Cracking-Resistant Password Vaults using Natural Language Encoders

(Full version)

Rahul Chatterjee*, Joseph Bonneau†, Ari Juels‡, Thomas Ristenpart*

*University of Wisconsin–Madison, Email: {rchat,rist}@cs.wisc.edu
†Stanford University & The Electronic Frontier Foundation, Email: jbonneau@cs.stanford.edu
‡Cornell Tech, Email: juels@cornell.edu

***Abstract*—Password vaults are increasingly popular applications that store multiple passwords encrypted under a single master password that the user memorizes. A password vault can greatly reduce the burden on a user of remembering passwords, but introduces a single point of failure. An attacker that obtains a user's encrypted vault can mount offline brute-force attacks and, if successful, compromise *all* of the passwords in the vault. In this paper, we investigate the construction of encrypted vaults that resist such offline cracking attacks and force attackers instead to mount online attacks.**

Our contributions are as follows. We present an attack and supporting analysis showing that a previous design for cracking-resistant vaults—the only one of which we are aware—actually degrades security relative to conventional password-based approaches. We then introduce a new type of secure encoding scheme that we call a natural language encoder (NLE). An NLE permits the construction of vaults which, when decrypted with the wrong master password, produce plausible-looking decoy passwords. We show how to build NLEs using existing tools from natural language processing, such as n-gram models and probabilistic context-free grammars, and evaluate their ability to generate plausible decoys. Finally, we present, implement, and evaluate a full, NLE-based cracking-resistant vault system called NoCrack.

I. Introduction

To alleviate the burden of memorization, many security experts recommend the use of *password vaults*, also known as "wallets" or "managers" [7]. Password vaults are applications that store a user's website passwords, along with the associated domains, in a small database that may be encrypted under a single *master password*. Vaults enable users to select stronger passwords for each website (since they need not memorize them) or, better yet, use cryptographically strong, randomly chosen keys as passwords. Many modern vault applications also backup encrypted vaults with an online storage service, ensuring that as long as users remember their master passwords, they can always recover their stored credentials.

These vault management services provide attackers with a rich target for compromise. The popular password vault service LastPass, for example, reported a suspected breach in 2011 [40]. Security analyses of several popular web-based password managers revealed critical vulnerabilities in all of them, including bugs which allow a malicious web site to exfiltrate the encrypted password vault from the browser [31].

An attacker that captures users' encrypted vaults can mount offline brute-force attacks against them, attempting decryption using (what the attacker believes to be) the most probable master password mpw_1, the second most probable mpw_2, and so on. With standard password-based encryption (PBE) algorithms (e.g., PKCS#5 [27]), only the correct master password produces a valid decryption, so an attacker knows when a vault has been successfully cracked. Such attacks are offline in the sense that they can be conducted without interaction with a server and hence are limited only by the attacker's computational capabilities, as measured by the number of decryption attempts performed per second.

Available evidence (c.f., [5]) suggests that most master passwords selected by users are weak in the sense that even a modestly resourced attacker can feasibly crack them in a matter of minutes or hours.

This state of affairs prompts the following core question addressed by our work: *Can password vaults be encrypted under low-entropy passwords, yet resist offline attacks?* We call such vaults *cracking-resistant*. Cracking-resistant vaults would have the benefit that attackers, even after investing significant resources in offline cracking, would still need to perform a large number of online login attempts. Such online login attempts would significantly impede attackers and enable service providers to detect vault compromises evidenced by failed login attempts.

Breaking Kamouflage. At first glance, it may seem that preventing offline cracking is impossible. After all, the attacker has access to the ciphertext and the key is likely low-entropy and so easily guessable. But Bojinov, Bursztein, Boyen, and Boneh, who first investigated the question of cracking-resistance for vaults, suggest a clever idea: include decoy or honey vaults in an attempt to force offline attacks to become online attacks [4]. Their system, called Kamouflage, permits seemingly successful decryption under some *incorrect* master passwords. Kamouflage stores not only an encryption of the true vault (under the true master password) but also a large number $N-1$ of decoy vaults encrypted under decoy master passwords. The security goal is for an offline attack to reveal only a set of equally plausible vaults, forcing an attacker to attempt a login with a password from

each of the decrypted vaults in order to find the true one. The quantitative security claimed in [4] is that an attacker must first perform offline work equivalent to that needed to crack conventional PBE, but additionally perform an expected $N/2$ online queries.

Our first contribution is discovery of a subtle vulnerability in this approach; we show a cracking attack that exploits it, and evaluate the attack's efficacy. To explain briefly, Kamouflage stores $N-1$ decoy vaults encrypted under decoy master passwords that are generated by parsing the true master password and using its structure to help select the decoys. Analysis of password leaks shows that learning the true master password's structure reduces the attacker's search space substantially more than one might expect. Successfully cracking *any* of the N vaults (decoy or true), therefore, reveals this structure and confers a big speed-up on an attacker. We explain the issue in greater detail in the body.

The upshot is that using Kamouflage actually *degrades* overall security relative to traditional PBE: an attacker can perform significantly less computational work (on average about 40% less for $N = 1{,}000$) and make just a handful of online queries. In practice, an attack against Kamouflage would therefore require less total computational resource than one against traditional PBE, in almost all cases.

Unfortunately this flaw is not easy to fix, but rather seems fundamental to the Kamouflage design. Thus our finding reopens the question of how to build cracking-resistant vaults.

Natural-language encoders. Our next contribution is a new approach to cracking-resistant vault design. The vulnerability in Kamouflage is leakage of information by the explicitly stored set of decoy ciphertexts about the true master password. So we seek a design that stores a single explicit ciphertext and has the property that decryption with the wrong key ends up generating on-the-fly decoy vaults. Here we are inspired by Hoover and Kausik's cryptographic camouflage system [22] and a generalization called honey encryption (HE) [25]. These prior schemes, however, work on plaintexts that are drawn from simple distributions, i.e., uniform over some set. They would work (with some adaptations) for computer-generated random passwords, but fail, as we confirm experimentally, to impart cracking-resistance to password managers that may include human-selected passwords.

To meet the challenge of good decoy vault generation, we introduce the concept of *natural language encoders* (NLEs). An NLE is an algorithm that encodes natural language texts (in our case, lists of human-selected passwords) such that decoding a uniformly selected bit string results in a fresh sample of natural language text. Technically, NLEs are a new class of the more general concept of a distribution-transforming encoder [25], previous examples of which generated uniform prime numbers and uniform integers. NLEs require different techniques given the (widely understood) additional complexity of modeling natural languages, even in the restricted domain of passwords [29], [37], [39].

Rather than design NLEs from scratch, we instead show how to convert existing models from statistical natural language processing into NLEs. Models are compact representations of the distribution of natural language texts; particular choices might include n-gram Markov models, probabilistic context-free grammars (PCFGs), etc. We focus on the first two because they have been used to build effective password crackers [29], [32], [37], [39]. Our constructions of NLEs from these model types therefore gives, in effect, a way to retool password crackers to help us build cracking-resistant vaults. This approach has significant benefits, as it permits future improvements to natural language models to be incorporated easily into password vault systems.

We build a number of NLEs based on existing models trained from password leaks as well as one new PCFG-based model. To evaluate security, we must determine whether decoys generated via the decode function are distinguishable from true passwords. More specifically, given a sample that is either a decoy or a random sample from a password leak (which acts as proxy for a real password), the adversary must classify its input as a decoy or not. We experimentally evaluate a number of machine learning classification attacks that allow the adversary to train on decoys as well as passwords from the true distribution. These analyses show that basic machine-learning attacks do not break our schemes. A human might be able to distinguish real from decoy, but forcing attackers to rely on the manual effort of humans (across thousands of decoys in a typical parametrization) would also be a huge improvement in security over existing solutions.

We also provide the first model of the password sets associated with password vaults which may include related or repeated passwords. Unlike single-password models, for which there is a large amount of public data available, there is not yet much public data available to researchers on full password vaults. (This dearth of data is also presumably problematic for would-be attackers.) Again we perform a preliminary analysis using a small number of leaked vaults. This analysis suggests that an extension of our PCFG-based NLEs generates password vaults resistant to simple machine-learning attacks that attempt to distinguish fake vaults from real ones. We caution that future work might find better attacks against our particular NLEs. By construction, however, our approach never yields security worse than conventional PBE, unlike the case with Kamouflage (as shown by our attack). Our NLEs at least provide another obstacle that crackers must find ways to overcome, one that may be bolstered by improvements in language models.

A full vault system. We incorporate NLEs into a full encrypted vault service called NoCrack that resists offline

brute-force attacks better than alternative password vault schemes. NoCrack addresses several previously unexplored challenges that arise in practical deployment of a honey vault system: concealing website names associated with vault passwords, incorporating computer-generated passwords in addition to user-selected ones, and authenticating users to the NoCrack service while ensuring service compromise does not permit offline brute-force attacks. We report on a prototype implementation of NoCrack that will be made public and open-source.

Summary. In summary, our contributions include:

- *Breaking Kamouflage:* We show that Kamouflage provides less security than its original analysis suggested, and in most realistic cases even less than traditional PBE schemes.
- *Natural-language encoders (NLEs) for vaults:* We introduce the concept of natural language encoders, and show how to build them from typical password models. This approach allows us to generate realistic decoys on the fly during brute-force attacks.
- *Cracking-resistant password vaults:* We use our new NLEs as the basis for a password vault system called NoCrack. It addresses a number of issues for the first time, including how to deal with authentication, concealment of the websites for which a user stores vault passwords, and more.

II. Background and Existing Approaches

In practice, password vaults are encrypted under a user's master password using a password based encryption (PBE) scheme. In more detail, a user's set $\vec{P}$ of passwords is encrypted under a master password mpw using authenticated encryption. Encryption and decryption use a password-based key-derivation function (KDF), meaning encryption is under a key computed by applying a hash chain to a random salt and mpw [27]. (The salt is stored with the ciphertext.) A decryption operation under an incorrect master password mpw' will fail, as it will not authenticate correctly[1]. It is for this reason that an adversary can mount an offline brute-force attack against a vault. The adversary's trial decryption will fail until the correct master password mpw is discovered.

Currently in-use KDFs prevent precomputation attacks [21] (such as rainbow tables [34]) by salting. They also increase the cost of brute-force attacks by iterated hashing, which slows down each decryption attempt. See [2] for a formal analysis. But all this only slows down brute-force attacks and does not prevent them; the protection conferred on $\vec{P}$ by PBE is ultimately a function of the resistance of mpw to guessing attacks. Numerous studies, e.g., [6], have shown that users tend in general to select passwords that have low guessing entropy. There is no available evidence indicating that users choose significantly stronger master passwords. Given the growing use of password vaults by consumers and the vulnerabilities they introduce—bulk compromise on servers or compromise on consumer devices—the risk of brute-force attacks is real and pressing.

[1]Some systems may use unauthenticated encryption such as CBC mode. Decryption under the wrong master password is still detectable with overwhelming probability, c.f. [25].

Enter decoys. It is the inherent limitations of conventional PBE in thwarting offline brute-force attack that motivated Bojinov et al. [4] to propose the Kamouflage system. As explained above, the idea behind Kamouflage is to enable multiple master passwords to successfully decrypt a vault, while only the true master password yields the correct vault plaintext. The hope is that the attacker cannot distinguish between real and decoy offline, and must instead make online queries using decrypted credentials to identify the true vault.

Specifically, to protect the true master password and vault $(mpw^*, \vec{P}^*)$, Kamouflage generates $N-1$ decoy $(mpw_i, \vec{P}_i)$ pairs and stores them in a list. Decoys are generated using dictionaries of commonly seen tokens (strings of contiguous letters, numbers or symbols) found in a password leak. We describe decoy generation in more detail in the next section. Every $\vec{P}_i$ is encrypted using a conventional PBE scheme under mpw_i and kept at a location L_j that also calculated from mpw_i. If a collision results during the generation process, i.e., a master password is created that maps to an already occupied location, then a fresh $(mpw_i, \vec{P}_i)$ pair is generated. Thus, given master password mpw', it is possible to locate the corresponding vault and attempt to decrypt it with mpw'.

Kamouflage has some notable deployment limitations. As it stores N vaults, its storage cost is linear in the security parameter N. Additionally, Kamouflage discourages user passwords that are not parsable using its dictionaries, a major obstacle to practical use. Such password rejection may also degrade security by encouraging users to choose weaker passwords than they otherwise might. Finally, Kamouflage provides no guidance on protecting the confidentiality of domains paired with the passwords in a vault and on storing computer-generated passwords, both features of today's commercial systems. More fundamentally, the approach of hiding the true vault among an explicitly stored list of decoy vaults has inherent security limitations, as we explain in the next section.

Threat Model. The primary threat model that we consider is theft of encrypted password vault due to adversarial compromise of a vault storage service, a lost or stolen client device, or a software vulnerability allowing exfiltration of the encrypted vault (c.f., [31]). The attacker does not know anything about the passwords present inside the vault but she has knowledge about the distribution of human generated passwords (learned from publicly available password leaks).

Statistics	RockYou	RY-tr	RY-ts	Myspace	Yahoo
Number of accounts	32.6 M	29.6M	2.98M	41,537	442,846
Number of Unique passwords	14.3 M	13.0M	1.3M	37,136	342,517
Min-entropy (bits)	6.8	6.7	5.3	9.1	8.1
Avg. password length	7.9	7.9	7.9	8.5	8.3
Avg. letters per password	5.7	5.7	5.6	6.4	6.2
Avg. digits per password	2.2	2.1	2.3	1.8	2.0
Avg. symbols per password	0.05	0.05	0.05	0.3	0.04
Letter-only passwords	44.0%	44.1%	43.8%	7.0%	34.6%
Passwords w/ digits	52.2%	54.0%	54.3%	84.8%	64.7%
Passwords w/ symbols	3.7%	3.7%	3.7%	10.7%	2.8%

Figure 1. Statistics of the password leak datasets used in this paper.

She also knows the encryption algorithm and the other information (if any) used by the algorithm at the time of encryption or decryption. She does not know the master password or any randomness consumed by the encryption algorithm.

Her objective is to learn the correct master password (and corresponding in-vault passwords) using a minimal amount of computation and online querying. She tries to decrypt the vault using offline brute force attack against the master password. She tests the correctness of a vault by attempting to log into a domain with a corresponding credential in the vault.

We will measure both the offline and online resources used by the attacker in terms of number of decryption attempts and number of online queries, respectively. To attack a conventionally encrypted vault, no online queries are required and the offline work is a function of the adversary's uncertainty about the master password; for Kamouflage, the claimed security is offline work equivalent to that of conventional encryption and an expected $N/2$ online queries. In each of our experiments, we will calculate expected offline and online work over a choice of master password drawn from a distribution that we will explicitly specify. The distributions we use will be informed by password leaks, as we now discuss.

Datasets. In the course of this paper we use a number of datasets to train password language models, to train attackers, and to test attackers. We primarily use three large-scale password leaks: RockYou, Myspace, and Yahoo. RockYou is the largest leaked clear-text password set to date and is used extensively in modeling distributions of human-chosen passwords [23], [37], [39]. The Myspace leak occurred when passwords for user accounts were publicly posted; the passwords were gleaned from a phishing attack against Myspace's home page. In 2012, Yahoo lost nearly 450,000 passwords after a server breach. Note that all of these leaks contain only original plaintext passwords and not cracked password hashes, as in some other leaks.

As we wish to ensure different training sets for the password model and attacker in some cases, we partition the RockYou passwords into two sets, randomly assigning 90% of the passwords to a set denoted RY-tr and the remaining 10% to RY-(ts). We train our language model with RY-tr only, but use all sets for adversarial training and testing (with cross-fold validation as appropriate). Figure 1 presents statistics on our data sets. Given its large size, we did not make use of multiple splits of the RockYou data set in our experiments. We use RY-ts as a testing set to model settings in which the adversary and defender have equivalently accurate knowledge of the distribution from which user passwords originate. The Myspace and Yahoo data sets serve to model more challenging scenarios in which the adversary has more accurate knowledge of the password distribution than the defender.

III. Cracking Kamouflage

The stated security goal for Kamouflage is that given an encrypted vault, an attacker must first perform an offline brute-force attack to recover all of the plaintext vaults and then perform an expected $N/2$ online login attempts to identify the true vault. Through simulations using the leaked password datasets discussed above, we now show that Kamouflage falls short of its intended security goal. We first present more details on Kamouflage, and then describe and analyze an attack against it.

Decoy generation in Kamouflage. To construct decoy plaintext vaults, Kamouflage uses an approach based on deriving templates from the true plaintext vault. Let $\mathtt{L}_n$, $\mathtt{D}_n$, and $\mathtt{S}_n$ each be a subset of all n-digit strings containing only upper or lower case English letters, only decimal digits, and only punctuation marks and other common ASCII symbols, respectively. We refer to the sets $\mathtt{L}_n$, $\mathtt{D}_n$, and $\mathtt{S}_n$ as *token dictionaries* and individual strings in token dictionaries as *tokens*. No token appears in multiple token dictionaries. Token dictionaries are initially populated with tokens by parsing the passwords in a password leak[2] and placing each resulting token in the appropriate dictionary. Here parsing of each password in the leak is performed by greedily picking the longest contiguous prefix of just letters, digits, or

[2]Also suggested in [4] is to use the dictionaries used by password cracking tools such as "John the Ripper". In our experience leaks work better for Kamouflage.

symbols of a password, moving this prefix to the appropriate token dictionary, and then repeating the process on the remainder of the original password.

Once token dictionaries are fixed, Kamouflage parses a user-input password as a sequence of tokens found in the dictionaries. Successful parsing yields a sequence of tokens whose concatenation equals the original password. As above, parsing involves greedy decomposition of the password into tokens of contiguous letter, digit, or symbol strings; the presence of each token in the appropriate dictionary is then checked. (If any token is not present in its appropriate dictionary, then Kamouflage rejects the input password and prompts the user to pick another.)

A *password template* is the structural description of a password, expressed as the sequence of token dictionaries corresponding to tokens yielded by parsing of the password. For example, the password `password@123` is parsed as `password`, `@`, `123`, and the associated template is therefore $\mathtt{L}_8\mathtt{S}_1\mathtt{D}_3$.

A *vault template* extends the notion of a password template to sequences of passwords, while also keeping track of reuse of the same token in multiple locations. For a sequence of passwords $mpw^*, \vec{P}^*$, the vault template is generated as follows. First, parse each password. Then, for each unique token across all of the parsings, replace it with a symbol $\mathtt{X}_n^i$ for $\mathtt{X} \in \{\mathtt{L}, \mathtt{D}, \mathtt{S}\}$ where n denotes the length of the token and i denotes that this is the i^{th} token in $\mathtt{X}_n$ found in the sequence of parsings. The resulting sequence of password templates constitutes the vault template. In the example below, the first column contains the passwords composing a small vault: a master password followed by two in-vault passwords (for logging into websites). The second column contains the sequence of corresponding password templates that make up the full vault template.

`password@456123`		$\mathtt{L}_8^1\mathtt{S}_1^1\mathtt{D}_6^1$
`password4site`	$\rightarrow$	$\mathtt{L}_8^1\mathtt{D}_1^1\mathtt{L}_4^1$
`bob!Site`		$\mathtt{L}_3^1\mathtt{S}_1^2\mathtt{L}_4^2$

Observe that the symbol L_8^1 is used twice, as the substring `password` appears twice, but the distinct substrings `site` and `Site` are respectively replaced by distinct symbols $\mathtt{L}_4^1$ and $\mathtt{L}_4^2$.

Given the vault template for $(mpw^*, \vec{P}^*)$, Kamouflage produces each decoy vault $(mpw_i, \vec{P}_i)$ by replacing each unique symbol X_n^i with a token chosen uniformly at random from the token dictionary X_n.

The attack. We exploit two vulnerabilities in Kamouflage. First, all of the decoy master passwords have the same template as the true master password. As soon as an adversary recovers any (decoy or real) master password during an offline brute-force attack, the corresponding template is revealed. Knowledge of this template enables the adversary to narrow its search significantly, to master passwords that match the revealed template. This strategy permits an offline attack to be accelerated to the point where it is *faster than cracking a conventional PBE ciphertext*.

Second, decoy master passwords are chosen *uniformly* with respect to the master-password template, i.e., tokens are selected uniformly at random from their respective symbol dictionaries. The decoy master passwords that result are distributed differently than real, user-selected passwords. Thus, if an adversary guesses master passwords in order of popularity, the real master password is more likely to be assigned a high rank (and so guessed sooner) than the decoy master passwords.

Given these two vulnerabilities, we craft an attack that employs a simple model of password likelihood in which the probability of a password is the product of the probability of its template and the probabilities of replacements for each template symbol. For example, $\Pr[\mathtt{password9}] = \Pr[\mathtt{L}_8\mathtt{D}_1] \cdot \Pr[\mathtt{password}|\mathtt{L}_8] \cdot \Pr[\mathtt{9}|\mathtt{D}_1]$. The model is trained using a password leak. Specifically, $\Pr[\mathtt{L}_8\mathtt{D}_1]$ is defined to be the empirical probability that a password in the dataset has template $\mathtt{L}_8\mathtt{D}_1$, and similarly for other passwords. (Other models may be used, of course.)

Given this model and a challenge consisting of N Kamouflage-encrypted vaults, the attack proceeds using two guessing strategies, one offline and one online. First, in an offline effort, the attacker generates trial master passwords in decreasing order of their probability within the password-likelihood model, until one decrypts one of the N vaults decrypts successfully. At this point, the adversary has learned the template of the true master password and may narrow its offline search to master passwords that match this template, still in order of their probability within the password-likelihood model.

Upon decrypting any vault successfully, i.e., discovering its corresponding (true or decoy) master password, the attacker makes an online login attempt against a website[3] with one of the retrieved vault passwords. If the login succeeds, the adversary has identified the true vault and halts. Otherwise, the adversary resumes the offline attack against master passwords.

Attack evaluation. To evaluate the speedup of our attack over a naïve Kamouflage-cracking strategy, we perform simulations in a simplified attack model. The adversary is given a Kamouflage-encrypted vault and access to an oracle that indicates whether a queried password is the true master password or not. (This oracle corresponds in a real attack to an adversary's ability to test a password from a decrypted vault via an online query to a real website.) We count oracle queries as well as offline decryption attempts. (We treat the KDF-induced slowdown as a unit measure.) Thus the challenger takes a master password (mpw^*) and a number

[3]Recall that Kamouflage does not encrypt the sites associated with each entry in the vault.

N, and generates an encrypted Kamouflage vault set of size N. Then the attacker is given this vault set and access to the oracle. Its goal is to guess mpw^*.

The Kamouflage decoy generation algorithm uses a dictionary of replacements for $\mathtt{X_n}$. Given that this is public (being used by any implementation of Kamouflage), we assume the adversary has access to it as well.

Evaluation. We use the RY-tr dataset for training both Kamouflage's parser (i.e., populating the token dictionaries) and the attacker mentioned above. We use master passwords sampled from the RY-ts, Myspace, and Yahoo leaks for testing the performance of the attack. Our evaluation therefore covers both the case when master passwords are chosen from a distribution (RY-ts) similar to that used to train Kamouflage as well as the case when they are chosen from a difference distribution (Myspace and Yahoo). For every password in each of the three sets, we use the password as the true master password mpw. We then construct $N-1$ decoy master passwords[4] for both $N = 10^3$ and $N = 10^4$ (the values suggested in [4]). We then calculate the median offline and online work of 100 iterations for each (mpw, N) combination using fresh coins for decoy generation in each iteration.

We used only the first 50 million trial master passwords generated by the attacker's password-likelihood model for the attack, meaning the attack will not succeed against every master password in the datasets. We also exclude any passwords that can't be parsed by Kamouflage (only 0.01%, 11.43%, and 13.49% of passwords for RY-ts, Myspace, and Yahoo).

This allows us to compute a number of statistics regarding the attack's efficacy. We start with the average difficulty of cracking a given password sampled from the three challenge distributions, assuming it is crackable using the first 50 million guesses by our password cracker. Figure 2 gives a breakdown of this statistic across the various settings. (The results for Kamouflage+ are explained below.)

The takeaway is that for $N = 10^3$, breaking a Kamouflage vault requires on average only 44% of the computational cost of breaking PBE and incurs only 11 online queries if master passwords are sampled from Myspace leak. If the value of N is increased to 10^4, an attacker needs less than 24% of the computational work required in the case of PBE to break a Kamouflage vault. The offline work therefore goes down with increasing N, which stems from the fact that more decoys means more chances of learning the master-password template early in the attack. The online work does increase with increasing N, for $N = 10^4$ requiring an expected 108 queries for Myspace or 219 for Yahoo challenges. Nevertheless, all the numbers for Kamouflage are an order of magnitude smaller than the goal that an attacker must make expected number of queries $N/2 = 5,000$ (for $N = 10^4$). For RY-ts the offline speedup is less but the attacker requires only 2 online queries on an average for $N = 10^3$ and 18 for $N = 10^4$ to break RY-ts challenge. The PBE numbers are also quite low for RY-ts: the attacker does better when trained on a master password distribution close to that used by the target user.

A perhaps more refined measure of attack success is the α-guesswork factor introduced by Bonneau [6]. This factor measures the work required to break any αt-sized subset of a corpus of t total passwords. So at $\alpha = 0.5$, we give the α-guesswork in terms of the number of offline computations to break any half of the passwords in each leak and the number of online queries to break any half of the passwords. Figure 3 shows the results for the range of α we considered. The graphs end at $\alpha = 0.82$ for Yahoo and $\alpha = 0.76$ for Myspace, reflecting the 18% and 24% of challenge passwords we either did not crack in the first 50 million guesses or which were not parsable by Kamouflage. For RY-ts the first 50 million were able to crack all but a handful of the passwords. While we display both online and offline queries on the same charts, they do not necessarily correspond to the same subsets of master passwords.

The charts give a more expansive view of Kamouflage's resistance to attacks as compared to conventional PBE. Looking at the top left chart in Figure 3, we see that for conventional PBE (the highest solid black line), up to $\alpha = 0.5$ of Myspace passwords can be cracked in 2^{21} offline decryptions, while for Kamouflage the same can be achieved with only 2^{19} offline queries (second highest red dashed line). Half of the master passwords can be broken in two online queries for $N = 10^3$ (lowest densely dashed red line). Increasing Kamouflage's security parameter to $N = 10^4$ (bottom left chart) slightly increases the online work for some α (bottom blue curve), but at the cost of yielding even more efficient offline decryption costs. Breaking Kamouflage for a set of $\alpha = 0.5$ of Myspace passwords is achievable with 2^{15} offline computations. The Yahoo charts (middle two) show a similar picture. Half the master passwords can be cracked in just two online queries for $N = 10^3$. The attack performs best in the case of RY-ts. Here the attacker can break more than 95% of master passwords with only one online query (top right chart). This is because the attacker has a very good estimate of the distribution of the user's true password.

The largest number of online queries needed at $\alpha = 0.5$ across all six settings is just 11 for Myspace with $N = 10^4$.

Kamouflage+. Our attack against Kamouflage required few online queries, which is due to the fact that decoys were mostly ranked lower by our cracker than the true master password. This vulnerability arises because Kamouflage's uniform selection of tokens produces relatively unlikely decoy master passwords. We therefore experimented with

[4]Since our attack does not exploit the vault contents, we dispense with simulating generation of site passwords.

Approach	N	Myspace		Yahoo		RY-ts	
		Offline ($\times 10^3$)	Online	Offline ($\times 10^3$)	Online	Offline ($\times 10^3$)	Online
PBE	–	5,740	0	2,467	0	114	0
Kamouflage	10^3	2,550	11	1,261	22	84	2
	10^4	1,381	108	751	219	64	18
Kamouflage+	10^3	165	344	76	302	9	190
	10^4	150	3,118	70	2,168	7	777

Figure 2. The expected amount of offline work (rounded to nearest thousand decryption attempts for integer) and online work (rounded to nearest number of login attempts to a website) required to break a conventional vault (PBE), Kamouflage, and Kamouflage+ for $N \in \{10^3, 10^4\}$. Here expectations are taken over the distribution of Myspace, Yahoo, or RY-ts passwords, normalized after removing those that cannot be parsed using the Kamouflage grammar and that cannot be cracked using the first 50 million guesses of our cracker.

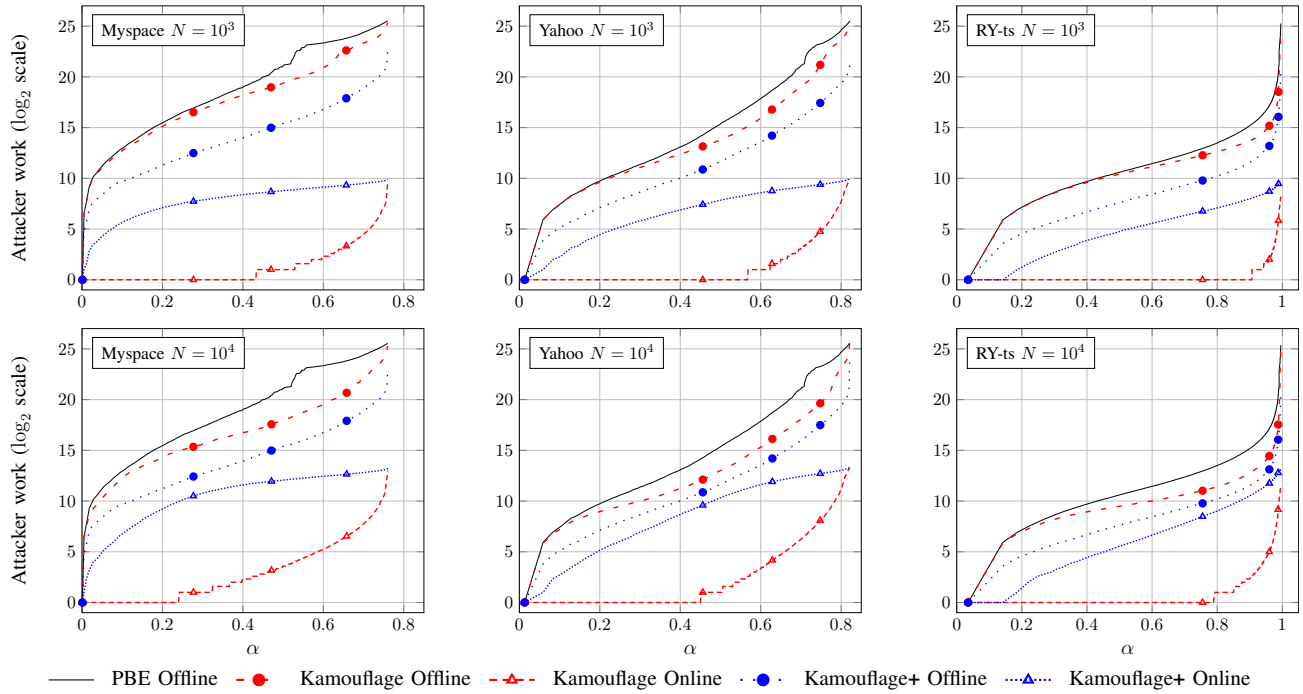

Figure 3. Experimental results for attacking conventional PBE (the baseline) and for Kamouflage and Kamouflage+ vaults for $N \in \{10^3, 10^4\}$. Shown is the α-guesswork measured in number of offline attacker decryptions and number of online attacker queries (both $\log_2$ scale).

a modified version of Kamouflage, called Kamouflage+, that instead chooses tokens according to their frequency in the RY-tr dataset. We repeated simulations of our attack, exactly as above but now with decoy master passwords generated by Kamouflage+. The expected work for our various experimental configurations is given in Figure 2. The results in terms of α-guesswork are shown in Figure 3 (the dotted blue line, third from top being offline work and the dotted blue line second from bottom being online work).

This modification to Kamouflage actually backfires in the case of offline work. This is because decoys are now more likely to be probable passwords, speeding up even further the attacker's discovery of the master password template. On the other hand, Kamouflage+ does increase the online work for our attack, but this is still noticeably below the intended goal of $N/2$ in all cases. Even with $N = 10^4$, the expected number of online queries is never more than 65% of the goal of 5,000 queries.

Summary. This relatively simple attack shows how existing approaches to cracking-resistant vaults actually degrade security relative to conventional PBE. Specifically, the general Kamouflage approach of storing an explicit list of ciphertexts, with one true vault embedded among many decoys generated as a function of the true master password, provides brute-force attackers with an offline speedup that increases with the number of decoys. Attackers may even be able to do better in the online portion should they analyze the vault contents more carefully (our attack above orders online queries solely by master password likelihood).

The lesson here is a warning against building decoys as a function of the true master password. One might consider modifying Kamouflage to use decoy master passwords independently sampled from a model of the password distribution, in order to eliminate the speedups obtained by our attack. But this approach gives rise to other limitations, such as a cap of $N/2$ expected online queries irrespective of the entropy of the true master password. We therefore go a different route.

IV. Overview of Our Approach

The approach of hiding a true vault in a list of encrypted vaults appears to have fundamental limitations. Most obviously it only allows a number of decoys linear in the storage size, and the construction of effective decoy master passwords, as shown for Kamouflage in the last section, is challenging.

We take a significantly different approach. Instead of explicitly storing decoy vaults, we construct single ciphertext which, when decrypted with *any* wrong master password, yields a decoy vault that appears to have been sampled from the distribution of plaintext vaults (across the entire user population). This approach is inspired by the theory of honey encryption [25]. As we will see, though, successfully building a honey vault using honey encryption raises a number of challenges, chief among them accurately modeling the distribution of human-generated password sets in vaults.

Honey encryption. Juels and Ristenpart [25] introduced *honey encryption* (HE) as a mechanism for encryption of secrets under low-entropy passwords. We describe their general approach, with terminology tailored to our setting. Encryption with HE takes as input a master password mpw and plaintext M and outputs a ciphertext, which we denote by $C = \mathsf{HEnc}(mpw, M)$. Encryption is usually randomized. Decryption takes as input a master password and ciphertext C, and outputs a plaintext, denoted $M = \mathsf{HDec}(mpw, C)$. In our setting, a plaintext is a single password P or vector of passwords $\vec{P}$. We require that $\mathsf{HDec}(mpw, \mathsf{HEnc}(mpw, M)) = M$ with probability one for all mpw, M (over any randomness used in encryption).

HE schemes are designed so that a ciphertext, when decrypted under an incorrect master password $mpw' \neq mpw$, emits a "plausible" plaintext M'. This requires a good model, built into the HE scheme specification, of the distribution from which plaintexts are drawn.

As a concrete example Juels and Ristenpart gave an HE scheme for a message M that is a uniformly sampled prime number. (This scheme can be leveraged to encrypted RSA private keys.) Their construction follows a general approach to building HE schemes that composes a distribution-transforming encoder (DTE) with a carefully chosen, but still conventional, PBE scheme. The latter can be, for example, CTR-mode encryption with key derived from the master password using a PBKDF. A DTE scheme specifies a randomized encoding of a message as a string of bits. Decoding does the reverse (deterministically). For a secure HE scheme, it is a requirement that the output of the encoding, for M drawn from some target distribution, looks uniformly distributed and that decoding a uniform bit string give rises to a distribution for M similar to the target distribution. For prime numbers, Juels and Ristenpart give a secure DTE that essentially converts a sampling algorithm for uniformly distributed prime numbers into a DTE encoding and decoding algorithm pair.

Given DTEs suitable for password vaults, an HE-based approach has several benefits over the hide-in-an-explicit-list approach of Kamouflage. First, regardless of the quality of the DTE, because there is only one ciphertext, the amount of offline cracking effort required by an attacker is never less than for a conventional PBE-only ciphertext. This means that, unlike Kamouflage, an HE-based scheme will never provide attackers with a speed-up in offline work. Additionally, the size of ciphertexts in HE does not depend on the number of decoys possible, rather decoys are generated "on-the-fly" during a brute-force attack for each guessed master password. Therefore, given a good DTE, the online work required by an adversary is essentially the strength (guessing entropy) of mpw. A strong mpw will mean the attacker must make many online queries.

We would like a cracking-resistant vault to support use of both computer-generated and human-chosen passwords. The former is relatively straightforward, as computer-generated passwords come from an easy-to-characterize distribution. Human-chosen passwords present a significant challenge, however, as they raise the question of whether one can build DTEs that accurately model natural language-type distributions. We show in the next section that such modeling is possible, and handle further challenges of building a full-fledged, encrypted vault management service in the sections that follow.

V. Natural Language Encoders for Passwords

Formally, a DTE is a pair of algorithms $\mathsf{DTE} = (\mathsf{encode}, \mathsf{decode})$, where encode is randomized and decode is deterministic. In our context, encode takes as input a vector of passwords $\vec{P}$ and outputs a bit string of some length s. The deterministic decoder decode takes as input an s-bit string and outputs a password vector. We require that the DTE be *correct*, meaning that $\mathsf{decode}(\mathsf{encode}(\vec{P})) = \vec{P}$ with probability 1 over the coins used by encode. Our DTEs will be designed so that the length s of outputs of encode depends only on ℓ, the number of passwords in $\vec{P}$.

As an example, it is simple to construct a DTE for uniformly random, fixed-length strings of symbols drawn from an alphabet Σ that consists of the 96-character ASCII printable characters. Encoding works in a symbol-by-symbol manner on an input string $\sigma_1 \,\|\, \sigma_2 \,\|\, \ldots \,\|\, \sigma_k$, where $\sigma_i \in$

Σ. Let $\overline{\sigma_i}$ denote the position of σ_i in Σ under some canonical ordering of Σ . Then for each symbol σ_i in turn, encode outputs a large (e.g., 128-bit) integer X_i selected randomly subject to the constraint $X_i \bmod 96 = \overline{\sigma_i}$. (See Appendix B for details on security bounds and other considerations.) Decoding operates in the natural way: Given input $X_1 \,\|\, X_2 \,\|\, \ldots \,\|\, X_k$, it yields output $\sigma_1 \,\|\, \sigma_2 \,\|\, \ldots \,\|\, \sigma_k$ such that $\overline{\sigma_i} = X_i \bmod 96$. Straightforward extensions that we omit for brevity allow construction of a DTE over passwords that conform to standard password-composition policies (such as needing at least one integer, one special symbol, etc.). We refer to this DTE as UNIF.

We will use such a simple *uniform DTE* for computer-generated passwords later. But it provides poor security for human-selected passwords, which are clearly not distributed uniformly. This observation brings us to one of our core tasks: building DTEs that securely encode samples from distributions of natural language-type text. Because we feel that such DTEs will be of broad use, we give them a special name: *natural language encoders*, or NLEs. We focus on DTEs for messages consisting of a single password and, later, lists of passwords. We note, however, that our constructions are quite general and may be applicable in other natural language contexts.

NLEs from password samplers. A starting point for our NLE constructions is password crackers. Early crackers, such as John the Ripper, simply have stored dictionary lists of popular passwords, and can produce samples in order of likelihood. More modern crackers instead learn compact representations of password distributions from password leaks [37], [39], and permit efficient sampling of passwords over the distribution model.

In a bit more detail, we can view a sampling model for passwords as a deterministic algorithm Samp that takes as input a uniformly random bit string U of sufficient length (often called the "coins") and produces a password P. We can characterize Samp in terms of a distribution p, meaning that a password P is output by Samp with probability $p(P)$ (over the selection of bit string U). The goal of a password cracker is to learn an algorithm Samp from one or more password leaks whose corresponding distribution p closely approximates that of human-generated passwords seen in practice.

We might hope, a priori, that one can build a secure DTE from any sampling algorithm Samp. Unfortunately this seems unlikely to work, in the sense that there exists (admittedly artificial) Samp for which building a DTE appears intractable. Briefly, let $\mathsf{Samp}(U) = H(U)$ for some cryptographic hash function H, where U is a random bit string. Then the natural approach for DTE construction is to set $\mathsf{decode}(U) = H(U)$. But for such decode, correctness would mandate that $\mathsf{encode}(P)$ somehow can sample from the set $H^{-1}(U)$ of preimages of U, which would contradict the hash function's security. Of course there may be other ways to build encode, decode that use Samp only as a black-box, and yet achieve correctness and security. We conjecture that a full counter-example exists, but do not have proof.

Such artificial counter-examples aside, for various classes of Samp we can in fact use the straightforward approach of having $\mathsf{decode}(U) = \mathsf{Samp}(U)$. The only requirement is that we can build $\mathsf{encode}(P)$ that samples uniformly from $\mathsf{Samp}^{-1}(P)$. We show how to do so below for a couple of useful classes of samplers: n-gram models and probabilistic context-free grammar (PCFG) models. We start with the single password case for the different models, and then discuss extensions to the (trickier) case of a vault of possibly related passwords.

NLEs from n-gram models. So-called n-gram models are used widely in natural language domains [13], [15], [32]. For our purposes, an n-gram is a sequence of characters contained in a longer string. For example, the 4-grams of the word '`password12`' are {`pass`, `assw`, `sswo`, `swor`, `word`, `ord1`, `rd12`}. In building models, it is convenient to add two special characters to every string: `^` to the beginning and `$` to the end. Given this enhancement, the 4-gram set in the example above would also include `^pas` and `d12$`.

An n-gram model is a Markov chain of order $n-1$. The probability of a string is estimated by

$$\Pr[\, w_1 w_2 \cdots w_k \,] \approx \prod_{i=1}^{k} \Pr\left[\, w_i | w_{i-(n-1)} \cdots w_{i-1} \,\right]$$

where the individual probabilities in the product are calculated for a given model empirically via some text corpus. For example, for the RockYou password leak, we let $c(\cdot)$ denote the number of occurrences of a substring in the leak. The empirical probability is then

$$\Pr[\, w_i | w_1 w_2 \cdots w_{i-1} \,] = \frac{c(w_1 w_2 \cdots w_i)}{\sum_x c(w_1 \cdots w_{i-1} x)}$$

for any string $w_1 \cdots w_i$ of any length i. Let $F_{w_{i-(n-1)} \cdots w_{i-1}}$ denote the CDF associated the probability distribution for each history. Then the Markov chain associated to such an n-gram model is a directed graph with nodes labeled by n-grams. An edge from node $w_{i-(n-1)} \cdots w_{i-1}$ to $w_{i-(n-2)} \cdots w_i$ is labeled with w_i and $F_{w_{i-(n-1)} \cdots w_{i-1}}(w_i)$. To sample from the model one starts at node `^`, samples from $[0, 1)$, finds the first edge[5] whose CDF value is larger than the sample, follows it to move to the next node, and repeats. The process finishes at a node having the stop symbol. The sequence of w_i values seen on the edges is the resulting string.

Note that such a Markov chain may not have edges and

[5]We again assume an ordering on edges for which CDF values are strictly increasing.

nodes sufficient to cover all possible strings. For use in encoding, then, we extend the Markov chain to ensure that each node has an edge labeled with each character. We set the probabilities for these edges to be negligibly small, and re-normalize the other edge weights appropriately. If this implies a new node we add it as well, and have its output edges all have equal probability.

We can build a DTE by encoding strings as their path in the Markov chain. To encode a string $p = w_1 \cdots w_k$, process each w_i in turn by choosing randomly from the values in $[0, 1)$ that would end up picking the edge labeled with w_{i+1}. Decoding simply uses the input as the random choices in a walk. Both encoding and decoding are fast, namely $\mathcal{O}(n)$ for a password of length n.

In our experiments reported on later, we use a 4-gram model trained from RY-tr. We denote the resulting DTE by NG. We also explored 5-gram models, but in our experiments these used up more space without a significant improvement in security.

NLEs from PCFG models. A PCFG is a five-tuple $G = (N, \Sigma, R, S, p)$ where N is a set of *non-terminals*, Σ is a set of *terminals*, R is set of *relations* $N \rightarrow \{N \cup \Sigma\}$, $S \in N$ is the *start symbol*, and p is a function of the form $R \rightarrow [0, 1]$ denoting the probability of applying a given rule. We require that for any non-terminal $X \in N$, it holds that $\sum_{\beta \in N \cup \Sigma} p(X \rightarrow \beta) = 1$. PCFGs are a compact way of representing a distribution of strings in a language. Each derivation for a member of the language defined by the underlying CFG has a probability associated to it.

Weir, Aggarwal, de Medeiros, and Glodek [39] were the first to apply PCFGs to the task of modeling password distributions. They constructed a password cracker that could enumerate passwords (in approximate order of descending probability) in a way that ensured faster cracking compared to previous approaches like John the Ripper. Weir et al. parsed passwords into sets of contiguous sequences of letters, digits or symbols. Further improvements are possible by employing their approach with alternative parsing schemes. Jakobsson and Dhiman [23] and later Veras et al. [37] used a (so-called) maximum coverage approach for parsing passwords with the help of external language specific dictionaries. Veras et al. also used the semantic meaning of passwords to provide finer granularity parsing, and improved PCFG cracking performance.

We now show how to build a DTE for a single password from any PCFG model. Intuitively, the encoding of a password will be a sequence of probabilities defining a parse tree that is uniformly selected from all such giving rise to the same password. Decoding will just emit the string indicated by the encoded parse tree. We first fix some definitions. A rule $l \rightarrow r$ can be specified as a pair (l, r), where l is a non-terminal and r is a terminal or non-terminal. Every edge in a parse tree is a *rule*. A *rule set* is a lexicographically ordered set of rules with the same left-hand-side. A *rule list* is an ordered list of rules generated by depth-first search of the parse-tree of a string / password (with siblings taken in left to right order).

A CFG is completely specified as a set of rule sets. A PCFG is completely specified by what we call here an *admissible* assignment of probabilities to CFG rules. Let $\mathcal{S}$ be a rule set of size $|\mathcal{S}|$ and $p_{\mathcal{S}}(l \rightarrow r)$ be the probability assigned to a rule $l \rightarrow r$ in $\mathcal{S}$. An admissible assignment of probabilities has the property that $\sum_{(l \rightarrow r) \in \mathcal{S}} p_{\mathcal{S}}(l \rightarrow r) = 1$. We refer to the probability distribution over rules in a rule set $\mathcal{S}$ for a given admissible assignment as its *induced* probability distribution.

As a technical modification to such a PCFG, we add a special *catch-all rule*. Its left-hand side is the start symbol and its right hand side represents any string. We assign this catch-all rule a very low probability (and normalize other probabilities accordingly). This rule ensures all passwords can be parsed (and generated) by the PCFG model and that the model will never fail to encode any real password.

For a given PCFG, a parse tree, and thus a string str, may be specified as a sequence of probabilities $p_1, \ldots, p_k$ (for sufficiently large k). To construct this parse tree, a rule is selected using p_1 from the rule set for the start symbol S, producing the children of S in the tree. A rule from the rule set for each child is then selected using p_2, p_3, etc., from left to right. Recursing in this way produces the full parse tree; its leaves, read left to right, constitute str.

As shown in Appendix B we can represent a rule-set probability by an b-bit integer. It follows that a parse tree for a PCFG, and thus a generated string P, may be completely specified by a vector $\vec{X} = \langle X_1, \ldots, X_k \rangle$ of k integers, where $X_i \in \{0, 1\}^b$. This vector is not necessarily unique: There may, of course, be multiple vectors corresponding to str.

We can now build a DTE from any PCFG model. Decoding takes as input a vector $\vec{X}$ of integers, uses it to determine a parse tree, and outputs the corresponding password P. This requires time $\mathcal{O}(n \log s)$ for n-character passwords and where s is the size of the largest rule set in the PCFG. Encoding takes as input a password P and selects uniformly at random a vector $\vec{X}$ from the set of all that decode to P. This inverse sampling can be efficiently implemented by finding all parse trees (also known as parse forest) of P, and picking one at random. This is an $\mathcal{O}(k^3)$ time operation [3], [17]. Note that $\vec{X}$ is of a fixed size k; thus encoding pads out the resulting vector with random bit strings representing sufficiently many extra integers.

Of course, all of the above relies on having a PCFG that accurately models the password distribution, a research topic in its own right [29], [32], [37], [39]. Our general approach has the benefit of allowing us to use any of these prior PCFG construction approaches. We built our own hand-tuned PCFG using the RockYou training set, employing a

combination of techniques from Weir et al. [39] and Veras et al. [37]. In initial evaluations it performs better than the Weir et al. PCFG (in terms of security; see Section VI). Some further details on the process for generating it are provided in Appendix A. We refer to the DTE built from our new PCFG as PCFG. As baselines for decoy generation quality, we built two additional DTEs, WEIR and WEIR-UNIF. WEIR uses the grammar proposed by Weir et al. [39]. WEIR-UNIF is the same grammar except it ignores frequency information and treats all rules inside a rule-set with equal probability. These grammars are functionally equivalent to those used by Kamouflage+ and Kamouflage, respectively, when restricted to a single password.

From one-password DTEs to vault DTEs. We can easily extend any single-password DTE to handle multiple passwords by applying an encoder independently to each password in the vault. This models a vault distribution in which passwords are independent of one another. This is especially useful when we have both computer-generated passwords in a vault as well as human-chosen; we can choose appropriate single-password DTEs for each case. We denote this independent-password DTE by MPW (for multiple passwords).

Such a DTE may not work well when users repeat or have related passwords in their vaults, however, motivating a decode algorithm that generates a vector of passwords $\vec{P}$ in which passwords repeat in full or part. We introduce a technique for embellishing PCFG-based single-password DTEs to handle vaults in this way, what we refer to as the *sub-grammar approach*. The intuition is that if DTE-decode samples passwords from a smaller domain than the actual trained PCFG, it will often end up using the same password components or full passwords.

In more detail, SG (for sub-grammar) is the following DTE scheme. Encoding first parses all the passwords in $\vec{P}$ using the trained PCFG. It then generates a new sub-grammar PCFG that consists of the cumulative set of rules used in parsing the passwords in $\vec{P}$. The rule probabilities are copied from the original PCFG and then normalized over the sub-grammar PCFG. (We also copy special rule sets described in detail in Appendix A. For example, `T`, the catch-all rule, is always included in the sub-grammar.) This sub-grammar is encoded as the first part of the DTE output, as detailed below. Finally, the DTE separately encodes each $P \in \vec{P}$ as in PCFG, but using the sub-grammar PCFG.

Decoding works in the natural way: first decode the sub-grammar PCFG, then decode the encoding of each password using the resulting sub-grammar PCFG.

Encoding/Decoding of the sub-grammar. Given a canonical representation of the trained PCFG, a sub-grammar can be specified by simply encoding for each non-terminal (except `T`) the number of corresponding rules used by the sub-grammar followed by a list of such rules. Each rule in the list is encoded in the same way as a derivation rule for a password.

To encode the size of a rule set we proceed as follows. Using a set of leaked password vaults in Pastebin (see Section VI-A), we generate the sub-grammar PCFG for all the vaults of size ≥ 2. For each non-terminal in the PCFG (except `T`), we then create a histogram of the number of the non-terminal rules used by each sub-grammar. This gives a per-non-terminal empirical distribution on the number of rules used, which we use as the distribution for sizes that should arise when decoding a random string to a sub-grammar. The DTE encodes this distribution via the inverse transform sampling mechanism of [25].

We have explained now how SG encodes an input $\vec{P}$ in a way that captures structure across passwords, making SG suitable for encoding of password vaults. One additional step is required in the full specification of encode: SG pads out all encodings to a constant length with random bits. This is important because the size of the encoding will otherwise leak the size of the sub-grammar.

VI. Evaluating the Encoders

We have shown how to construct functional NLEs that model real-world password selections by human users of password vaults. To evaluate the quality of these NLEs, we now study their resilience to attack using standard machine-learning techniques.

Recall that in an offline brute-force attack the adversary makes repeated guesses at the master password and decrypts the target vault under each guess. The task of the adversary is to identify the result of decryption under the true master password, i.e., to determine the true plaintext for the vault.

Suppose q is the number of such guessing / decryption attempts. If the true master password is among the adversary's guesses, the result will be $q-1$ random samples from the NLE, as well as the true plaintext, and thus q plaintext candidates in total.

We consider an adversary that orders these q plaintexts in a list from highest to lowest likelihood of being the true vault (in the adversary's view). The adversary's best strategy for attacking the vault is then to make online authentication attempts using one password from each plaintext (decrypted vault) in order from the list. Thus the position of the true plaintext vault in the list indicates the number of online authentication attempts the adversary must make. We evaluate such an attack for an adversary that ignores master password likelihood, and instead uses machine learning (ML) algorithms on the plaintexts to order the list.

Evaluating single decoy passwords. We start by evaluating the security of NLEs for single decoy passwords, and leave full vault analysis to the next subsection. The security goal for a single-password NLE is to produce a decoy password

that is indistinguishable by an adversary from a true, user-selected one. We evaluate security in two ways.

First, we look at the accuracy with an binary adversarial classifier can assign the right label ("true" / "decoy") to a password. Second, we evaluate the ability of such a classifier to assign a high rank to a true password in an ordered list of plaintexts (single passwords) as described above. For this second evaluation, we use the confidence measure of the classifier for a label assignment of "true" as the basis for ranking passwords in the list.

Methodology. We explored a number of approaches to attack, and settled on building machine learning (ML) classifiers to distinguish between true and decoy passwords. We treat this as a supervised learning problem. That means we train a classifier with two sets: labeled true passwords and labeled decoy passwords. We test by drawing from two (disjoint) sources of real passwords and decoy passwords. After experiments with a number of feature and classifier types, we have chosen to report only on the best-performing option, random forest classifiers [12] with 20 estimators using the following features:

(1) *Bag of characters:* This feature captures the frequency distribution of characters in a password. We represent this feature as a vector of integers of size equal to the number of printable characters. We also append the length of the password to the vector.

(2) *n-gram probability:* We train two 4-gram models separately on each of the two classes of password (true and decoy) provided for training. For a given password, we use the probability yielded by each of these two models as a feature. (These two probabilities / features do not sum in practice to 100%, as they would for perfectly complementary models.)

We apply this classifier to the various training set / testing set pairs explored in our experiments.

We evaluate five distinct NLEs as sources of decoys: WEIR-UNIF, WEIR, UNIF, NG, and PCFG. These are all trained using the RockYou training set RY-tr. To generate a decoy using any of these NLEs, we decode a fresh, random bit string of suitable length. As a sixth source of "decoys" we sample directly from RY-tr.

As sources of true passwords, we use the RY-ts, Myspace, and Yahoo data sets. Use of RY-ts creates a case where the NLE is trained using samples from the same data set (but not the same data) as the classifier is tested upon. Use of Myspace and Yahoo data sets creates a case where true passwords originate from a different distribution, which we expect to make the task of distinguishing true from decoy easier for the adversary.

As we have six sources of decoy passwords and three sets of true passwords, we have a total of eighteen true / decoy source pairs on which to conduct our experiments.

For each experiment, given a true / decoy password source pair, we first sample t passwords from the true data set uniformly without replacement to obtain a derived set of true passwords. We set $t = 100,000$ or the size of the true data set, whichever is smaller. We treat the sampled data set as a multiset, meaning that the probability of selecting a password is proportional to the number of times it appears in the set. We also treat the derived data set as multiset, meaning that a given true password can be sampled and thus appear multiple times. We then generate a set of t decoy passwords using the decoy source, i.e., by using the appropriate NLE or sampling from the "decoy" set RY-tr. Using the resulting pair of derived data sets, we do a 10-fold cross-validation of the random forest classifier with 90% / 10% training / testing splits.

NLEs	Myspace		Yahoo		RY-ts	
	α	$\bar{r}$	α	$\bar{r}$	α	$\bar{r}$
WEIR-UNIF	66	24	72	13	82	5
WEIR	63	35	54	36	60	25
UNIF	86	2	97	<2	97	<2
NG	70	22	68	41	61	41
PCFG	70	26	58	39	57	39
RY-tr	70	22	64	50	50	50

Figure 4. For different decoy / true password source pairs, percentage classification accuracy (α) and percentage average rank ($\bar{r}$) of a real password in a list of $q = 1{,}000$ decoy passwords for ML adversary. Lower α and higher $\bar{r}$ signify good decoys.

For our experiments in true / decoy password classification, we measure α, the average accuracy of the classifier on testing data. In those experiments involving ranking of q passwords in order of likelihood of being the true password, we order passwords according to the confidence of the classifier in assigning a "true" label. We measure $\bar{r}$, the average rank of the true password in the resulting list. (Thus $\bar{r}$ is an estimate of the number of online authentication attempts required for a brute-force attacker that uses the classifier to identify the true password.)

An effective classifier will achieve a high value of α and a low value of $\bar{r}$. For example, a classifier that performs no better than random on a given decoy generation algorithm will on expectation achieve $\alpha = 50\%$ and $\bar{r} = (q+1)/2$. A perfect classifier will achieve $\alpha = 100\%$ and $\bar{r} = 1$.

Figure 4 reports the average classification accuracy (α) and average rank ($\bar{r}$) (expressed as a percentage) across our eighteen true / decoy password source pairs. For experiments, we set q = 1,000. In other words, we drew one password from the true password set and inserted it in a randomly selected position among 999 decoys generated from the decoy source. (We chose $q = 1{,}000$ as larger values, e.g., $q = 10{,}000$, yielded similar results in preliminary experiments, but resulted in significantly longer times generating decoys and thus for overall experiment execution.)

Several outcomes of our experiments are notable. As expected, the uniform NLE UNIF does quite poorly, with

the classifier strongly distinguishing it from true passwords. Also as expected, the adversary performs better in nearly all cases against Myspace and Yahoo data than RY-tr, that is, when the adversary trains on the true password source, and the decoy generator designer does not. (As a sanity check, given the use of RY-tr as a source of "decoy" passwords, and RY-ts as a source of true passwords, i.e., a common source for both, the adversary does no better than random guessing in distinguishing true from "decoy.")

It is important to observe that no decoy generator is consistently superior to others across the board. For example, WEIR resists attack best for Myspace and Yahoo data sets, while PCFG is superior in the case of RY-ts. As all decoy generators are trained on RY-tr, these results suggest that WEIR generalizes better than PCFG, in the sense that it can be deployed effectively to protect password sources different from those on which it has been trained. Strikingly, in the task of distinguishing decoys from true passwords drawn from the the Myspace and Yahoo data sets, WEIR generates decoys that are harder for the adversary than "decoys" (true passwords) from RY-tr.

A. Evaluating complete password vaults

We now describe our evaluation of SG, our NLE for generating full decoy vaults.

Our evaluation relies on a set of leaked password vault contents that we obtained from an anonymous public post to Pastebin. We refer to this data set as Pastebin. Pastebin appears to have been gathered via malware running on a number of clients, and is thus suggestive of the kind of data an adversary might exploit. A limitation of the data set is that it takes the form of unordered username / password pairs. Thus the only way to ascertain that two passwords came from the same vault is on the basis of a common or similar associated usernames. We organized the passwords into hypothesized vaults through a manual sanitization procedure on Pastebin whose details we omit for brevity, but which we will publish with our code.

Some statistics on Pastebin are given in Figure 5. Here m denotes the number of passwords in a vault.

m	Statistic	Value
$2 \leq m \leq 3$	Users	100
	Domains	140
	Total no. of unique PWs	155
	Avg. domains/users	2.06
	Avg. PWs/user	1.60
	Median PWs/user	2
$4 \leq m \leq 8$	Users	89
	Domains	329
	Total no. of unique PWs	378
	Avg. domains/users	5.27
	Avg. PWs/user	2.58
	Median PWs/user	2
$9 \leq m \leq 50$	Users	87
	Domains	887
	Total no. of unique PWs	741
	Avg. domains/users	15.01
	Avg. PWs/user	4.21
	Median PWs/user	4
Min. entropy of full data-set		6.13

Figure 5. Summary statistics for Pastebin password vault leak. Here, m is the number of passwords in a vault.

Adversarial classifier. To construct a classifier capable of distinguishing true vaults from those generated by SG, we make use of the following four feature vectors:

(1) *Repeat count:* A vector of three vault features: (1) Number of unique passwords, (2) Number of passwords unique up to leet transformation and capitalization, and (3) Number of unique tokens inside a vault. These counts are normalized by (divided by) the number of passwords in the vault.

(2) *Edit distance:* A vector $\vec{x} =< \vec{x}_1, \vec{x}_2 \ldots \vec{x}_k >$ of size k, where $\vec{x}_i$ denotes the number of passwords pairs within edit-distance i (case insensitive). This feature aims to capture user "tweaking" of passwords across sites, i.e., small modifications made to a password to create unique per-site variants.

(3) *n-gram structure:* A vector $\vec{x}$ of size k, where $\vec{x}_i$ is the percentage frequency of the i^{th} most popular n-gram in the vault, for some n. (We use $n = 5$.) This feature characterizes token reuse among passwords.

(4) *Combined:* By this we denote a combination of all three aforementioned feature vectors.

In all individual vectors, we let $k = 5$. We experimented with several ML engines, including random forest and clustering, but the best-performing was a Support Vector Machine (SVM). We construct four support vector machine (SVM) classifiers with a radial basis function kernel, one for each of the feature vectors given above.

Our training set consists of an equal number of labeled decoy and true password vaults of equal size (m). We use the Pastebin data set for true vaults. (See the "Users" statistic in Figure 5 for the number of available instances.) By analogy with our evaluation of single passwords, the adversarial algorithm runs the SVM on q vaults in turn, and orders these vaults according to the SVM-estimated probability of being in the "true" class.

For each size m, let c_m be the number of true vaults in the Pastebin data-set with m passwords. We perform $(c_m - 1)-$fold cross-validation, meaning we set aside one vault of size m for testing as the true vault and train on all others; we repeat this process for every vault. To generate decoy vaults for training and testing we generate uniformly random bit strings and decode them using the appropriate NLE.

As previously described, the success metric is $\bar{r}$, the average value over all true passwords vaults of the rank r in the adversarially ordered list, expressed as a percentage.

Figure 6 summarizes results for two full-vault NLEs. Recall that the one labeled SG is the sub-grammar NLE built from our PCFG. For comparison, we also consider a naïve

$\bar{r}$ (average rank %)		
Feature used	MPW	SG
Repeat count	0.6	37.4
Edit dist.	1.2	41.4
n-gram	0.6	38.5
Combined	1.0	39.7

$\bar{r}$ (average rank %)		
m	MPW	SG
2–3	13.2	32.8
4–8	0.8	38.0
9–50	0.3	38.9

Figure 6. Performance of ML attacks against NLEs. **(Left)** Average rank of the real vault ($\bar{r}$) over all vault sizes ($m \in [2, 50]$) for MPW and SG using different feature types. **(Right)** Average rank $\bar{r}$ of the true vault obtained with Repeat-count feature vector, broken down by vault size m.

one, labeled MPW, that outputs m independent applications of the PCFG NLE. We set $q = 1,000$. The table on the left compares the different attacks across all vaults of size 2–50 against the two NLEs. As expected, treating passwords as independent in MPW yields decoys that an attacker can easily distinguish from a true vault, with $\bar{r} = 0.6\%$ for two of our adversarial algorithms. In contrast, our sub-grammar approach SG achieves $\bar{r} > 30\%$ in all of our presented experiments.

The right table compares the performance of the Repeat-count feature vector against MPW and SG by vault size. This attack is quite effective against MPW, confirming the observation that treating passwords within a vault independently poorly models real vaults. This series of experiments also brings to light the fact that the security of the NLE degrades as the vault size m increases. Again, SG fares much better.

VII. Honey Encryption for Vaults

Given the NLEs from Section V, we can now build an encryption service for vaults. Our construction uses a similar design as Juels and Ristenpart's [25] honey encryption (HE) construction, which composed a DTE with a conventional password-based encryption scheme. But ours will necessarily be more complicated. We will combine multiple different DTEs (an NLE for human-generated and a regular DTE for computer-generated passwords) and handle side information such as domains in a privacy-preserving manner. We will also need to handle adding new entries to an existing vault and removing entries.

Abstractly, vault encryption takes as input a set of domains $\vec{D} = (D_1, \ldots, D_\ell)$, associated passwords $\vec{P} = (P_1, \ldots, P_\ell)$, and a vector of bits $\vec{h} = (h_1, \ldots, h_\ell)$ for which a 1 signifies that the password was input by a user and a 0 signifies that the password was randomly generated. The client selects randomly-generated passwords by selecting uniformly from a large set (of size about 2^{90}) that includes passwords accepted by all the website policies that we tested.

We now present a basic HE scheme. This scheme hides passwords, but takes the simple approach of storing domains in the clear with the ciphertext.

Basic HE scheme. Upon input $\vec{D}, \vec{P}, \vec{h}$, this scheme proceeds as follows. We first apply the sub-grammar NLE SG to the subset of passwords in $\vec{P}$ for which $h_i = 1$. To each of the remaining passwords with $h_i = 0$ we apply the DTE UNIF. The result from both steps is a bit string $S = S_0 \,\|\, S_1 \,\|\, \cdots \,\|\, S_\ell$ with S_0 the output from encoding the sub-grammar and each S_i either an encoding under PCFG using the sub-grammar ($h_i = 1$) or an encoding under UNIF ($h_i = 0$).

The string S is then encrypted as follows. First, derive a key $K = \mathsf{KDF}(mpw, \mathsf{sa})$ for a freshly generated uniform salt sa and where mpw is the user's master password. Here KDF is a password based key derivation function (PBKDF) that is strengthened to be as slow as tolerable during normal usage [27]. Then, encrypt each S_i (for $i = 0$ to ℓ) independently using AES in counter mode with key K and a fresh random IV. This produces a sequence of $\ell + 1$ CTR-mode ciphertexts $\vec{C} = (C_0, \ldots, C_\ell)$. The final vault ciphertext includes a (conventional) encoding of $\vec{D}, \vec{h}, \mathsf{sa}, \vec{C}$. Decryption works in a straightforward way.

This HE scheme is relatively simple (once the NLE and DTE are fixed) and space efficient. However, $\vec{D}$ and $\vec{h}$ are stored in the clear and this means that attackers obtaining access to the ciphertext learn the domains for which the user has an account as well as which passwords were randomly generated. One approach to rectify this would be to specify a DTE for encoding $\vec{D}$. One could use popularity statistics for domains, for example based on Alexa rankings. However note that this may sacrifice security against offline brute-force attacks in the case that an attacker knows with high certainty the set of domains associated with a user's vault.

This highlights a delicate challenge in the use of HE: if an attacker can easily obtain knowledge about a portion of the plaintext, it may be better to not apply HE to that portion of the plaintext. We may view domain information as such easily-obtainable side information that it is not worth encrypting. To provide domain privacy, though, we must do more. We now describe two approaches: HE-DH1 and HE-DH2.

HE-DH1. In this scheme, we hide an individual user's domains in the set of all domains used by users and include in each user's vault "dummy" entries for unused domains. To achieve privacy of domains the goal will be that an attacker cannot distinguish between a dummy and real entry. In the following, we fix a set of popular domains $\vec{D}^* = (D^*_1, \ldots, D^*_{s_1})$. We require that $\vec{D}^*$ be a superset of all the domains used by users. We discuss how to achieve this momentarily.

To encrypt an input $\vec{D}, \vec{h}, \vec{P}$ we first encode the passwords as described above for the basic scheme: apply SG to the set of human-generated passwords and UNIF to each of the computer-generated passwords. For any domain in $\vec{D}^*$ but not in $\vec{D}$, we generate a dummy encoding as follows. First we choose a bit h to select whether this domain should have a human-generated dummy entry or a computer-generated

one. We discuss how to bias this bit selection below. Then we generate a random bit string of length equal to the length of outputs of PCFG (if $h = 1$) or UNIF (if $h = 0$). We then generate the full encoding $S = S_0 \,\|\, S_1 \,\|\, \cdots \,\|\, S_{s_1}$ by inserting the per-password encodings or dummy encodings to match the order from $\vec{D}^*$. We then encrypt S as in the basic scheme.

The distribution alluded to above for dummy encodings does not affect confidentiality of mpw or $\vec{P}$, but rather confidentiality of the domains associated to a user and whether the user has human or generated passwords for each such domain. One can, for example, set this distribution to be biased towards human-generated passwords.

Note that decryption with either the correct mpw or an incorrect one produces s_1 passwords. The $s_1 - \ell$ honey passwords corresponding to the dummy entries obscure from an attacker which sites are in use by the user (even in the extreme case that the attacker has guessed mpw somehow). When mounting brute-force attacks, the dummy entries might hinder attempts to perform online checks of recovered passwords. In particular, if a domain's login web page follows best practices and does not leak whether a user has an account there (regardless of correctness of the provided password), then the attacker may not be able to distinguish between the situation in which a certain domain is not used by a user and the situation in which the decryption attempt resulted in a honey password.

The downside of the above approach is that s_1 may need to be very large and for each user the storage service (described below in Section VIII) must store $\mathcal{O}(s_1)$ bits of data. We can grow s_1 over time by having clients inform the service of when a new domain should be added to D^* and the server can insert dummy entries in previous vaults by just inserting random bit strings in the appropriate location for each vault ciphertext. (this is possible because a separate CTR-mode encryption is used for each vault entry.) When the system is first setup, an initial relatively small popular domains list can be seeded with highly popular domains.

Another approach to reducing overheads is to bucket users into separate groups, each group having their own popular domains list. This enables tuning the size of vaults relative to per-group domain confidentiality.

HE-DH2. In this scheme, we adopt an alternative approach to dealing with the long tail of domains, and use a honey-encrypted overflow table. Fix some number $s_2 > 0$. For each of the domains not in the current popular domain set, we use the following procedure. First apply the PCFG (using the sub-grammar or UNIF appropriately to the password to get a bit string S'. Then hash the domain name and take the result modulo s_2 to yield an index $j \in [0..s_2 - 1]$. Set $S^*_{s_1+j+1}$ to S'. Some indices in $[0..s_2 - 1]$ will be unused after handling all domains outside the popular domain set; we fill these with dummy encodings. The additional s_2 seeds are each encrypted with CTR mode, making the final, full ciphertext $\vec{D}^*, \vec{h}^*, \mathsf{sa}, \vec{C}$. Note that now, $\vec{C}$ contains $s_1 + s_2 + 1$ individual CTR-mode encryption ciphertexts for the sub-grammar and $s_1 + s_2$ individual, possibly dummy, passwords.

By setting s_2 large enough relative to the expected number of domains not in D^* we can ensure that with high probability no two domains hash to the same location. Note that the domains associated with the overflow table are not stored with the ciphertext. To decrypt, the requested domain is checked to see if it is in D^* and if not it is hashed to find the appropriate entry in the last s_2 HE ciphertexts.

Updating a vault. To update a password for a particular domain in the basic scheme or HE-DH1, one first decrypts the entire vault, changes the appropriate entry, and then encrypts the modified vault with fresh randomness (including the salt). needed to ensure the sub-grammar is consistent with the encoded content. For HE-DH2, one proceeds much the same, also decrypting each of the s_2 entries in the overflow table. The appropriate domain's entry is updated (found either by looking in the popular domains list or, failing that, hashing the domain to be updated to find it in the overflow table). Finally, the modified vault is encrypted (with fresh randomness).

Deletion of a password can be performed by converting the appropriate entry into a dummy entry while also updating the sub-grammar by removing any now unnecessary rules.

Security discussion. Our primary goal is confidentiality of the plaintext passwords. All passwords are first encoded using an appropriate DTE and then encrypted using a PBE scheme. Should the user's master password be strong, even an offline brute-force attack is infeasible and, in particular, it will require as much work to break any of the schemes above as would be to break a conventional PBE encryption. Should the user's master password be weak, then by construction decrypting the ciphertext under any incorrect master password gives back a sample from the DTE distribution. In particular, we believe there to be no speed-up attacks that allow the attacker to rule out a particular incorrect master password without having to determine if the recovered plaintext is decoy or not. As we showed in the previous section, our NLE is good enough that distinguishing human-generated passwords is challenging even for sophisticated adversaries.

The above is admittedly informal reasoning, and does not rule out improved attacks. We would prefer a formal analysis of plaintext vault recovery security akin to those given for simpler honey encryption schemes in [25], which would reduce security to solely depend on DTE quality. Those techniques rely on a closed-form description of the distribution of password vaults as produced by decoding uniform strings. Unfortunately we do not know how to determine one; even estimating the distribution of single passwords is impractical with sampled data [5]. Formal analysis remains an interesting open question.

If an attacker obtains the encryptions of a vault before and after an update, then security falls back to that of conventional PBE. One simply decrypts both vaults under each guessed master password, and with high probability the contents of the two plaintext vaults will match (except where updates occurred) with high probability only with the correct master password. This is a limitation of all decoy-based approaches we are aware of and finding a solution for update security is an interesting open question.

Another security goal is domain hiding. As discussed earlier, adding dummy ciphertexts (random bit strings) for the latter two schemes for unused domains means that an offline attacker will recover passwords for these domains as well. The same reasoning extends to the use of the overflow table. The complexity of the sub-grammar may leak some information about the overall number of human-generated passwords in-use, but not which of the domains marked as having human-generated passwords are dummy encodings.

VIII. The NoCrack System

We now turn to the design of the full honey-vault service that we call NoCrack. Our architecture closely follows deployed commercial systems, such as LastPass[6]. A web-storage service exposes a RESTful web API over HTTPS for backing up user vaults and synchronizing vaults across devices. To achieve the security benefits of HE, however, we must design this service carefully.

The challenge of password-based logins. One encounters an interesting challenge when attempting to build a decoy-based system which supports backup of user vaults: how to authenticate users to the service that is responsible for backing up their vaults. In particular, the status quo in industry is for users to choose a username and service password. The password would be sent over HTTPS to the server, hashed, and stored to authenticate future requests. But customers are likely to choose this service password to be the same as their vault's master password. If an attacker compromises the storage service and obtains both a user's encrypted vault and the service password hash, they can mount a brute-force attack against the service password hash, learn the service password, and then decrypt the vault.

One might attempt to mitigate with this by securing the password hash separately from vaults. Or one could avoid backup of encrypted vaults entirely, but this would leave users responsible and violate our goal of matching features of existing services. We therefore go a different route, and forego password-based login to the storage service completely.

Device enrollment. A new user registers with the service by providing an email address (also used as an identifier), to which a standard proof-of-ownership challenge is sent. To hinder abuse of the registration functionality, the service can rate limit such requests and require solution of an appropriate CAPTCHA [38]. The proof-of-ownership is an email including a randomly generated 128-bit temporary token (encoded in Base64 format, 22 characters long). The user copies this temporary token into the client program which submits the token over an enroll API call. The server verifies the temporary token, and returns to the client program a (long term) bearer token (also 128 bits) that can be used as a key to authenticate subsequent requests using HMAC. At this stage the client device is enrolled. Note that all communication is performed over TLS.

Additional devices can be enrolled in a similar manner by having an already-enrolled client device to generate a token for the new device or sending a new temporary token via email. Should a user lose all access to a device with a current bearer token, they can easily obtain a new token via the same enrollment process.

We note that two-factor authentication would be straightforward to support by requiring a proof-of-ownership of a phone number or a correct hardware token-generated one-time password to obtain a device bearer token.

Synchronizing with the server. An enrolled client device can compare their local information with that stored under their account on the server. This involves ensuring the client and storage service have the same version of the vault, which, in normal usage, is cached on the client device. To save bandwidth, downloads and uploads can be done in an efficient manner via any standard "diff" mechanism — in particular our HE schemes support sending only portions of the ciphertext at a time.

The client. We currently have only a command line client supported, but future versions could easily integrate with popular browsers via an extension. The client caches the vault locally, but never stores it in the clear on persistent storage. The client queries the service when run to determine if it needs to synchronize the vault. At the beginning of a browsing session, the user is prompted for the master password and the vault is decrypted. To check for typos, we can use dynamic security skins [16] (as suggested also for use with Kamouflage), which show a color or picture that is computed as a hash of the master password (but never stored). The output of the KDF can be cached in memory in order to decrypt individual domains as needed, while the master password itself is expunged from memory immediately.

Note that the HE scheme does not handle login names; we assume that browser caching mechanisms can handle this for a user if they desire. Should a login detectably fail for the user due to master password typo and the user does not observe the incorrect security skin, the client can prompt the user to reenter their master password. By construction, there might be dummy password entries in NoCrack for some

[6] http://www.lastpass.com

Operation	$s = 2$	200	2,000	20,000
Recover password	6.34 ms	6.41 ms	6.42 ms	6.50 ms
Add password	0.13 s	0.68 s	1.11 s	9.25 s
Vault size on disk	4.71 KB	164.00 KB	1.55 MB	15.26 MB

Figure 7. Running times (median over 100 trials) of operations for different vault sizes $s = s_1 + s_2$. The final row is size of encrypted vaults on disk.

domains where the user does not have an account. The user and/or the browser is responsible to distinguish the domains where the user has an account.

Implementation and performance. We implemented a prototype of NoCrack in Python-2.7. On the server side we used Flask and Sqlite3. To normalize domains we use the Python Public-Suffix library. All cryptographic operations use PyCrypto-2.6.1. We use AES within CTR mode encryption, and SHA-256 within PBKDF2 for key derivation. Many of the operations are parallelizable; we use the Python multiprocessing library for this but note that our prototype implementation does not yet fully take advantage of parallelization. The client and server consist of 3,102 total lines of code as counted by the utility cloc (not counting libraries). All experiments were performed on an Intel Core-i5 with 16 GB of RAM running Linux.

We provide some basic performance numbers for our most complex honey encryption scheme HE-DH2, but emphasize that this is a naive implementation and some improvements will be easy. We fix various vault sizes $s \in \{2\,,\ 200\,,\ 2{,}000\,,\ 20{,}000\}$ and set $s_1 = s_2 = s/2$ (these are the sizes of the popular domains table and overflow table, respectively). We used integer representation size $b = 128$. for encoding fractions. We start by generating a random ciphertext of size appropriate for the values of s_1, s_2 assuming some short arbitrary domain size and that all passwords are human generated (the worst-case for performance). We then measure the time to recover a particular vault password as well as to add a password to the vault. We report in Table 7 the median times over 100 trials. Variance in timing was negligible.

Time for recovering a single password is fast, and agnostic to the size of the vault. This is because our design allows random access into the vault. Time for adding passwords increases with s, since our scheme decrypts and decodes all s entries, updates the new password, then re-encodes and re-encrypts all s entries (this is required to keep the sub-grammar synchronized with the contents of the vault.) The bulk of the time is spent in encoding and re-encoding passwords. This operation is still only around one second for large vaults, and large vaults are needed only to support domain hiding. The encrypted vaults are also of reasonable size. We conclude that, while NoCrack does incur time and space overheads relative to conventionally encrypted vaults, the absolute performance is more than sufficient for the envisioned usage scenarios.

IX. Related work

Honey objects. The use of decoy objects such as honeypots or decoy documents [8], [9] is well-established in information security practice. More closely related to our work here are honeywords [26], decoy passwords associated with each user in a password database. The honeywords system involves fake individual passwords, rather than password sets, and does not help with decoy security for password vaults, our goal here.

We also note that decoy document and honeyword systems are distributed: they assume explicit storage of secrets that distinguish decoy from real objects in a trustworthy location (a "honeychecker") separate from the system containing the decoy objects. See [24] for a discussion of the distinction between such systems and those in which these secrets (e.g., master passwords) are provided by a user, as in NoCrack.

An early decoy system involving encryption under user-furnished secrets was proposed by Hoover and Kausik [22]; it only supports encryption of specially crafted RSA private keys. Honey encryption [25] introduced a general framework for incorporating honey objects into encryption. As explained earlier, it does not prescribe constructions for specific message types, which gives rise to one of the major technical challenges we faced in building NoCrack.

A detailed discussion of Kamouflage [4] was given in Section II.

Password-based key derivation. Key stretching, where one slows down key derivation, was first defined by Kelsey et al. [30], and standardized later in PKCS#5 [27]. Boyen proposed halting password puzzles [11] in which the key-derivation will run indefinitely on incorrect password guesses and only terminates (after an unspecified length of time) upon correct guesses. Another approach is to incorporate memory-hard functions, which require a significant amount of RAM to compute efficiently, such as done in scrypt [35]. Each of these techniques slows down offline brute-force attacks, but do not force attackers to make online queries.

Stateless password managers. Several schemes exist for strengthening user passwords (and preventing direct password reuse) by hashing a master secret with domain names to dynamically generate per-domain passwords. An early example was the Lucent Personal Web Assistant (LPWA) [19]; later variants include PwdHash [36] and Password Multiplier, a scheme by Halderman et al. [20]. Chiasson et al. conducted a usability study of both PwdHash and Password Multiplier and found the majority of users could not successfully use them as intended to generate strong passwords [14]. Another usability challenge is dealing with sites with a password policy banning the output of the password hash;

for this reason NoCrack uses a simple set of rules for computer-generated passwords.

Password managers. In addition to Kamouflage [4], several academic proposals have sought to improve the usability and security of stateful password managers. Passpet [41] generates random passwords per-domain and allows users to assign avatars to different websites to easily identify which passwords are used with which website. Tapas [33] is a prototype two-factor password manager which distributes passwords into shares between a computer and a mobile phone.

Karole et al. [28] performed a usability evaluation comparing three common approaches to password vaults: online services, phone applications, and USB tokens. Interestingly, they found that the online service was by far the easiest for participants to use, although participants stated a clear preference for the phone-based solution because most didn't want to entrust all of their passwords to a cloud-based service. These findings are a compelling justification for NoCrack, which enables the convenience of cloud-based password vault backup with higher security against compromise.

ACKNOWLEDGMENTS

We thank the Oakland 2015 anonymous reviewers for their valuable comments and feedback. We thank Michael Doescher for helping in several design choices of PCFG construction and cleaning the Pastebin dataset, Shoban Preeth Chandrabose for his feedback on the machine learning analysis of NoCrack, and Adam Everspaugh for valuable discussions and editorial assistance. This work was supported in part by NSF grants CNS-1330308 and CNS-1253870 and AFOSR grant FA9550-13-1-0138.

REFERENCES

[1] "Wikitionary: Frequency List," http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.

[2] M. Bellare, T. Ristenpart, and S. Tessaro, "Multi-instance security and its application to password-based cryptography," in *Advances in Cryptology – CRYPTO 2012*. Springer Berlin Heidelberg, 2012, pp. 312–329.

[3] S. Billot and B. Lang, "The structure of shared forests in ambiguous parsing," in *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*, ser. ACL '89. Stroudsburg, PA, USA: Association for Computational Linguistics, 1989, pp. 143–151. [Online]. Available: http://dx.doi.org/10.3115/981623.981641

[4] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh, "Kamouflage: Loss-resistant password management," in *ESORICS*, 2010, pp. 286–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=1888881.1888904

[5] J. Bonneau, "Guessing human-chosen secrets," Ph.D. dissertation, University of Cambridge, May 2012. [Online]. Available: http://www.cl.cam.ac.uk/~jcb82/doc/2012-jbonneau-phd_thesis.pdf

[6] J. Bonneau, "The science of guessing: analyzing an anonymized corpus of 70 million passwords," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 538–552.

[7] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, "The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes," in *2012 IEEE Symposium on Security and Privacy*, May 2012. [Online]. Available: http://www.jbonneau.com/doc/BHOS12-IEEESP-quest_to_replace_passwords.pdf

[8] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo, *Baiting Inside Attackers Using Decoy Documents*, 2009, pp. 51–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05284-2_4

[9] B. M. Bowen, V. P. Kemerlis, P. Prabhu, A. D. Keromytis, and S. J. Stolfo, "Automating the injection of believable decoys to detect snooping," in *WiSec*. ACM, 2010, pp. 81–86. [Online]. Available: http://doi.acm.org/10.1145/1741866.1741880

[10] R. Bowes, "Skull Security, Passwords," https://wiki.skullsecurity.org/Passwords.

[11] X. Boyen, "Halting Password Puzzles – Hard-to-break Encryption from Human-memorable Keys," in *16th USENIX Security Symposium*. Berkeley: The USENIX Association, 2007, pp. 119–134, available at http://www.cs.stanford.edu/~xb/security07/.

[12] L. Breiman, "Random Forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[13] C. Castelluccia, M. Dürmuth, and D. Perito, "Adaptive password-strength meters from markov models." in *NDSS*, 2012.

[14] S. Chiasson, P. van Oorschot, and R. Biddle, "A Usability Study and Critique of Two Password Managers," in *Proceedings of the 15th USENIX Security Symposium*, 2006.

[15] M. M. Devillers, "Analyzing password strength," *Radboud University Nijmegen, Tech. Rep*, 2010.

[16] R. Dhamija and J. D. Tygar, "The battle against phishing: Dynamic security skins," in *Proceedings of the 2005 symposium on Usable privacy and security*. ACM, 2005, pp. 77–88.

[17] G. Dick and H. Ceriel, "Parsing techniques, a practical guide," Technical Report, Tech. Rep., 1990.

[18] T. Fujisaki, F. Jelinek, J. Cocke, E. Black, and T. Nishino, "A probabilistic parsing method for sentence disambiguation," in *Current Issues in Parsing Technology*. Springer, 1991, pp. 139–152.

[19] E. Gabber, P. B. Gibbons, Y. Matias, and A. J. Mayer, "How to Make Personalized Web Browising Simple, Secure, and Anonymous," in *FC '97: Proceedings of the 1st International Conference on Financial Cryptography*. London, UK: Springer-Verlag, 1997, pp. 17–32.

[20] J. A. Halderman, B. Waters, and E. W. Felten, "A Convenient Method for Securely Managing Passwords," in *WWW '05: Proceedings of the 14th International Conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 471–479.

[21] M. E. Hellman, "A cryptanalytic time-memory trade-off," *Information Theory, IEEE Transactions on*, vol. 26, no. 4, pp. 401–406, 1980.

[22] D. Hoover and B. Kausik, "Software smart cards via cryptographic camouflage," in *IEEE Symposium on Security and Privacy*. IEEE, 1999, pp. 208–215.

[23] M. Jakobsson and M. Dhiman, "The benefits of understanding passwords," in *Mobile Authentication*. Springer, 2013, pp. 5–24.

[24] A. Juels, "A bodyguard of lies: the use of honey objects in information security," in *SACMAT*, 2014, pp. 1–4.

[25] A. Juels and T. Ristenpart, "Honey Encryption: Beyond the brute-force barrier," in *Advances in Cryptology – EUROCRYPT*. Springer, 2014, pp. 523–540.

[26] A. Juels and R. Rivest, "Honeywords: Making password-cracking detectable," in *ACM Conference on Computer and Communications Security – CCS 2013*. ACM, 2013, pp. 145–160.

[27] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," 2000, RFC 2289.

[28] A. Karole, N. Saxena, and N. Christin, "A comparative usability evaluation of traditional password managers," in *Information Security and Cryptology - ICISC 2010*, ser. Lecture Notes in Computer Science, K.-H. Rhee and D. Nyang, Eds. Springer Berlin Heidelberg, vol. 6829, pp. 233–251. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-24209-0_16

[29] P. Kelley, S. Komanduri, M. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. 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)*, 2012, pp. 523–537.

[30] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, "Secure applications of low-entropy keys," in *Information Security*. Springer, 1998, pp. 121–134.

[31] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[32] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2014, pp. 689–704.

[33] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: Design, Implementation, and Usability Evaluation of a Password Manager," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 89–98. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420964

[34] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *Advances in Cryptology-CRYPTO 2003*. Springer, 2003, pp. 617–630.

[35] C. Percival, "Stronger key derivation via sequential memory-hard functions," 2009.

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

[37] R. Veras, C. Collins, and J. Thorpe, "On the semantic patterns of passwords and their security impact," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[38] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, "CAPTCHA: Using hard AI problems for security," in *Advances in Cryptology—EUROCRYPT 2003*. Springer, 2003, pp. 294–311.

[39] 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)*, 2009, pp. 162–175.

[40] L. Whitney, "LastPass CEO reveals details on security breach," *CNet*, May 2011.

[41] K.-P. Yee and K. Sitaker, "Passpet: convenient password management and phishing protection," in *Proceedings of the second symposium on Usable privacy and security*. ACM, 2006, pp. 32–43.

APPENDIX

A. Constructing a PCFG

For completeness we describe our PCFG construction process, which uses techniques, with a few small embellishments, from prior work [37], [39]. We start by creating a parser for passwords based on some dictionaries and/or manual tuning. The parser itself uses a *base PCFG*. Using it, the passwords in a password leak are parsed into 'tokens' and the frequency of each token and each token combination is computed to generate a *trained PCFG*.

The process starts with input a password corpus (a list of passwords selected by real users in the wild) as well as a source dictionary. The dictionary is a list of words which we might expect to appear as or within passwords, with accompanying information about such words' expected likelyhood of occurrence. We construct a dictionary from the following two sources. The first, denoted **D1**, is the Wiktionary [1] list of the top 30,000 English words along with their frequency count as found in books of Project Gutenberg up through April 16, 2006. The second, denoted **D2**, is a list of Facebook first and last names and frequencies compiled by SkullSecurity [10]. After removing names with frequency less than five the final size of **D2** was 1,678,411. The Facebook first and last names capture not only names, surnames of persons, cities and popular entities but also a comprehensive list of popular words used by internet users.

For example, **D2** covers modified spellings of words like 'gurl', acronyms like 'ciao' or 'afaik', etc.

Of course one could use other dictionaries for different languages, cultures, etc., but we leave the evaluation of such localization to future work.

The base weighted CFG. After manual inspection of the RockYou password corpus, we constructed an underlying grammar (CFG) for the base CFG of the following form:

```
S → W | D | Y | K | R | WW | WD | WY
    | WYD | WDY | WK | ...
W → <english-words>T | <names>T
D → <dates> | <numbers>
Y → <symbols>
K → <keyboard-sequence>
R → <repeated-characters>
T → Capitalize | ALL-CAPS | all-lower
    | L33t
```

where "..." signifies all remaining combinations of the non-terminals `W`, `D`, `Y`, `R` and `K`. The variables on the right hand side of each rule written within angle brackets (`<.*>`) are terminal place-holders. `<english-words>` and `<names>` are placeholders for sets of non-termanals associated to **D1** and **D2** respectively, while the rest of the placeholders are initialized with regular expressions. We explain the placeholders below in detail. The grammar is ambiguous (i.e., it can parse a password in multiple ways): 'password2015' could be parsed as {'pass', 'word', '2015'} or {'password', '2015'}. Obviously the latter seems like a more meaningful parsing. To disambiguate among different parse trees we assign some heuristic cost to each rule and we used the parse tree that incurs minimal cost. The costs can be seen as weighting various parsings, which is why we refer to the resulting CFG as weighted.

For the placeholder `<english-words>`, we specify a non-terminal rule for each word in **D1**. The cost of each rule for $X \in$ **D1** is $1 - \frac{f_X}{\sum_X f_X}$, where f_X denotes the frequency count of X in **D1**. For the placeholder `<names>`, the cost function is defined analogously, but using **D2**.

For the remaining non-terminals, we created simple regular expressions or rules that recognizes the underlying languages as closely as possible. Non-terminal `<dates>` produces dates in common formats (e.g. '08232013', '19790925', 2008 etc). We created a simple regex to parse dates; we found through hand tuning that we achieved the most reliable parsing by setting the cost of a date string str to be $\frac{|\text{str}|}{8}$ where |str| is the string's length. The constant 8 was chosen as the most popular date-based password length in practice and also the maximum length date that our regex will accept.

The non-terminal `<keyboard-sequence>` includes patterns of standard US keyboard sequences (e.g. `'qwerty'`, `'zxcvbn'` etc). We achieved the most reliable parsing by setting the cost of a sequence str to be proportional to $\frac{c}{|\text{str}|}$ where c is the number of lateral changes in direction performed while typing 'str' on a standard US keyboard, a basic metric of typing complexity.

The non-terminal `<repeated-characters>` tries to predict whether a string is a sequence of identical characters, such as `'aaaaaaa'`, `'0000000'`; we set the cost of this rule to $\left(\frac{\text{\# unique characters in str}}{|\text{str}|}\right)^{|\text{str}|}$.

The non-terminal `T` leads to one of four special terminals that symbolize transforms applied in a post-processing step to the preceding string str generated by `W`. `ALL-CAPS` capitalizes all letters in the preceding string; `Capitalize` capitalizes the first letter; and `all-lower` sets all letters to lowercase. `L33t` applies a number of leetspeak transformations or selected capitalization. The cost for each of these rules is set to $\frac{c}{|\text{str}|}$, where c is 0 for `all-lower`, 1 for `ALL-CAPS` and `Capitalizes`, and the number of transformed characters for `L33t`.

The resulting base weighted CFG permits the parsing of passwords in a password corpus, yielding insight into the structure of human-generated passwords. We consider for a given input the parse tree with the lowest cost under the heuristic explained above (i.e., sum of costs for rules in the parse tree). To identify the minimum cost parse tree we use a standard bottom-up dynamic programming approach similar to CYK [18]. Below, the term "parse tree" refers to the parse tree with minimum cost unless explicitly stated otherwise.

The trained PCFG. We now explain how we train a PCFG on the RockYou corpus, bootstrapping from the base weighted CFG.

We first fix some definitions used in this process. A production rule $l \to r$ can be specified as a pair (l, r). Every edge in a parse tree is a rule. A *rule set* is a lexicographically ordered set of rules with the same left-hand-side. A *rule list* is an ordered list of rules generated by depth-first search of the parse-tree of a string / password (with siblings taken in left to right order).

A CFG is completely specified as a set of rule sets. A PCFG is completely specified by an admissible assignment of probabilities to CFG rules. Let $\mathcal{S}$ be a rule set of size $|\mathcal{S}|$ and $p_{\mathcal{S}}(l \to r)$ be the probability of a rule $l \to r$ in $\mathcal{S}$. An admissible assignment of probabilities has the property that $\sum_{(l \to r) \in \mathcal{S}} p_{\mathcal{S}}(l \to r) = 1$. We refer to the probability distribution over rules in a rule set $\mathcal{S}$ for a given admissible assignment as its *induced* probability distribution.

To train a PCFG on our training corpus, we first parse each password using the base weighted CFG to obtain a parse tree for each password. The aggregate set $\mathcal{L} = \{R_1, \ldots, R_w\}$ of all rules in all these parse trees specifies a CFG. A rule set $\mathcal{S}$ in this CFG is simply a set of all rules with a common left-hand side.

To specify a PCFG based on this CFG, we make use of the count of repetitions of each rule in $\mathcal{L}$ to construct an

admissible assignment of probabilities. Let $f_{l\to r}$ denote the number of repetitions (the "frequency") of rule $l \to r$ in $\mathcal{L}$ and $f_{\mathcal{S}}$ denote the total number of repetitions of all rules in $\mathcal{S}$. Letting $p_{\mathcal{S}}(l \to r) = f_{l\to r}/f_{\mathcal{S}}$ yields an admissible assignment. We refer to this PCFG as the *trained PCFG*.

The non-terminals in the trained PCFG are as in the base PCFG, except that any special placeholder non-terminals that used post-processing (e.g., `L33t`) are at this stage replaced by a concrete (finite) set of non-terminals and/or terminals.

B. Securely Encoding Fractions

We have to define an encoding scheme for floating point numbers. We focus on empirically derived distributions, for which each probability that needs encoding is a fraction p/q where p is the frequency count and q is some cumulative total number of values observed. We have that $p \leq q$. In such cases, encoding can be done by choosing a large b-bit number subject to the constraint that it should be equal to p modulo q. For correctness, b has to be larger than number of bits in the binary representation of q. Encoding and decoding are defined below:

encode$_q(p)$**:**	**decode**$_q(r)$**:**
1. $x \leftarrow\!\!\$\ \{0,1\}^b$	1. $p \leftarrow r \bmod q$
2. $r \leftarrow x + p - (x \bmod q)$	2. **return** p
3. **if** $r \geq 2^b$ **then**	
4. $\quad r \leftarrow r - q$	
5. **return** r	

Correctness is straightforward: by construction, $0 \leq r < 2^b$ and so

$$\begin{aligned} r \bmod q &= (x + p - (x \bmod q)) \bmod q \\ &= (x \bmod q) + (p \bmod q) - (x \bmod q) \\ &= p \bmod q\ . \end{aligned}$$

In terms of security of encoding, we want the output values of the encoding scheme to be uniformly distributed over $\{0,1\}^b$ when p is uniformly selected from $[0, q-1]$. Otherwise an adversary can possibly rule out master passwords using the fact that encoded strings are distributed differently from uniform bits during decryption (under the wrong master password). Just for convenience we define $h = 2^b \bmod q$ and $l = \lfloor \frac{2^b}{q} \rfloor$. (Thus, $lq + h = 2^b$.) Now, consider four random variables A, B, C and D defined as follows: A and B are uniform distributions over $\{0,1\}^b$ and $\mathbb{Z}_q$, respectively; $C = A - (A \bmod q)$; and D is the output of the **encode** function for a randomly chosen $p \in [0, q-1]$. Note that D is almost equal to $C + B$, but for the potential additional modification made should $C + B$ has value equal to or larger than 2^b (see the conditional of line 3 of **encode**). Note also that C always takes values of the form kq for some $k \in \mathbb{Z}_{l+1}$. So, we have that the probability mass function for C is defined by:

$$\Pr[\,C = kq\,] = \begin{cases} \frac{q}{2^b} & \text{if } 0 \leq k < l \\ \frac{h}{2^b} & \text{if } k = l \end{cases}$$

Using this, we can see that for D it holds that:

$$\Pr[\,D = r\,] = \begin{cases} \frac{1}{2^b} & \text{if } r \in [0,\ (l-1)q + h) \\ \frac{1}{2^b} + \frac{h}{2^b q} & \text{if } r \in [(l-1)q + h,\ lq) \\ \frac{h}{2^b q} & \text{if } r \in [lq,\ 2^b) \end{cases}$$

The statistical distance between A and D, denoted by $\Delta(A, D)$, can then be calculated as follows.

$$\begin{aligned} \Delta(A, D) &= \sum_{x \in \{0,1\}^b} |\Pr[\,A = x\,] - \Pr[\,D = x\,]| \\ &= \frac{(q-h)h}{2^b q} + \frac{h(q-h)}{2^b q} \\ &\leq \frac{2(q - q/2)(q/2)}{2^b q} \leq \frac{q}{2^{b+1}} \end{aligned}$$

In a similar way we can show using a union bound that, while encoding m fractions, together the encoded values deviate from a uniform distribution over such values by at most $mq_{\text{max}}/2^{b+1}$, where q_{max} is the largest q we shall need during encoding of fractions and $2^b \gg q_{\text{max}}$. We found that we never encountered q more than 2^{32}, and do not need $m > 10^4$. Setting $b = 128$, and plugging in $q_{\text{max}} = 2^{32}$ and $m = 10^4$ gives that the statistical distance from uniform for m encodings is strictly less than 2^{-83}, which suffices for our purposes.