any stored ones.

We feel strongly that a system should *not* store old passwords or their hashes. The motivation of this paper is that the hashes are frequently inadequate protection for the passwords themselves; hash functions can be inverted on weak passwords, and most passwords are pretty weak. Moreover, the reason a user may have decided to change her password is that she decided it was indeed weak, and moreover she is using it on several systems. If the system keeps around her old password, it may be placing her account on the other system(s) at risk.

A better option is to not store old passwords on a per-user basis and instead record previously used passwords across the full user population. A newly created password that conflicts with *any* password in this list may then be rejected. The list should not consist of explicitly hashed passwords, but should be represented in a more compact, efficiently checkable data structure, such as a Bloom filter, that does not reveal passwords directly [37]. The related proposal by Schechter, Herley, and Mitzenmacher [36], which relies on a similar data structure called a "count-min sketch," allows one to reject new passwords that are already in common use within a user population.

When storing old passwords is required, however, we recommend storing them in a protected module strongly isolated from the basic functionality of the computer system—perhaps in a special server. A weaker alternative helpful for achieving legacy compatability might reside in the computer system itself. In this case, the set O_i of old passwords for user u_i should be encrypted, when not in use, under a user-specific key κ_i. When the system must access O_i for a password change, it decrypts O_i. The key κ_i should not, of course, be stored in the computer system itself, but might be stored in the honeychecker, which releases κ_i to the computer system only after successful authentication by the user using current password p_i. After reading and updating O_i, the computer system then re-encrypts it and immediately erases κ_i.

Alternatively, of course, old passwords could themselves be stored with honeywords.

5.4 Storage optimization

Some honeyword generation methods, such as tweaking and take-a-tail, can be optimized to reduce their storage to little more than a single password hash. Consider tail-tweaking where the tails are t-digit numbers.

Suppose that $T(p_i)$ is of reasonable size—for example, with $t = 2$ digit tails we have $|T(p_i)| = 100$. Let $k = |T(p)|$ and let $W_i = T(p_i) = \{w_{i,1}, \ldots, w_{i,k}\}$, sorted into increasing order lexicographically.

We select a random element $w_{i,r}$ of $T(p_i)$ and store $H(w_{i,r})$ on the computer system. (We pick this element by selecting the index $r \in \{0, 1, \ldots, k-1\}$ uniformly at random.) To verify a proffered password g, the computer system computes the hash of each sweetword in $T(g)$; if one is found equal to $H(w_{i,r})$ then $w_{i,r}$ is known, so W_i can be computed, as can the position j of g in W_i. The honeychecker operates as usual: The computer system sends j to the honeychecker to check whether $j = c(i)$.

We can also handle cases where $k < |T(p_i)|$. To do so, we restrict the set of sweetwords W_i to a subset of k passwords from $T(p_i)$. Suppose here that $T(p_i)$ is sorted into increasing lexicographical order as above. Then we might choose $W_i = \{w_{i,1}, \ldots, w_{i,k}\}$ to be a set of k consecutive elements (with wraparound) from $T(p_i)$ that includes p_i. Randomly selecting $c(i) \in \{1, \ldots, k\}$ and setting $w_{i,c(i)} = p_i$ yields such a set W_i with p_i in a random position. In this case the computer system should store both $w_{i,1}$ and the index of $w_{i,1}$ in the list $T(p_i)$, so it knows exactly what the relevant segment of $T(p_i)$ is.

5.5 Hybrid generation methods

It is possible to combine the benefits of different honeyword generation strategies by composing them into a "*hybrid*" scheme.

As an example, we show how to construct a hybrid legacy-UI scheme that combines chaffing-by-tweaking-digits with chaffing-with-a-password-model. We assume a password-composition policy that requires at least one digit, so that tweaking digits is always possible.

Here is a simple hybrid scheme:

1. Use *chaffing-with-a-password-model* on user-supplied password p to generate a set of a (≥ 2) seed sweetwords W', one of which is the password. Some seeds may be "tough nuts."
2. Apply *chaffing-by-tweaking-digits* to each seed sweetword in W' to generate b (≥ 2) tweaks (including the seed sweetword itself). This yields a full set W of $k = a \times b$ sweetwords.
3. Randomly permute W. Let $c(i)$ be the index of p such that $p = w_{c(i)}$, as usual.

Note the importance of the ordering of steps 1 and 2. The alternative approach of tweaking first and then chaffing-with-a-password-model would likely reveal p to an adversary as the sole tweaked password.

As an example, suppose we have $a = 3$, $b = 4$, and $k = 12$. The list W_i might look as follows:

`abacad513`	`snurfle672`	`zinja750`
`abacad941`	`snurfle806`	`zinja802`
`abacad004`	`snurfle772`	`zinja116`
`abacad752`	`snurfle091`	`zinja649`

A convenient choice of parameters is $a = b = \sqrt{k}$. We assume this choice in describing the properties of the hybrid scheme; concretely, $a = b = 10$ is a possible choice, given that $T(p) \geq 10$.

To detect a DoS (denial of service) attack against this scheme (see Section 7.5) we might disregard submission of honeywords in $T(p)$, and raise an alarm for all other $k - b$ honeywords. DoS attacks, then, are very unlikely to succeed. An adversary that has stolen F and guesses a honeyword, though, will still be caught with probability $1 - 1/a = 1 - 1/\sqrt{k}$. (For $a = b = 10$, this is 90%.)

The storage costs can be optimized along the lines of Section 5.4 if desired, storing only a hashes.

Table 4.3 illustrates how this hybrid honeyword scheme inherits desirable characteristics from both component methods. Note that the scheme is flat only if both flatness

conditions given in the table hold. But if *either* condition is met, the scheme is still reasonably secure: it is ϵ-secure for $\epsilon = 1/\min(a, b) = 1/\sqrt{k}$. (For $a = b = 10$, an adversary is caught with 90% probability.)

Because the hybrid method is legacy-UI, achieves excellent flatness under reasonable assumptions, and provides resistance to DoS attacks, it is our recommended honeyword generation method.

6 Policy choices

6.1 Password Eligibility

Some words may be ineligible as passwords because they violate one or more policies regarding eligibility, such as:

- **Password syntax:** A password may be required to have a minimum length, a minimum number of letters, a minimum number of digits, and a minimum number of special characters. The initial character may be restricted. See [25] for an example of a common password-syntax (or "password complexity") policy. Cheswick [13] calls such rules "eye-of-newt" rules, because they are promulgated as if they had magical properties. We agree that today such rules seem to be more trouble than they are worth.
- **Dictionary words:** A password may not be a word in the dictionary, or a simple variant thereof.
- **Password re-use:** A password may be required to be different than any of the last r passwords of the same user, for some policy parameter r (e.g. r=10).
- **Most common passwords:** A password may not be chosen if it is on a list of the 500 most common passwords in widespread use (to prevent online guessing attacks).
- **Popular passwords:** A password may not be chosen if m or more other users in a large population of users are currently using this password.

6.2 Failover

The computer system can be designed to have a "failover" mode so that logins can proceed more-or-less as usual even if the honeychecker has failed or become unreachable. In failover mode, honeywords are temporarily promoted to become acceptable passwords; this prevents denial-of-service attacks resulting from attack on the honeychecker or the communications between the system and the honeychecker. The cost in terms of increased password guessability is small. Temporary communication failures can be addressed by buffering messages on the computer system for later delivery to and processing by the honeychecker.

6.3 Per-user policies

We can have policies that vary per-user; this is not uncommon already.

- *Honeypot accounts:* The use of honeypot accounts, as mentioned in the introduction, is a useful addition to honeywords. Such accounts can help identify theft of F and distinguish over a DoS attack (see Section 7.5). Which accounts are honeypot accounts would be known only to the honeychecker.
- *Selective alarms:* It may be helpful raise an alarm if there are honeyword hits against administrator accounts or other particularly sensitive accounts, even at the risk of extra sensitivity to DoS attacks. Policies needn't (and perhaps shouldn't) be uniform across a user population.

6.4 Per-sweetword policies

The "**Set:** i, j" command to the honeychecker could have an optional third argument $a_{i,j}$, which says what action to take if a "**Check:** i, j" command is later issued. The actions might be of the form "Raise silent alarm," "Allow login," "Allow for single login only," etc... There could be k different entries for a given user, with potentially k different policies, one per sweetword. This feature could be used, for example, with the take-a-tail strategy to note which honeywords have small edit distance to the password (e.g., a single transposition or change of character), so that user typos invoke a less severe reaction. This gives added flexibility to the policies enabled by the use of the honeychecker.

7 Attacks

This section reviews more carefully various attacks possible against the methods proposed here.

7.1 General password guessing

Legacy-UI methods don't affect how users choose passwords, so they have no beneficial effect against adversaries who try common passwords in an online guessing attack.

We do favor methods such as those proposed by Schecter et al. [36] requiring users to choose uncommon passwords.

Modified-UI methods like take-a-tail also affect the choice of password—appending a three-digit random tail to a user-chosen password effectively reduces the probability of the password by a factor of 1000.

7.2 Targeted password guessing

Personal information about a user could help an adversary distinguish the user's password from her honeywords. It is often feasible to deanonymize users, that is, ascertain their real-world identities, based on their social network graphs [27] or just their usernames [31]. Given a user's identity, there are then many ways to find demographic or biographical data about her online—by exploiting information published on social networks, for example [5].

Knowing a user's basic demographic information, specifically his/her gender, age, or nationality, is known to enable slightly more effective cracking of the user's hashed password [7, 8]. Similarly, attackers often successfully exploit biographical knowledge to guess answers to personal questions in password recovery systems and compromise victims' accounts [35]. (The hacking of Governor Sarah Palin's Yahoo! account is a well known example.) As chaffing-with-a-password-model creates honeywords independently of user's password, this method of honeyword generation may enable adversaries to target data-mining attacks against users and gain some advantage in distinguishing their passwords from their honeywords.

7.3 Attacking the Honeychecker

The adversary may decide to attack the honeychecker or its communications with the computer system.

The updates ("**Set**" commands) sent to the honeychecker need to be authenticated, so that the honeychecker doesn't incorrectly update its database.

The requests ("**Check**" commands) sent to the honeychecker also need to be authenticated, so that the adversary can't query the honeychecker and cause an alarm to be raised.

The replies from the honeychecker should be authenticated, so that the computer system doesn't improperly allow the adversary to login.

By disabling communications between the computer system and the honeychecker, the adversary can cause a failover (see Section 6.2. The computer system then either has to disallow login or take the risk of temporarily allowing login based on a honeyword and buffering messages for later processing by the honeychecker.

While our intention is that the honeychecker should be hardened and of minimalist design, the deployment of the computer system and the honeychecker as two distinct systems itself brings the usual benefits of separation of duties in enhancing security. The two systems may be placed in different administrative domains, run different operating systems, and so forth.

7.4 Likelihood Attack

If the adversary has stolen F and wishes to maximize his chance of picking p_i from W_i, he can proceed with a "likelihood attack" as follows.

We assume here that we are dealing with an approach based on generating honeywords using a probabilistic model. Let $G(x)$ denote the probability that the honeyword generator generates the honeyword x.

Similarly, let $U(x)$ denote the probability that the user picks x to be her password. (This may not be mathematically well-defined; it can be interpreted as a Bayesian prior for the adversary on such probabilities, and may or may not be user-specific.)

Let $W_i = \{w_{i,1}, \ldots, w_{i,k}\}$. The likelihood that $c(i) = j$, given W_i, is equal to

$$U(w_{i,j}) \prod_{j' \neq j} G(w_{i,j'}) = C\, R(w_{ij})$$

where

$$C = \prod_{j'} G(w_{i,j'})$$

and where

$$R(x) = U(x)/G(x)$$

is the *relative likelihood* that the user picks x compared to the honeyword generator picking x. Note that it is desirable that for all eligible x, $G(x) > 0$ (that is, the honeyword generator is capable of generating all possible words); otherwise the password may be recognizable as one the honeyword generator could not possibly have produced.

The adversary wants to maximize his likelihood of picking the password, so he will pick the one maximizing $R(w_{ij})$. This is the password that is maximally more likely to be picked by the user than to be generated by the honeyword generator. As an example, a password like

```
NewtonSaid:F=ma
```

is not very likely to be generated by a honeyword generator, but is plausibly generated by a (physics-knowledgeable) user. An adversary might easily notice this password in a set of honeywords.

In this context, a user might be well advised either to (a) choose a very strong password that the adversary will never crack, or (b) choose a password of the sort that the honeyword generator might generate. That is, don't pick a password that has "obvious structure" to a human of a sort that an automatic generator might not use. Alternatively, a generator might take as input a private, handcrafted list of such distinctive passwords for (one-time) use as honeywords; "obvious structure," then, wouldn't always signal a true password to an adversary.

The above theory about relative likelihood is incomplete: it doesn't tell an adversary what to do when only some of the sweetword hashes are solved, as happens when "tough nuts" are used.

7.5 Denial-of-service

We briefly discuss denial-of-service (DoS) attacks—a potential problem for methods such as chaffing-by-tweaking that generate honeywords by predictably modifying user-supplied passwords. (In contrast, chaffing-with-a-password-model and the hybrid scheme of Section 5.5 offer strong DoS resistance.)

The concern is that an adversary who has not compromised the password file F, but who nonetheless knows a user's password—e.g., a malicious user or an adversary mounting phishing attacks—can feasibly submit one of the user's honeywords. For example, with chaffing-by-tweaking-digits, with $t = 2$, such an adversary can guess a valid honeyword with probability $(k - 1)/99$. A *false appearance* of theft of the password file F results.

An overly sensitive system can turn such honeyword hits into a DoS vulnerability. One (drastic) example is a policy that forces a global password reset in response to a single honeyword hit. Conversely, in a system inadequately sensitive to DoS attacks, an adversary that has stolen F can guess passwords while simulating a DoS attack to avoid triggering a strong response. So a policy of appropriately calibrated response is important. Reducing the potency of DoS attacks can help.

Mitigating DoS attacks To limit the impact of a DoS attacks against chaffing-by-tweaking, one possible approach is to select a relatively small set of honeywords randomly from a larger class of possible sweetwords. For example, we might use take-a-tail with a three-digit tail ($t = 3$), yielding $|T(p_i)| = 1000$. Setting $k = 20$, then, means randomly picking $k - 1 = 19$ honeywords from $T(p_i)$. Knowing the correct password p_i only gives an adversary (or malicious user) a chance of $(k-1)/1000 \approx 0.02$ of hitting a honeyword in this case, greatly reducing her ability to trigger an alarm. The vast majority ($\approx 98\%$) of her attempts will be passwords in $T(p_i)$, but *not* in W_i.

7.6 Multiple systems

As users commonly employ the same password across different systems, an adversary might seek an advantage in password guessing by attacking two distinct systems, system A and system B—or multiple systems, for that matter. We consider two such forms of attack, an "intersection" attack and a "sweetword-submission" attack.

Intersection attack. If a user has the *same* password but *distinct* sets of honeywords on systems A and B, then an adversary that compromises the two password files learns the user's password from their intersection. (Of course, without honeywords, an attacker learns the password by compromising *either* system.) We would aim instead that an intersection attack against systems using honeywords offer an adversary no advantage in identifying the password on either system.

Our favored approach, in the case where management of multiple systems is of concern, would be the take-a-tail generation approach of Section 4.2 on each system. Although the compromise of the password-hash file F would reveal the password-head to an adversary, the user's sugar would be independently and randomly generated on each system.

A significant advantage for system-chosen tails is that it becomes very likely that the user's password will be different on different systems, even if the user chooses the same password-head. This should increase overall security, as users can no longer use the exact same password on each system.

The burden on memory is increased, but in our judgment this increase is well worth the cost—too many systems are compromised by having user passwords cracked on other systems.

Note that this ability of ensuring that a user has different passwords on different systems is achieved *without* coordination between the systems—it is a statistical guarantee. An adversary who discovers a user's password on system A may still be caught trying to a login with a honeyword on system B.

The definition of flatness for a generation procedure **Gen** can be extended to handle this case: An adversary gets the outputs $(W_i, c(i))$ from multiple invocations of **Gen**, and then has to guess $c(i)$ for an additional, last invocation of **Gen** in which the adversary sees W_i (but not $c(i)$). Even with this additional information, the adversary's chance of guessing $c(i)$ should be at most $1/k$.

The take-a-tail method also protects users against an adversary who monitors changes in the password-hash file F *over time*. (In contrast, chaffing-with-a-password-model doesn't help much if the password is changed only slightly, but entirely new chaff is chosen.)

We note too that if system A and B make use of the same tweaking method, they will automatically generate identical sweetwords for a given password. For example, if both systems employ chaffing-by-tweaking-digits with $t = 1$ digit, they will both output the same ten tweaks (including the original password) as sweetwords. Under a suitable parameterization, our recommended hybrid scheme, which includes tweaking, may generate partially intersecting honeyword sets across systems.

Emerging laws in the United States, such as the Cyber Intelligence Sharing and Protection Act (CISPA)[28], encourage the exchange of cybersecurity intelligence across organizations. Honeyword generation methods might be shared in this context to help prevent intersection attacks.

Sweetword-submission attack. It is possible, as above, that the user has the same password on systems A and B, but distinct corresponding sets of honeywords; alternatively, system B may not use honeywords at all. In either case, an adversary that compromises the password file on system A can submit the user's sweetwords as password guesses to system B without special risk of detection: To system B, system A's honeywords will be indistinguishable from any other incorrect passwords. We call this a "sweetword-submission" attack.

If system B uses honeywords, then, the same countermeasures to intersection attacks—particularly take-a-tail—can also provide resistance to sweetword-submission attacks. Even if system B doesn't use honeywords, though, it still benefits somewhat from the presence of honeywords on system A in a sweetword-submission attack: The adversary will have to submit more than $k/2$ sweetwords on average before successfully guessing the password (assuming system A uses a flat **Gen**).

If the adversary compromises a system B that doesn't use honeywords, he can of course learn the user's password and impersonate her on system A, even if system A does use honeywords.

8 Related Work

Password strength. The current, state-of-the-art heuristic password cracking algorithm, due to Weir et al., is based on probabilistic, context-free grammars [40]. In a recent study, Kelley et al. [23] characterize the vulnerability of user-generated passwords to Weir-style cracking attacks under various password-composition policies. One such policy is a common, weak one dubbed "basic8," in which users are instructed, "Password must have at least 8 characters." One billion guesses suffice to crack 40.3% of such passwords. Recent work shows that cracking speeds for some hash functions (e.g., MD5) can approach three-billion guesses per second on a single graphical-processing unit (GPU); see, e.g., Table 15 of [3]. Also in recent work, Bonneau develops a framework to assess the strength of passwords (and other user secrets). Based on study of published password corpora, including one representing 70 million Yahoo! users, he estimates that a majority of passwords have little more than 20 bits of effective entropy against an optimal attacker [7, 8].

Together, these results underscore the weakness of current password protections even with the use of sound practices, such as salting. There is good reason to believe that many systems don't even make use of salt [29]. While the reason for this lapse is unclear, we emphasize that honeywords may be used with or without salt (and even in principle with or without hashing).

Bonneau and Preibusch [9] offer an excellent survey of current password management practices on popular web sites, including password composition requirements and advice to users, account lockout policies, and update and recovery procedures. Herley and van Oorschot [21] argue that use of passwords will persist for many years, and highlight key research questions on how to create strong passwords and manage them effectively.

Password strengthening. The take-a-tail method may be viewed as a variant on previously proposed password strengthening schemes. Forget et al. [18], randomly interleave system-generated characters into a password. The user may request a reshuffling of these characters until she obtains a password she regards as memorable. The extra characters here are essentially sugar. (Rejected or unpresented interleavings could serve as honeywords.) Houshmand and Aggarwal [22] recently proposed a related system that applies small tweaks to user-supplied passwords to preserve memorability while adding strength against cracking, specifically via [40]. Various schemes, e.g., PwdHash [34], have also been proposed to strengthen passwords within password managers.

Password storage and verification. There are stronger approaches than honeywords for splitting password-related secrets across servers. Some proposed and commercialized methods employ distributed cryptography to conceal passwords fully in the event of a server breach [11, 12, 15]. While such methods are perferrable to honeywords where practical, they require substantial changes to password verification systems and, ideally, client-side support as well. Honeywords may be seen as a stepping stone to such approaches.

Password-authenticated key-exchange methods, such as the Secure Remote Password Protocol (SRP)[4], provide another approach towards verifying that a remote party knows a correct password. However, the remote party must have a trusted computer to perform the necessary mathematical operations. If successful, both parties end up with the same secret key, which they may use to encrypt and/or authenticate further communications.

Decoys. The use of decoy resources to detect security breaches is an age-old practice in the intelligence community. Similarly, honeypots are a stock-in-trade of computer security. A survey of the use of honeypots and related decoys and of pertinent history and theory may be found in [14]. It is a common industry practice today to deploy "honeytokens," bogus credentials such as credit card numbers [38], to detect information leakage and degrade the value of stolen credentials. (Honeywords could likewise reduce the value of stolen passwords.) Similarly, fabricated or decoy files have been proposed as traps to detect intrusion [41] and insider attacks [10].

Honeywords also bear some resemblance to duress codes, plausible-looking but invalid secrets that users may submit to trigger a silent alarm.[5] A related idea are "collisionful" hash functions [2, 4]; these yield hash values with multiple, feasibly computed pre-images, thus creating ambiguity as to which pre-image is correct.

Most closely related to our proposed use of honeywords is the Kamouflage system of Bojinov et al. [6]. The setting in that work differs from ours, though. Kamouflage aims to protect a user's list of passwords in a client-side password manager against misuse should the user's device (e.g., laptop or tablet) be stolen or otherwise compromised. Kamouflage conceals the correct password list within a set of decoy lists, which contain honeywords created using the scheme described in Section 4.1.2. Password-consuming servers need not be aware of Kamouflage deployment. (The authors do note, though, that servers might store some honeywords to facilitate detection of compromise.)

[4] `http://en.wikipedia.org/wiki/Secure_Remote_Password_protocol`

[5] `http://en.wikipedia.org/wiki/Duress_code`

9 Open Problems

This paper is just an initial stab at the issues surrounding the use of honeywords to protect password hash files; many open questions remain, such as:

- How should an adversary act optimally when some "tough nuts" are included among the honeywords?
- What is the best way to enforce password-reuse policies?
- Can the password models underlying cracking algorithms (e.g., [40]) be easily adapted for use in chaffing-with-a-password-model?
- How effective is targeted password guessing in distinguishing passwords from honeywords?
- How can a honeyword system best be designed to withstand active attacks, e.g., code modification, of the computer system (or the honeychecker)?
- How well can targeted attacks help identify users' passwords for particular honeyword-generation methods?
- How user-friendly in practice is take-a-tail?

10 Discussion and Conclusion

Someone who has stolen a password file can brute-force to search for passwords, even if honeywords are used.

However, the big difference when honeywords are used is that a successful brute-force password break does *not* give the adversary confidence that he can log in successfully and undetected.

The use of an honeychecker thus forces an adversary to either risk logging in with a large chance of causing the detection of the compromise of the password-hash file F, or else to attempt compromising the honeychecker as well. Since the honeychecker's interface is extremely simple, one can more readily secure the honeychecker.

The use of honeywords may be very helpful in the current environment, and is easy to implement. The fact that it works for *every* user account is its big advantage over the related technique of honeypot accounts.

One could imagine using an auxiliary server in other ways in support of password-based authentication. However, the architecture proposed here is clean and simple,

reverts to current practice if auxiliary server files are compromised, and is even robust against auxiliary server failure (if one allows logins with honeywords).

Honeywords also provide another benefit. Published password files (e.g., one stolen from LinkedIn [30]) provide attackers with insight into how users compose their passwords. Attackers can then refine their models of user password selection and design faster password cracking algorithms [23]. Thus every breach of a password server has the potential to improve future attacks. Some honeyword generation strategies, particularly chaffing ones, obscure actual user password choices, and thus complicate model building for would-be hash crackers. It may even be useful to muddy attacker knowledge of users' composition choices intentionally by drawing some honeywords from slightly perturbed probability distributions.

Despite their benefits over common methods for password management, honeywords aren't a wholly satisfactory approach to user authentication. They inherit many of the well known drawbacks of passwords and something-you-know authentication more generally. Eventually, passwords should be supplemented with stronger and more convenient authentication methods, e.g., [16], or give way to better authentication methods completely, as recently predicted by the media [24, 39].

In the meantime, honeywords are a simple-to-deploy and powerful new line of defense for existing password systems. We hope that the security community will benefit from their use. (See our note below on IP.)

Acknowledgments

The second author thanks Andrew and Erna Viterbi for their support. The authors also thank Ben Adida, Sudhir Aggarwal, Ross Anderson, Mihir Bellare, Jeremiah Blocki, Dan Boneh, Joe Bonneau, Bill Cheswick, Burt Kaliski, Silvio Micali, Mike Mitzenmacher, Benny Pinkas, Raluca Ada Popa, Tom Ristenpart, Phil Rogaway, Tomas Sander, Stuart Schechter, Bruce Schneier, Eugene Spafford, Matt Weir, Nickolai Zeldovich, Yinqian Zhang, and their colleagues in RSA Labs for numerous helpful comments and suggestions regarding this note.

Note on Intellectual Property (IP)

We do not know of any patent or other restrictions on the use of the ideas proposed here. We have not filed for any such patents (and will not). The authors place into the public domain any and all rights they have on the use of the ideas, methods or systems proposed here. As far as we are concerned, others may freely make, use, sell, offer for sale, import, embed, modify, or improve the ideas, methods, or systems described in this note. There is no need to contact us or ask our permission in order to do so. We ask only that this paper be cited as appropriate.

References

[1] A. Evans, Jr., W. Kantrowitz, and E. Weiss. A user authentication scheme not requiring secrecy in the computer. *Commun. ACM*, 17(8):437–442, August 1974.

[2] R. J. Anderson and T.M.A. Lomas. On fortifying key negotiation schemes with poorly chosen passwords. *Electronics Letters*, 30(13):1040–1041, 1994.

[3] M. Bakker and R. van der Jagt. GPU-based password cracking. Technical report, Univ. of Amsterdam, 2010.

[4] T. A. Berson, L. Gong, and T.M.A. Lomas. Secure, keyed, and collisionful hash functions. Technical Report SRI-CSL-94-08, SRI International Laboratory, 1993 (revised 2 Sept. 1994).

[5] L. Bilge, T. Strufe, D. Balzarotti, and E. Kirda. All your contacts are belong to us: automated identity theft attacks on social networks. In *WWW*, pages 551–560, 2009.

[6] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh. Kamouflage: loss-resistant password management. In *ESORICS*, pages 286–302, 2010.

[7] J. Bonneau. *Guessing human-chosen secrets*. PhD thesis, University of Cambridge, May 2012.

[8] J. Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *IEEE Symposium on Security and Privacy*, pages 538–552, 2012.

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

[10] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *SecureComm*, pages 51–70, 2009.

[11] J. Brainard, A. Juels, B. Kaliski, and M. Szydlo. A new two-server approach for authentication with short secrets. In *USENIX Security*, pages 201–214, 2003.

[12] J. Camenisch, A. Lysyanskaya, and G. Neven. Practical yet universally composable two-server password-authenticated secret sharing. In *ACM CCS*, pages 525–536, 2012.

[13] William Cheswick. Rethinking passwords. *Comm. ACM*, 56(2):40–44, Feb. 2013.

[14] F. Cohen. The use of deception techniques: Honeypots and decoys. In H. Bidgoli, editor, *Handbook of Information Security*, volume 3, pages 646–655. Wiley and Sons, 2006.

[15] EMC Corp. RSA Distributed Credential Protection. http://www.emc.com/security/rsa-distributed-credential-protection.htm, 2013.

[16] A. Czeskis, M. Dietz, T. Kohno, D. Wallach, and D. Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *ACM CCS*, pages 404–414, 2012.

[17] Defense Information Systems Agency (DISA) for the Department of Defense (DoD). Application security and development: Security technical implementation guide (STIG), version 3 release 4, 28 October 2011.

[18] A. Forget, S. Chiasson, P. C. van Oorschot, and R. Biddle. Improving text passwords through persuasion. In *SOUPS*, pages 1–12, 2008.

[19] C. Gaylord. LinkedIn, Last.fm, now Yahoo? don't ignore news of a password breach. *Christian Science Monitor*, 13 July 2012.

[20] D. Gross. 50 million compromised in Evernote hack. *CNN*, 4 March 2013.

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

[22] S. Houshmand and S. Aggarwal. Building better passwords using probabilistic techniques. In *ACSAC*, pages 109–118, 2012.

[23] 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 (SP)*, pages 523–537, 2012.

[24] O. Kharif. Innovator: Ramesh Kesanupalli's biometric passwords stored on devices. *Bloomberg Businessweek*, 28 March 2013.

[25] Microsoft TechNet Library. Password must meet complexity requirements. Referenced March 2012 at http://bit.ly/YAsGiZ.

[26] R. Morris and K. Thompson. Password security: a case history. *Commun. ACM*, 22(11):594–597, November 1979.

[27] A. Narayanan and V. Shmatikov. De-anonymizing social networks. In *IEEE Symposium on Security and Privacy (SP)*, pages 173–187, 2009.

[28] U.S. House of Representatives. H.R. 624: The Cyber Intelligence Sharing and Protection Act of 2013. 113th Cong., 2013.

[29] B.-A. Parnell. LinkedIn admits site hack, adds pinch of salt to passwords. *The Register*, 7 June 2012.

[30] I. Paul. Update: LinkedIn confirms account passwords hacked. *PC World*, 6 June 2012.

[31] D. Perito, C. Castelluccia, M. A. Kaafar, and P. Manils. How unique and traceable are usernames? In *Privacy Enhancing Technologies*, pages 1–17, 2011.

[32] N. Perlroth. Hackers in China attacked The Times for last 4 months. *New York Times*, page A1, 31 January 2013.

[33] G. B. Purdy. A high security log-in procedure. *Commun. ACM*, 17(8):442–445, August 1974.

[34] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J.C. Mitchell. Stronger password authentication using browser extensions. In *USENIX Security*, 2005.

[35] S. Schechter, A. J. B. Brush, and S. Egelman. It's no secret. measuring the security and reliability of authentication "secret" questions. In *IEEE Symposium on Security and Privacy (SP)*, pages 375–390, 2009.

[36] S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: a new approach to protecting passwords from statistical-guessing attacks. In *USENIX HotSec*, pages 1–8, 2010.

[37] E. Spafford. Observations on reusable password choices. In *USENIX Security*, 1992.

[38] L. Spitzner. Honeytokens: The other honeypot. Symantec SecurityFocus, July 2003.

[39] T. Wadhwa. Why your next phone will include fingerprint, facial, and voice recognition. *Forbes*, 29 March 2013.

[40] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. In *IEEE Symposium on Security and Privacy (SP)*, pages 162–175, 2009.

[41] J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. In *Information Assurance Workshop*, pages 116–122, 2004.

[42] Y. Zhang, F. Monrose, and M. K. Reiter. The security of modern password expiration: an algorithmic framework and empirical analysis. In *ACM CCS*, pages 176–186, 2010.

Appendix. Chaffing with a password model

This appendix describes a simple way to generate honeywords; this method is one way to implement *chaffing-with-a-password-model*. It is just a simple example of such a probabilistic model; better models certainly exist.

This method uses a list L of sample passwords. Honeywords are not taken from this file; rather, this list is used as an aid to generate plausible-looking honeywords. This list is intended to look like plausible passwords users might generate; it is not intended to be a list of "high-strength" passwords.

Thus, this honeyword generation scheme is qualitatively different than the process of tweaking a password to generate a new password: it is OK for a honeyword to be much weaker than the true password in an attempt to trick the adversary. However, it should not be so weak that high-probability (i.e. very common) passwords are generated, as this would cause an online guessing attack to hit honeywords.

A simple model for generating a single honeyword: The password list L is initialized to a list of many thousands of real passwords, as well as some truly random passwords of varying lengths.

A "tough nut" is generated with some fixed probability (e.g. 8%).

Otherwise a honeyword is generated as follows. A target length d is first determined by picking a random password w from L and measuring its length.

Let the characters of the new password be denoted c_1, c_2, ..., c_d. These are determined sequentially. The first character c_1 is just the first character w_1 of w. Let $w = w_1 w_2 \ldots, w_d$.

To determine the jth character of c, for $j = 2, 3, \ldots, d$:

- With probability 0.1, replace w by a randomly chosen password in L of length t. Then let $c_j = w_j$.
- Else with probability 0.4, replace w by a randomly chosen password in L of length t that has $w_{j-1} = c_{j-1}$. Then let $c_j = w_j$.
- Else with probability 0.5, let $c_j = w_j$.

If the honeyword c is ineligible (§6.1), then begin again to generate c (excluding the tough-nut option this time).

Write us for python code implementing this model.