Kamouflage: Loss-Resistant Password Management

Abstract

We introduce Kamouflage: a new architecture for building theft-resistant password managers. An attacker who steals a laptop or cell phone with a Kamouflage-based password manager is forced to carry out a considerable amount of online work before obtaining any user credentials. We implemented our proposal as a replacement for the built-in Firefox password manager, and provide performance measurements and the results of user studies to evaluate the effectiveness of our approach. We expect Kamouflage to become the standard architecture for password managers on mobile devices.

1 Introduction

All modern web browsers ship with a built-in password manager to help users manage the multitude of passwords needed for logging into online accounts. Most existing password managers store passwords encrypted using a master password. Firefox users, for example, can provide an optional master password to encrypt the password database. iPhone users can configure a PIN to unlock the iPhone before web passwords are available.

Suppose the browser runs on a mobile device such as a laptop or cell phone. An attacker who steals the device will obtain the password database encrypted under the master password. He or she can then run an offline dictionary attack using standard tools [14, 12] to recover the master password and then decrypt the password database. Once decrypted, the attacker can login as the user to any web site where the user has an account. Note that the attack is purely offline. We examined a long list of available password managers, both for laptops and smartphones, and found that all of them are vulnerable to such offline attacks following the theft of the device.

Several potential defenses quickly come to mind. First, one can use salts and slow hash functions to slow down a dictionary attack on the master password. Unfortunately, these methods do not prevent dictionary attacks [4]; they merely slow them down. Rainbow tables [12], implemented in standard tools [14], can be very effective at circumventing these methods, unless great care is used in deploying them [1].

Another potential defense is to use a password generator [15] rather than a password manager. A password generator generates site-specific passwords from a master password. Users, however, want the ability to choose memorizable passwords so that they can easily move from one machine to another; this may not be that easy with the meaningless strings typically issued by password generators. Indeed, if a password generator is not ubiquitous, and sadly none are, then majority of users will not use it.

Yet another defense is to rely on volume encryption, such as Windows BitLocker which uses special hardware (a TPM) to manage the volume encryption key. An attacker who steals the laptop will be unable to decrypt the volume, unless he or she can extract the key from the TPM. This solution, however, cannot be used on devices that have no TPM chip, such as smartphones and some laptops; it also suffers from portability problems. Clearly, we prefer a solution that does not rely on special hardware. Other potential defenses and their drawbacks are discussed in Section 6.

Our contribution. We propose a new architecture for building theft-resistant password managers called *Kamouflage*. Our goal is to force the attacker to mount an *online* attack before he can learn any user passwords. Since online attacks are easier to block (e.g. by detecting multiple failed login attempts on the user's account) we make it harder for an attacker to exploit a stolen password manager.

The basic Kamouflage idea is very simple. While standard password managers store a single set S of user passwords, Kamouflage stores the set S along with n decoy sets $S_1, \ldots, S_n$. A typical value for n is $n = 10,000$, but larger or smaller values are acceptable. The challenge for Kamouflage is to generate decoy sets that are statistically indistinguishable from the real set. If we succeed, then an attacker who steals the device will be forced to perform, on average, $n/2$ online login attempts before recovering the user's credential. Web sites can detect these failed login attempts and react accordingly. The Kamouflage architecture moreover provides a supplemental proactive defense strategy with the participation of the web sites: it allows the user's machine to register a few decoy passwords (say 10) to web sites where the user has an account. If a web site ever sees a login attempt with a decoy password, the site can immediately block the user's account.

Assuming each user has about 100 passwords and each password takes about 10 bytes, the decoy sets will take about 10MB of storage, which is roughly the size of three MP3 files, and is negligible for the latest laptops and smartphones.

As mentioned above, the main challenge is to generate decoy sets that are, by themselves, indistinguishable from the real set. We list a few difficulties that we had to deal with:

- Human-memorable passwords: Since decoys must look like human memorizable passwords, we need a model for generating such passwords. To build such a model we performed a study of human passwords, as discussed in Section 2. We also took advantage of previous work on this topic [19, 11, 18, 17]. Interestingly, most previous work studied this problem for the purpose of speeding up dictionary attacks: an attacker sorts the dictionary by the likelihood that a word is human generated. Here we give the first "positive" application for these password models, namely hiding a real password in a set of decoys.

- Related passwords: Since humans tend to pick related passwords, Kamouflage must pick decoy sets that are both chosen according to a password model and related to each other as real users tend to do. Our password study in Section 2 develops a model for *password sets* that mimics human behavior.

- Relation to master password: In some cases it makes sense to encrypt the password database using a master password. Unfortunately, our experiments found that users tend to pick master passwords that are themselves related to the passwords being protected. Hence, we had to develop an encryption scheme that cannot be used to rule out decoy sets. We present our approach in Section 5.1.

- Site restrictions: Finally, different sites have different password requirements. Some have lax requirements while others have strict ones requiring both alphanumeric and non alphanumeric characters. When generating decoy sets we have to make sure that all passwords in the decoy set are consistent with the corresponding site policy.

We built a Kamouflage prototype as a drop-in replacement for the Firefox password manager. We give performance numbers in Section 4.2 where we show that Kamouflage has little impact on user experience.

2 How users choose passwords

In this section we present the results of our quantitative and qualitative experiments on user password behavior that we use to support our assumptions that users are uncomfortable with random independent passwords, and as a result tend to select predictable and related passwords across multiple sites. In order to test this and related hypotheses, we conducted two experiments: a qualitative one and quantitative one.

User survey. The qualitative experiment was a survey conducted with undergraduate and post-doctoral students at a major university campus[1]. The goal of the survey was not to learn people's passwords, but to elicit their approach to the "password problem". The survey was administered as an in-person questionnaire, and completed by 87 individuals: 30 CS undergraduate students, and 57 post-doctoral researchers from various fields including biology, chemistry and psychology. 33% of the respondents were female, and over 61% were between 26 and 35 years old. It can be said that our subjects represent the "worst case" scenario for an attacker, in the sense that they were highly educated (more than 60% having a PhD) and use the internet everyday.

In the survey we asked direct questions about users' password habits. 81% of our subjects admitted to reusing the same password on many websites, thereby supporting our hypothesis on password reuse. This hypothesis is also supported by the number of passwords used by our subjects: 65% of the sample reported using at most 5 passwords, and 31% reported using between 6 and 10 passwords. In a separate question, 67.5% of the sample admitted selecting related but not necessarily identical passwords across sites, which stresses the importance of taking password similarities into account in our password manager design. We also found that 82.7% of our subjects reported that they did not password-protect their smartphone, despite the presence of private data on the phone.

Data analysis. Our second experiment, the quantitative one, was the analysis of an actual real-world password database, namely that of the developer web site `phpbb.com`, recently leaked into the public. This particular database is interesting for two reasons: first, the `phpbb.com` web site does not enforce any password requirements, so users were free to use whatever they want; second, this is the largest password database ever leaked with about 343 000 passwords. The passwords were not listed in the clear, but as hashes created by one of two schemes: the first scheme is a simple MD5 hash and corresponds to the old encryption used in `phpbb2`; the second scheme, used in `phpbb3`, uses 2048 calls to MD5 with a salt. Using a cluster of computers over several months, we where able to recover 241 584 cleartext passwords, or 71% of the full database. Since we did not have direct access to the passwords as plaintext, our recovery process induces a *selection bias* towards easier passwords; however, our conclusions remain significant, owing to the fact that we managed to recover such a large fraction of the entire database.

The observed password lengths range from 1 character (147 entries) to 24 (2 entries), and peak at 6 characters (75 275 entries), then 8 (61 177 entries), then 7 (47 106 entries). This observation supports the notion that most people do not select long passwords, even when they are allowed to do so. We also tested the hypothesis that people use dictionary words in their passwords, by comparing the recovered passwords with the dictionary created by *openwall* to work in conjunction with the famous cracker "John the ripper" [14]. This dictionary contains around 4 millions words.

[1]anonymized for submission

Overall, 142 471 passwords, or 59% of the recovered set, were related to dictionary words. More precisely, 94 451 passwords were actual dictionary entries, while another 48 020 passwords consisted of a dictionary word with additional digits or characters. 16% of the passwords have digits as prefix, 36% of the passwords have digits as suffix, and fewer than 1% of the passwords have digits in the middle. Once again, the placement of digits supports our assumption that the "shape" of passwords preferred by most users is predictable and recognizable. Our experiments also confirm the observations about password shape made in [19].

3 Threat model

We begin by describing our threat model. In the next section we present our architecture for a theft-resistant password manager.

The basic threat model. We consider an attacker who obtains a device, such as a laptop or smartphone, that contains user data stored in a password manager. The password manager stores user passwords for online sites (banking, shopping, corporate VPN) and possibly personal data such as social security and credit card numbers. The attacker's goal is to extract the user's data from the password manager.

If the password manager stores the data in the clear then the attacker's goal is straight forward. We therefore assume that the password manager encrypts the data using a master password. On Windows, for example, password managers often call the Windows DPAPI function `CryptProtectData` to encrypt data using a key derived from the user's login credentials. In this case we treat the user's login password as the master password. Other passwords managers, such as the one in Firefox, let the user specify a master password separate from the login password.

We already explained that an attacker can defeat existing password managers by an offline dictionary attack on the master password. Hence, simply encrypting with a master password cannot result in a secure password manager (unless the master password is quite strong). We capture this intuition in our threat model by saying that the attacker has "infinite" computing power and can therefore break the encryption used by the password manager. This is just a convenience to capture the fact that encryption with a human password as the key is weak.

In our basic threat model we assume that the attacker has no side information about the user being attacked. That is, the attacker does not know the user's hobbies, age, etc. The only information known to the attacker are the bits that the password manager stores on disk. We relax this assumption in an extended threat model discussed below.

We measure security of a password manager by the expected number of online login attempts the attacker must try before he obtains some or all of the data stored in the password manager. Note that the attacker can try login attempts at different and unrelated sites. We count the expected total number of attempts across all sites before some sensitive data is exposed.

For this approach to have value, we must assume that all web sites implement some defense against online dictionary attacks against a specific username. If some site does not, then the attacker can mount an online dictionary attack against that site and potentially expose the user's password at all sites. If some web sites have no online attack protections, the password manager can compartmentalize passwords into groups to ensure that more sensitive passwords are protected in all cases, even if the less important ones are cracked successfully.

Extended model: taking computing time into account. In the basic model we ignored the attacker's computing time needed to break the encryption of the password manager. In the extended model we measure security more accurately. That is, security is represented as a pair of numbers: (1) expected offline computing time; and (2) expected number of online login attempts until information stored in the password manager is exposed. This will allow us to more accurately compare schemes.

In the extended model we can employ encryption with a master password to slow down the attacker. The challenge is to design an encryption scheme that does not reduce the amount of online work that the attacker needs to do. Once we allow for encryption, we can relax our basic-model assumptions and allow the attacker to have side information beyond the device data. As we will see, we can offer some security even if the attacker has intimate knowledge of the victim.

Non-threats. In this paper we primarily focus on extracting data from long-term storage. We do not discuss attacks against the device or its operator while in active use (such as shoulder snooping and key loggers), and similarly omit hardware-based side-channel attacks, which can come in many guises, based, e.g., on electromagnetic emanations ("van Eck phreaking") and capacitive data retention ("cold-boot attacks"), to name but a few possibilities. In particular, we assume the device is turned off at the time it is lost or stolen, and hence no sensitive information is being computed or stored, other than the password-manager database itself (and older versions thereof).

Similarly, we do not address social engineering attacks such as phishing. While phishing can be an effective way to steal user passwords, it is not the only way. Our goal is to provide security in case of device theft or loss, which is an orthogonal problem to phishing.

4 Architecture

At any point in time, the password management software maintains a large collection of plausible password sets; exactly one of these sets is the real one, and the rest are *decoys*. Figure 1 illustrates the storage format. Apart from passwords, all other site information is kept in the clear. That is, an attacker looking at the database knows which sites the user has passwords for—she just has no idea what the passwords are. When the real user launches his web browser, he is prompted for the master password (MP) which is then cryptographically transformed to obtain the index in the collection of the real password set.

Database operations. The following are the core operations that a password database needs to support:

- **Add a new password to the database.** In our design, this amounts to adding a new password to each password set in the collection. The real password set gets the user-supplied value, while decoy sets get auto-generated entries influenced by the true one.

  Optionally, this step can be optimized by pre-populating all password sets with a number of passwords, so that the only change required is in the real password set. If, for example, at initialization time all sets are generated to hold 100 passwords, the decoy sets, as well as the real one are populated with auto-generated, plausible values. Whenever the real password set is modified, its completion to 100 entries is regenerated, taking into account the new

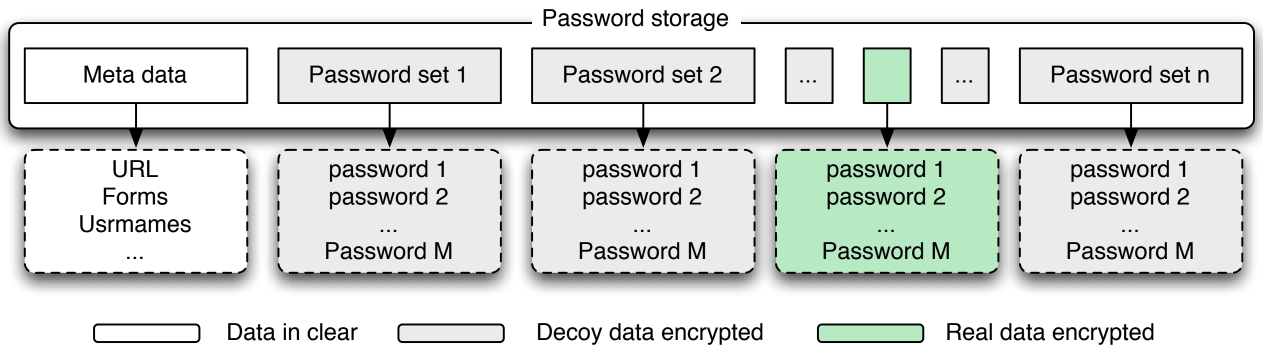

Figure 1: Kamouflage database: cleartext metadata and encrypted real and decoy password sets. Eencryption is discussed in Section 5.1.

password. Such a step is necessary to ensure that the sets look consistent internally, and the real password set is not distinguishable from the rest.

- **Remove.** Remove a password from the database. This is the reverse to adding a password. Assuming that the decoy password sets in the collection were generated well, removing a password should only require removing the web site's entry from each password set. No regeneration is needed.
- **Update.** Update a password entry, presumably when a password is changed. While the size of the password set collection will not change, we can regenerate all decoy sets to ensure that decoys remain indistinguishable from the real password set. This is important, as there are two ways in which an attacker can benefit from a password change operation. First, the new password could no longer convincingly correspond to the old decoys (different length or distribution), hence the need to regenerate the decoys based on the update to maintain the correspondence. Second, the attacker may very well gain access to both versions of the password sets: before and after the update (e.g., owing to the way data is stored on journaling file systems, which rarely allows reliable overwriting). With both versions in hand, the attacker would have an easy time determining which password set was updated, unless the true password and all its decoys are updated at the same time.
- **Find.** Find the right password given some web page characteristics like URL and form field names. Given the master password, this is a straightforward database access in the real password set, ignoring all decoys. (Of course, the MP is needed to locate the correct set.)

4.1 Password set generation

Decoy password sets must be indistinguishable from the real password set. We extend the context-free grammer rules from [17] to work for the case where multiple passwords are being generated for the same user. Decoys are generated based on the abstract model as well as the user's real password set.

Recall that in [17] passwords are generated using a probabilistic context-free grammar $G = (V, \Sigma, S, P)$ that is derived, or learned, from a large training set of passwords. For the purpose of generating decoy password sets we want to derive a set of grammars $\mathcal{G} = \{G_1, G_2, ..., G_N\}$, one for each decoy set that needs to be generated. Each grammar G_i is used to produce decoy set i by applying G_i's rules in a probabilistic manner. We let G^* denote a grammer that generates the real password set chosen by the the user. Note that we are not looking to generate the most probable password set, but merely plausible ones. This is a simpler task compared to [17] which aims to produce the most likely passwords first.

The grammars in $\mathcal{G}$ differ along two dimensions: the structure of passwords generated, and the relationships between passwords in each set. The former follows directly from the use of distinguished variables corresponding to letters (L), digits (D), and special characters (S), in the grammars' production rules; the latter is specific to the way we generate and use the collection $\mathcal{G}$:

1. **Reflective derivability:** It should be possible to derive most password sets in the collection using any of the sets as a starting point, not just the real set. This in particular implies that the real password set W^* should be derivable from most other decoy sets under our algorithm. This property ensures that the attacker can not analyze the collection and focus only on a small number of sets which can produce the rest of the collection.

2. **Derivation of password sets:** Intead of a single password, grammars in $\mathcal{G}$ derive *sets of passwords*. Let's assume that we are generating sets of size M, that is, the user's real set and all decoy sets contain exactly M passwords each. A derivation using G_i starts with the M-tuple of special characters $\left\langle S^{(1)}, S^{(2)}, ..., S^{(M)} \right\rangle$, and ends with M passwords: $\langle W_1, W_2, \ldots, W_M \rangle$. (Here the starting symbols need not be all different; in addition, note that ordering is important because each password corresponds to a specific web site and similarities between passwords follow specific patterns that we are trying to establish.) At every step in the derivation, a production rule $P_i^j \in G_i$ is applied to *all* of the sentential forms in the M-tuple, and the further derivation to terminal characters is also synchronized for that portion of the derived passwords. This is done in order to preserve similarity within groups of passwords.

3. **Production rule grouping:** Password similarities in the real password set result (probabilistically) in similarities in decoy sets. This is most easily achieved by first deriving the grammar G^* for the real set, then modifying that grammar in a way that either preserves similarities, completely removes them, or extends them even further. For example, a mutated (decoy) grammar G_i could be obtained by replacing the occurences of a variable L in a rule in G^* with several different variables, effectively reducing similarities within the set. Alternatively different variables of the same "type" (L, D, or S) and "valency" could be replaced by a single variable, causing a group of passwords in the set to become more similar when generated by the resulting grammar.

A short example will clarify the password generation mechanism outlined above: let's assume the real set is of size two ($M = 2$), and we need to generate three different decoy sets ($N = 3$). If the real password set consists of the passwords "jones34monkey" and "jones34pass", then G^* contains the rules $S^{(1)} \rightarrow AL_6$, $S^{(2)} \rightarrow AL_4$, and $A \rightarrow L_5D_2$ (for brevity we are omitting the rules mapping L, D, and S variables to actual characters). Three grammars are derived from G^* using random mutations, for example:

- G_1 contains the rules $S^{(1)} \rightarrow AL_6$, $S^{(2)} \rightarrow BL_4$, $A \rightarrow L_5D_2$, and $B \rightarrow L_3D_4$ (more diversity),

- G_2 contains the rules $S^{(1)} \to AB$, $S^{(2)} \to AB$, $A \to L_5 D_2$, and $B \to L_4$ (more similarity),
- and G_3: $S^{(1)} \to AL_6$, $S^{(2)} \to AL_4$, and $A \to D_2 L_5$ (permutation).

The final decoy sets could then be:

$$\begin{aligned} S_1 &= \{\textit{“boots92recall”}, \textit{“ill2009root”}\}, \\ S_2 &= \{\textit{“tired05amos”}, \textit{“tired05amos”}\}, \\ S_3 &= \{\textit{“13chesscamera”}, \textit{“13chesstree”}\}. \end{aligned}$$

Optionally, the process of generating decoys can be customized by the user to better mimic the real password distribution. The customization choices should not be stored on disk as they can help the attacker.

4.2 Implementation

In order to prove the feasibility of our architecture, we built a proof-of-concept extension for the Firefox web browser, called Kamouflage. The extension implements the `nsILoginManagerStorage` interface and acts as an alternative storage for login credentials.

The main goal in developing the extension was to show that the overhead of maintaining decoy password sets is acceptable, particularly from a user's point of view. We intentionally used no optimizations in the handling of the password database, because we wanted to get a sense of the worst-case performance implied by our approach. Completing the extension to a point where it can be deployed for real-world use is straight forward.

When it is loaded, Kamouflage registers with Firefox as a login manager storage provider (`nsILoginManagerStorage`). Each of the implemented API methods (`addLogin`, `findLogin`, etc.) calls some internal methods that deal with reading and writing the password database file from and to persistent storage, as outlined earlier (see Figure 1). If the password storage file does not exist, it is assumed that the user's password set is the empty set.

Performance. We measured how individual API calls are impacted by various password set collection sizes. We show that performance in Table 1. In our completely naive implementation every time the password storage file is read completely and, upon modification, written out completely. From a user's point of view, there is no impact when maintaining approximately 10^3 decoy password sets; at 10^4 decoy sets the performance drop becomes clearly noticeable.

In practice, the performance of our prototype could be further improved in a number of ways:

- **Caching.** Our measurements of user-visible latency often include several invocations of the `nsILoginManagerStorage` interface, each of which reads the whole file from scratch. The login manager could cache the database contents, reading the file only once, at launch time.
- **Read size.** Password storage does not need to be read in its entirety. Given a master password (input by the user), only one password set needs to be read from disk. In the context of a Firefox extension this would require writing a native implementation for the read function: the JavaScript file I/O API available does not allow random access inside a file. Alternatively, password storage could be split up into several (e.g. 10 or 100) equal parts, and only the part (file) containing the needed password set would be read.

Table 1: User-visible performance of the Kamouflage Firefox extension for three typical use cases.

Collection size (number of decoy sets)	10^3	10^4	10^4
Password set size (number of user passwords)	100	100	20
Database size on disk	2MB	20MB	4MB
Measured performance (access and update time)	< 1 sec	5 sec	< 1 sec

- **Write size.** Password sets do not all have to be rewritten on every `addLogin` or `updateLogin` operation, if we can guarantee that older versions are overwritten and unavailable to an attacker. Due to the random nature of decoy password sets, adding a password to the real set is not going to affect the plausibility of any of the decoy sets (assuming they are pre-initialized to contain enough passwords in the first place). The cost of updating decoy sets can be amortized across many subsequent invocations via the `nsILoginManagerStorage` interface. Of course, if the attacker can see the evolution of the database over time (e.g., in a versioning or a journaling filesystem), then all sets must be updated simultaneously.

5 Extensions and Limitations

The system described in Section 4 does not encrypt the password database with a master password. The master password is only used to identify the location of the real password set. As we will see, encrypting the password database without exposing the system to an offline dictionary attack takes some thought. In this section we extend the system to address this issue and others.

Limtations. We describe a few limitations of the proposed system and then explain how to address them.

- **Side information is dangerous.** An attacker armed with any kind of side information about the victim can be highly successful, being able to guess the correct password set in the collection by searching for victim-related keywords in the password storage, hoping that those keywords appear as part of a password. Alternatively, if the user elects to use a very weak password at a specific, unimportant web site, the attacker may be able to recover it in an online attack, and use that information to crack the master password offline later on. Another, more subtle challenge is that of dealing with site-specific password constraints: unless our decoy password set generator takes those into account, it might generate a password set that looks plausible, however does not meet the constraints at all web sites in the set, which would immediately reveal to the attacker that the set is a decoy.

- **Web sites have a conflict of interest.** Web sites might not have an incentive to block account access when multiple unsuccessful login attempts are made. For example, the web site might be concerned about DoS attacks—if account access is blocked automatically after several unsuccessful login attempts, this presents an easy way to block a specific account. A problem of this kind is applicable to on-line auctions, where there are benefits in being able to block a competitor out of the system, even for a short time at the end of the auction [13].

- **Users perceive a lack of control.** Users of the password manager might feel they are getting insufficient feedback from the system: there is no indication about whether the correct master password was entered. Since each master password leads to a plausible password set, the user only gets feedback when she tries to login to a web site for which there is a stored password. This in unavoidable if offline dictionary attacks are to be prevented.

We will now explain how the above shortcomings can be overcome by extending our architecture with encryption, tighter web site integration, and visual master password fingerprinting.

5.1 Encryption against Side Information

Password managers often encrypt the password database with a master password, denoted MP. In our settings this is non-trivial, and if done incorrectly, can cause more harm than good. To see why, suppose the password database is encrypted using an MP. Our user study from Section 2 shows that people tend to choose master passwords that are related to the passwords being protected. An attacker who obtains the encrypted password database can find the MP with an offline dictionary attack and then quickly identify the real password set by looking for a set containing passwords related to the MP.

Our approach to encryption. We use the following technique to avoid the preceding problem. Recall that each password set S_i contains a set of related decoy passwords generated as discussed in Section 4.1. We use the same approach to generate a master password MP_i for the set S_i, so that MP_i will likewise be related to the passwords in S_i. The master password for the real set is the user-selected MP. Now, for each set S_i do:

- generate a fresh random value IV_i to be stored in the clear with the set S_i, and
- use two key derivation functions (KDF) to generate two values K_i and L_i from MP_i as follows:

$$K_i \leftarrow \mathrm{KDF}_1(\mathrm{MP}_i,\ \mathrm{IV}_i) \qquad \text{and} \qquad L_i \leftarrow \mathrm{KDF}_2(\mathrm{MP}_i)$$

The key K_i is used to encrypt the set S_i. The index L_i deteremines the position of the set S_i in the password database. In other words, the index of the set S_i in the database is determined by the master password for the set S_i. Collisions (i.e., two sets that have the same index L) are handled as in hash-table data structures. Once the database is generated, all master passwords MP_i are deleted. When the user enters the real master password MP the system can recompute the index L to locate the encrypted set and its IV and then recompute K to decrypt the set.

The best strategy for an attack on this system is to run through all dictionary words (candidate MPs) and for each one to compute the corresponding index L and candidate key K. Whenever the attacker eventually tries to decrypt a password set using the actual MP that was used to encrypt it, he can generally recognize this fact, causing that MP to become exposed along with the corresponding password set.[2] However, in the end, even after decrypting all the sets in the password

[2]It is possible to increase security even further by making it more difficult to recognize the correct decryption of a given set, by ensuring that all decryptions, even incorrect ones, result in plausible well-formed outputs. One way to do this is to "compress" the password set before encryption using a suitably pre-populated entropic encoder, and expand the set back upon decryption using the corresponding decoder. Although this may not provide much help if the compressed plaintext contains a lot of residual redundancy, compression is very cheap, and, employed in this manner, will never hurt security.

Table 2: Comparison of attack difficulty in traditional and the new password management schemes, for different master password strengths (distribution known by the attacker).

	Master Passowrd Strength		
Traditional	**Weak**	**Medium**	**Strong**
Offline (# of decryptions)	10^4	10^7	10^{10}
Online (# of login attempts)	1	1	1
Kamouflage	**Weak**	**Medium**	**Strong**
Offline (# of decryptions)	10^4	10^7	10^{10}
Online (# of login attempts)	10^4	10^4	10^4

database with their respective correct master passwords, the attacker must still do substantial online work to determine the good set.

Table 2 shows how the master password strength affects the offline computation effort and the number of online login attempts that an attacker needs to perform, even if the attacker has perfect knowledge of the real master password distribution.

Note that when the user adds or updates a password, the system cannot add passwords to the decoy sets since it does not have the master passwords for the decoy sets. Instead, when the real set is updated, all the decoy sets and their master passwords are regenerated, and all sets are re-encrypted using new random values IV_i. The previous IVs must be securely purged from the database to prevent an attacker to locate the correct set as the one that was re-encrypted under an unchanged MP.

Our performance numbers from Table 1 show that the running time to perform this whole update operation remains acceptable.

By adding encryption to Kamouflage, we have neutralized the threat of attacks based on side information: depending on the MP strength, the attacker may need to spend considerable offline effort before he is in a position to mount the online attack.

5.2 Website Policy Integration

Restrictive password policies can be detrimental to the security of passwords stored using Kamouflage. Imagine that a web site requires that user passwords consist only of digits, while the decoy set generator randomly uses letters and digits when generating all passwords. In this scenario, an attacker looking at all the password sets in the collection can zero in on the real password set, because most likely it is the only one which contains a numeric password for that specific web site. We surveyed the top 10 web sites listed by Alexa, along with a small list of bank web sites, and tabulated their password requirements. The results are shown in Table 3, and clearly demonstrate that this danger is real.

In order to avoid weakening our password management scheme, it appears to be sufficient to mimick the composition of a user's passwords, and ensure that passwords that contain specific classes of characters (lowercase letters, uppercase letters, digits, special characters) continue to contain those classes of characters in the generated decoy sets. Ideally, web sites should start to advertise their password requirements, so an intelligent password manager can contact them and ensure that most decoy passwords comply with the requirements. Password requirements can be

Table 3: Password strength requirements at top sites ranked by Alexa and a small group of finance-related web sites.

Web Site	**Password Requirement**
Google	at least 8 characters
Yahoo!	at least 6 characters
YouTube	at least 8 characters
Facebook	at least 6 characters
Windows Live	at least 6 characters
MSN	at least 6 characters
MySpace	between 6 and 10 characters, at least 1 digit or punctuation
Fidelity	between 6 and 12 characters, *digits only*
Bank of America	between 8 and 20 characters, ≥ 1 digit and ≥ 1 letter, no `$ < > & ^ ! [ ]`
Wells Fargo	between 8 and 10 characters, ≥ 3 of: uppercase, digit, or special characters

advertised in a special XML file at the root level, similar to the way browser favicons are supplied. Any of these two approaches would effectively eliminate the third side-information limitation of the basic architecture described in Section 4.

An additional feature provided to security-conscious users could be a mechanism to guide the password manager's decoy set generator, and instruct it on the distribution from which passwords should be derived for each web site. This would make sure that web sites with really cryptic requirements will not compromise the overall security of the password database.

5.3 "Honeywords" : Using Decoys as Attacker Traps

We have seen that decoy password sets carry certain risks when deployed without care. At the same time, they provide an opportunity to cooperate with web sites in detecting and blocking targeted attacks on user account, addressing the web site conflict of interest limitation.

Recall that most web sites are averse to blocking a user's account when they see a large number of failed login attempts. This is usually due to fear that a user's account will effectively suffer a denial-of-service attack. The reasoning behind this is sound: it is much more likely an attacker is trying to block a user's account, than trying to guess her password. It is attacks that are exceptions to this rule that have the greatest potential to cause damage however: an unauthorized user of an account could transfer money, attack other related accounts, or steal personal information to be used later for identity fraud.

Supplying web sites with some of their corresponding decoy passwords can provide them with an effective tool for identifying attacks that are based on compromised password files, and encourage them to take steps to block the user account in such scenarios. This presents little risk on the part of the web site, because the likelihood that a casual DoS attacker hits a decoy password, without having access to the user's device, should be very low. In other words, knowing that an attack is not a random DoS but a genuine impersonation attempt will make web sites more willing to take immediate and decisive actions to stop the attack.

5.4 Master Password Fingerprinting

The flip side of using decoy traps as a defense mechanism, is that it becomes vital to provide the user with positive feedback on the correctness of the master password being seized. With the honeyword mechanism, a mistake on the master password is indeed much more likely to result into a locked-out account than a mistake on the account's login password itself.

A simple technique similar to Dynamic Security Skins [3] can solve this problem. When the user selects his master password, he can be presented with an icon selected pseudo-randomly from several thousand possible ones, based on the master password. The user remembers the icon and uses its presence as a cue that he typed the correct master password when he logs in again later on. It is very unlikely that the user will mistype his password and at the same time get the same icon, believing the password he typed was correct. In particular, error-correction codes can be employed to ensure that single-character errors always result in different validation icons.

5.5 Limitations, Revisited

It is instructive to take a step back and compare our extended Kamouflage architecture to a "traditional" password manager design.

The encryption step we added ensures that our password database is at least as hard to crack as a traditional one: an attacker that guesses the master password can test her guess offline, however even upon successful decryption of a password set, there will be no guarantee that the decrypted set is the real one and not a decoy. Successfully uncovering all password sets takes time proportional to the size of the master password space, which is just the same as with a traditional design. In other words, with the decoy sets we have added an element of uncertainty that persists even when the whole space of master passwords has been explored offline.

Kamouflage also provides opportunities for additional security mechanisms to be deployed by web sites. We mentioned the use of honeywords, whereby a subset of the decoy password sets could be provided to web sites by the password manager, enabling them to identify and quickly respond to a targeted attack, without providing new opportunities for DoS.

Finally, the visual fingerprinting technique ensures that users have feedback on whether they entered the correct master password, without leaking any information to an attacker, and thus without weakening the strength of the master password.

We have thus addressed all serious limitations that were present in the basic Kamouflage architecture developed in Section 4, to a point where it is clearly advantageous to use the new approach. Obviously, a successful implementation requires a high-quality decoy password set generator: a problem that has to be solved in follow-up work.

6 Additional Related work

We survey in this section some additional related work on password management.

RSA key camouflage. The idea of camouflaging a cryptographic key in a list of junk keys was previously used by Arcot systems [10] to protect RSA signing keys used for authentication. Arcot hid an RSA private key among ten thousand dummy private keys. The user's password was a 4 digit PIN identifying the correct private key. The public key was also kept secret. An attacker who obtained the list of ten thousand private keys could not detemine which is the correct one.

Camouflaging an RSA private key is much easier than camouflaging a password since the distribution of an RSA private key is uniform in the space of keys. Camouflaging a password requires a good model for how humans generate passwords. We remark furthermore that our application to password managers is more general and therefore far easier to deploy as a plug-in replacement for existing password managers in browsers and mobile devices.

Remote password storage. Several password management systems store passwords on a remote third party server. As examples, we mention Verisign's PIP [16], the Ford-Kaliski system [5], and Boyen's Hidden Credential Retrieval [2], while noting that many other proposals exist. These systems, unlike ours, require additional network infrastructure for password management. Moreover, in some systems, like Verisign's, there is considerable trust in the third party since it holds all user passwords. The Ford-Kaliski system distributes the password among multiple severs to reduce trust in the third party. Some systems store the user's password at a single third party encrypted using the user's master password; user passwords are then vulnerable to dictionary attacks by the third party. An exception is Boyen's HCR scheme [2], which is designed to exploit the limited redundancy of stored passwords to prevent the third-party storage facility from validating the master password offline. Our approach has none of these complexities since there is no trusted party involved. Compared to Boyen's HCR [2], Kamouflage can also deal with (sets of) passwords with large amounts of redundancy.

Intelligent dictionary attacks. Several recent papers propose models for how humans generate passwords [11, 6, 17]. These results apply their models to speeding dictionary attacks. Here we apply these models defensively for hiding passwords in a long list of dummy passwords.

Slow hash functions. Many password management proposals discuss slow hash functions for slowing down dictionary attacks [4]. These methods are based on the assumption that the attacker has limited computing power. They can be used to protect the user's master password against dictionary attacks. Our approach, which is secure in the face of an attacker with signficant computing power, is complementary and can be used in conjuction with slow hashing methods for additional security.

Halting functions. A somewhat recent twist on the notion of slow hash function is to allow the user to pick an arbitrary "slowness factor" (e.g., to compensate for the password's weakness), and then *hide* such factor cryptographically as part of the ciphertext, without requiring the user to remember it either. Such "halting key derivation functions" [1] increase security because they force the attacker to guess not only the password, but also an upper bound on the time needed to validate it. HKDFs are orthogonal to, and can be used in the extended Kamouflage architecture to increase the time needed to break the encryption.

Graphical passwords. Graphical passwords [9, 7] are an alternative to text passwords. While they appear to have less entropy than textual passwords, our methods can, in principle, also be used to protect graphical passwords. One would need a model for generating dummy graphical passwords that look identical to human generated passwords. We did not explore this direction.

7 Conclusions

We presented a novel approach to password management that leverages our knowledge of user password selection behavior, substantially increases the expected online work required for a successful attack, and optionally meshes with web services to reliably detect targeted attacks. Our user survey confirms that the proposed architecture addresses real-world use cases and risks that have been overlooked by traditional password management systems. Using a prototype implementation we demonstrated that our architecture can be used as a drop-in replacement to existing systems.

References

[1] Xavier Boyen. Halting password puzzles: hard-to-break encryption from human-memorable keys. In *16th USENIX Security Symposium—SECURITY 2007*, pages 119–134, Berkeley, CA, USA, 2007. USENIX Association. 1, 14

[2] Xavier Boyen. Hidden credential retrieval from a reusable password. In *ACM Symposium on Information, Computer & Communication Security—ASIACCS 2009*, pages 228–238. ACM Press, 2009. 14

[3] Rachna Dhamija and J. D. Tygar. The battle against phishing: Dynamic security skins. In *SOUPS '05: Proceedings of the 2005 symposium on Usable privacy and security*, pages 77–88, New York, NY, USA, 2005. ACM. 13

[4] David Feldmeier and Philip Karn. UNIX password security – 10 years later. In *Proceedings of Crypto '89*, pages 44–63, 1989. 1, 14

[5] W. Ford and B. Kasliski. Server-assisted generation of a strong secret from a password. In *Proceedings of the 9th IEEE International Workshops on Enabling Technologies*, pages 176–180, 2000. 14

[6] Willian Glodek. Using a specialized grammar to generate probable passwords. Master's thesis, Florida state university, 2008. 14

[7] Joseph Goldberg, Jennifer Hagman, and Vibha Sazawal. Doodling our way to better authentication. In *proceedings CHI 2002*, pages 868–869, 2002. 14

[8] J. Alex Halderman, Brent Waters, and Edward W. Felten. A convenient method for securely managing passwords. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 471–479, New York, NY, USA, 2005. ACM.

[9] I. Jermyn, A. Mayer, F. Monrose, M. Reiter, and A. Rubin. The design and analysis of graphical passwords. In *Proc. 8th USENIX Security Symposium*, pages 135–150, 1999. 14

[10] B. Kausik. Method and apparatus for cryptographically camouflaged cryptographic key, 2001. US patent 6170058. 13

[11] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proc. of ACM CCS 2005*, pages 364–372, 2005. 2, 14

[12] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Proceedings of CRYPTO 2003*, pages 617–630, 2003. 1

[13] Account lockout attack, 2009. http://www.owasp.org/index.php/Account_lockout_attack. 9

[14] Openwall Project. John the ripper password cracker, 2005. http://www.openwall.com/john. 1, 3

[15] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. Mitchell. Stronger password authentication using browser extensions. In *proceedings of Usenix security*, 2005. 1

[16] Verisign. Personal identity portal, 2008. https://pip.verisignlabs.com/. 14

[17] Matt Weir, Sudhir Aggarwal, Bill Glodek, and Breno de Medeiros. Password cracking using probabilistic context-free grammars. In *proceedings of IEEE Security and Privacy*, 2009. 2, 6, 7, 14

[18] R. Yampolskiy. Analyzing user passwords selection behavior for reduction of password space. In *Proceedings of the IEEE international carnahan conference on security technology*, pages 109–115, 2006. 2

[19] Jeff Yan, Alan Blackwell, Ross Anderson, and Alasdair Grant. Password memorability and security: Empirical results. *IEEE Security and Privacy magazine*, 2(5):25–31, 2004. 2, 4