Massachusetts Institute of Technology

Honey Encryption Applications

Implementation of an encryption scheme resilient to brute-force attacks

Authors:
Nirvan Tyagi [ntyagi]
Jessica Wang [jzwang]
Kevin Wen [kevinwen]
Daniel Zuo [dzuo]

6.857 Computer &
Network Security

Spring 2015

13 May 2015

Contents

Abstract. Honey encryption is a new encryption scheme that provides resilience against brute force attacks by ensuring that messages decrypted with invalid keys yield a valid-looking message. In this paper, we present our implementation of honey encryption and apply it to useful real-world scenarios such as credit cards and basic text messaging. We also extend the basic honey encryption scheme to support public-key encryption. Finally, we discuss the limitations we faced in our implementations and further requirements for strengthening our applications.

1 Introduction

In the context of computer security, the term honey is used to describe a false resource designed to deflect or counteract attacker attempts of a system. False servers that are set up to distract attackers are known as honeypots. Honeywords are false passwords stored in a hash table, that when stolen can help detect an infiltration. In this project, we will be building on a message encryption scheme known as honey encryption.

Most current encryption schemes use an n-bit key, where the security of the encryption increases with the size of the key. Although we consider these schemes secure, with enough computational power they are vulnerable to brute-force attacks. The decryption of a ciphertext through brute-force guessing of keys can be confirmed with a valid-looking message output, but more importantly, an invalid-looking output as confirmation of an unsuccessful attempt. Honey encryption offers a solution to this vulnerability for certain types of messages. A ciphertext that is honey-encrypted has the property that attempted decryptions with invalid keys yield valid-looking output messages. Thus, attackers employing a brute-force approach gain no information from guess and checking of keys.

Juels and Ristenpart [4] proposed this concept of honey encryption specifically in the context of passwords. After a leakage of millions of real user passwords, it was observed that a significant number of people used weak, easily-predictable, and repeated passwords. Password-based encryption (PBE) and hashing techniques both carry the same vulnerabilities to brute-force guessing attacks due to the predictability of user-generated passwords. By employing honey encryption in lieu of traditional PBE, the certainty of an attacker for successful decryption of a password is weakened.

The core innovation of the honey encryption scheme is the distribution-transforming encoder (DTE), which maps the space of plain-text messages to a seed space of n-bit strings. The DTE takes into account a probability distribution of the message space and assigns a corresponding ratio of bit strings to the message. The intuition lies in the fact that all potential decryptions, regardless of correctness, map to some message and since possible decryptions are assigned via the expected probability distribution, the attacker gains no information. Constructing a suitable DTE for various applications of honey encryption requires an understanding of the message space distribution.

The contributions of this project fall into two main categories. The first is the implementation of the honey encryption scheme and its application to a variety of message spaces. We describe our approach to handling message spaces formed by a generic alphabet, credit card numbers, and simple text messaging. The second part includes extending the basic honey encryption scheme to support Public-key Encryption.

2 Previous Work

Many working systems in today's world rely on user-inputted secrets, such as password-based encryption (PBE). These secrets are inherently weak and of low entropy due to a fundamental problem in how they are generated - users pick easy-to-remember and thus weak passwords. Because of the drastically smaller space these secret keys are being generated from, systems that rely on this type of encryption are susceptible to brute-force guessing attacks. Honey encryption [4] explores a new approach to providing security against brute force attacks. Honey encryption aims to build a scheme where attackers gain no information about the message, even after trying every possible password. When a ciphertext is decoded with an invalid key, a seemingly valid message is produced. Not only is the produced message valid, but the probability at which it is produced is the same as its expected occurence. In this way, the honey encryption scheme protects against low entropy passwords as well as low entropy message spaces.

Honey encryption comes from a class of schemes involving deception and decoys with the goal of luring adversaries. There have been a number of "honey" schemes proposed over the past 20 years. Honeytokens [7] are decoy objects interspersed in a system that if used, signal a compromise. An example of a honeytoken scheme is in honeywords [5]. Honeywords are decoy passwords stored in a system's password database. If a log-in attempt using a honeyword is detected, the system infers that the password database has been compromised. Honeypots [6] are full decoy computer systems/servers with the sole purpose of being attacked. Once attacked, the use of the honeypot is to store information about the attack which can be studied and prepared for in the future.

Honey encryption also has close to ties Format-Preserving Encryption (FPE) [1] and Format-Transforming Encryption (FTE) [2]. Both of these encryption schemes have specific restrictions for the message and ciphertext spaces. In FPE, the ciphertext space is the same as the message space. In FTE, the ciphertext space is specified and different from the message space. These encryption schemes give similar security results to honey encryption when used with uniform message spaces. However, honey encryption offers stronger security for non-uniform message spaces since it is not bound to mappings between two message spaces but uses a mapping between a message space and a much larger seed space.

2.1 Honey Encryption Scheme Set-up

We now describe the original honey encryption scheme proposed by Juels and Ristenpart. In this construction, we have a message space M which contains all possible messages. We map these messages to a seed space S through the use of a distribution-transforming encoder (DTE). The seed space is simply the space of all n-bit binary strings for some predetermined n. Each message in $m \in M$ is mapped to a seed range in S. The size of the seed range of m is directly proportional to how probable m is in the message space M. We require some knowledge about the message space M in order for the DTE to map messages to seed ranges, specifically the DTE requires the cumulative distribution function (CDF) of M and some information on the ordering of messages. Additionally, the seed space must be large enough so that even the message with smallest probability in the message space is assigned at least one seed. With this information, we can find the cumulative probability range corresponding the message m and map it to the same percentile seed range in S. We illustrate the encryption process below with a basic example.

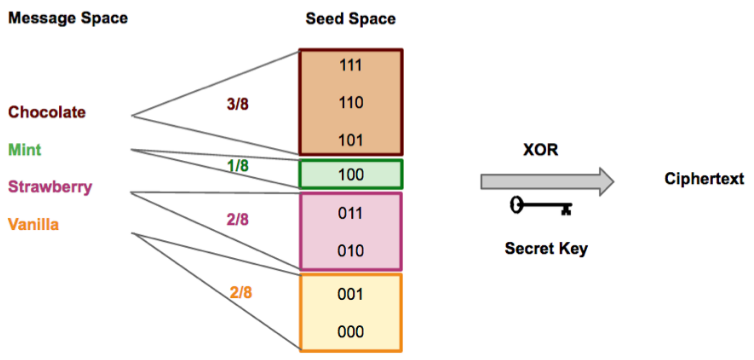

Figure 1: ...

Consider the simple example of encoding ice cream flavors in figure 1. Our message space M consists of different flavors, `M = {chocolate, mint, strawberry, vanilla}` . Through knowledge of some population's preference of ice cream flavors, probabilities are assigned to each flavor. Consider a seed space S of 3-bit strings. With these probabilities, we can then map each flavor to a seed range. In this case, the flavors are ordered alphabetically. Note that this is an arbitrary ordering and a different ordering would lead to a different mapping of seeds.

Now consider the process of encrypting a message, say `chocolate`. Using the DTE, we randomly select a seed in the corresponding seed range. This seed is XORed with a shared secret key to produce the ciphertext.

Decryption is slightly harder. The ciphertext is XORed with the secret key is return the seed. We know that the seed falls into a seed range that corresponds to a message's CDF value. Here we run into a problem. For most message spaces, the CDF is one-way and we cannot go back to a message given a value in it's CDF range. Instead, we use an inverse sampling table. Using a precalculated table of sampled CDF values to messages from the message space, we can run binary search on the inverse sampling table and then linear scan the message space from there.

3 Implementation Approach

In this section, we outline the structure and design decisions made in implementing the honey encryption scheme described above. The main observation to make here is that the work the distribution-transforming encoder (DTE) does is independent of the message space other than the need for a few functions describing the space. With this observation, we are able to code a DTE that performs encoding and decoding for a message space given a set of functions describing the message space. Applications for message spaces can be developed independently and plugged into the DTE program with the proper API. The language used for our implementation was Python.

3.1 Message Space Implementation API

To support the DTE implementation, we chose to require the following functions as input to describe the message space. The functions are wrapped inside of a class `MessageSpaceProbabilityFxns`. Subclasses implementing the required functions can be made for each specific message space.

- CUMULATIVE_DISTR(message) - takes in a message and outputs cumulative probability representing where in ordered message space the message lies
- PROBABILITY_DISTR(message) - takes in a message and outputs probability of that single message
- NEXT_MESSAGE(message) - takes in a message and outputs the next message in ordered message space
- GET_INVERSE_CUMUL_DISTR_SAMPLES() - returns list of pre-calculated sampling of cumulative distribution values to messages

In choosing the above functions, we made a couple of design decisions. We observe that the probability distribution function (PDF) can be created by finding the difference between the current message's cumulative distribution function (CDF) and the next message's CDF. However, we chose to include the PDF since it is called often in the DTE. Because of this, it is important that it is efficiently calculated. For many message spaces, there exists a simple way to calculate the PDF instead of calculating the difference between CDFs.

Both the next message function and inverse sampling table function are used in decryption for binary search of samples and then linear scan of message space. There exists some room for optimization when generating the inverse sampling table. Say we have a message space of n messages and choose to take m samples for the table. The worst case running time of our decryption algorithm would then be $O(\lg m + \frac{n}{m})$ where the first term comes from the binary search and the second from linear scan. To optimize the running time, we set the two terms equal to each other. After some algebra yielding:

$$m \lg m = n$$

This is a transcendental equation and has no algebraic solution. We can use numerical methods such Newton's method to solve for an optimal inverse table size with regards to running time. However, we realize that for many large message spaces, there also exists a space trade-off. Therefore, we made the design decision to allow the user to input the inverse sampling table to optimize for whatever use case they may prefer.

3.2 Distribution Transforming Encoder (DTE)

The DTE maps a message space to a seed space given the above message space functions and a seed space size (i.e 128, 256). We encode by using the CDF and PDF to find a seed range for the message and then randomly selecting a seed inside the range. The following pseudocode displays this process:

```
def encode(m):
    # get range of seed space to pick random string from
    start = cumul_distr(m) * SEED_SPACE_SIZE
    end = int(start + pfxns.prob_distr(m)*SEED_SPACE_SIZE) - 1
    start = int(start)

    # pick random string from corresponding seed space
    seed = random.random(range(start, end))
    return seed
```

When decoding, we find the proportion of the seed within the seed space and binary search for that value in the inverse sample table. This gives us a lower starting point upon which we linear scan through the message space until we find the correct message to return:

```
def decode(s, inverse_table):
    seed_prop = float(s)/SEED_SPACE_SIZE
    (prev_value, prev_msg) = binary_search(inverse_table, seed_prop)
    next_msg = next_message(prev_msg)
    next_value = cumul_distr(next_msg)
    # begin linear scan to find which range seed s falls in
    while seed_loc >= next_value:
        # update prev and next
        (prev_value, prev_msg) = (next_value, next_msg)
        next_msg = next_message(prev_msg)
        next_value = cumul_distr(next_msg)
    return prev_msg
```

4 Message Space Construction

In this section, we discuss applications of honey encryption to various message spaces and the development of the message space API for each application.

4.1 Generic Alphabet

The first message space we consider is the space of n length words of a generic alphabet. Given an alphabet of valid letters Σ, a letter probability function φ mapping letters to probabilities, and a message/word length n, we can construct a probability distribution function (PDF) as well as the cumulative distribution function (CDF) for the message space.

M =
x1
x2
...
xn-1
xn

Figure 2: Message M of generic alphabet space consisting of letters x_1 through x_n where each letter $x_i \in \Sigma$

Although implementing a probability distribution could easily be done through storage of large hash tables or lists, our approach instead takes advantage of the fact that each letter is independent of the rest and generates message probabilities more dynamically.

4.1.1 Generic Alphabet Implementation

In order to implement the generic alphabet message space, we define the functions described in the probability function API as seen in section 3.1 - the PDF, CDF, next message, and inverse sample table functions.

PDF

Because letters are independent of each other, the probability of a given message M consisting of letters $x_0, x_1, ..., x_n$ is given as the product of the individual probabilities of each letter.

$$Pr[M] = \prod_{i=1}^{n} \varphi(x_i)$$

CDF
The cumulative probability of a given message is calculated by iterating through the message starting at the least significant letter. Each letter has an individual CDF which splits the previous letter's PDF. Let us define `letter-cumul(l)` to be the CDF of letter l. Similarly, we define `letter-prob(l)` to be the PDF of letter l. For message M of length n, we calculate $CDF(M)$ as follows:

```
def cumul(m):
    # sum of each index cumulative contribution
    value = 0
    for i in range(1, msg_len)[::-1]:
        value += letter_cumul[m[i]]
        value *= letter_prob[m[i-1]]
    return value
```

Next message
The next message after a given message in the CDF is found by simply incrementing the order of the last letter of the given message. This is because we sort the messages in alphabetical order as given by the order of Σ.

4.2 Application - Credit Cards

As discussed in section 1, honey encryption works well in applications where the message space is highly structured. We explored the domain of credit card numbers and developed a working implementation of a honey encryption scheme on the space of credit cards.

4.2.1 Motivation

Measures to protect personal credit card numbers are motivated by the fact that card numbers are highly sensitive information. For the purposes of simplicity of terminology, we refer to all payment cards in this paper as credit cards, though in the real world, card numbers can represent a variety of card types. The vast majority of people own one or more debit or credit cards; these numbers hold a direct gateway to an individual's personal finances. Therefore, measures taken to protect the security of such card numbers are of high interest to both the owner of the card and to many other involved parties such as card issuers and banks.

Furthermore, credit card numbers consist solely of numerics 0-9. Thus, traditional encryption methods are susceptible to brute-force attacks because a decryption attempt with a guessed invalid key will likely yield immediate feedback about the correctness of the guess if the returned message contains characters other than numbers.

4.2.2 Credit Card Structure

A credit card number has the following properties. These are illustrated in Figure 3.

- 12-19 digits in length
- Consists only of numbers chosen from 0-9
- First digit is Major Industry Identifier Digit
- First 6 digits are the Issuer Identification Number (IIN)
- Rest of digits save for last digit are individual account identifier numbers
- Last digit is checksum digit calculated according to the Luhn algorithm

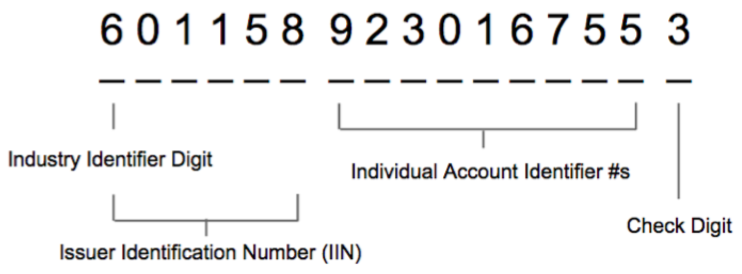
6 0 1 1 5 8 9 2 3 0 1 6 7 5 5 3
Industry Identifier Digit
Individual Account Identifier #s
Check Digit
Issuer Identification Number (IIN)

Figure 3: Example credit card number with card structure

In our implementation of a credit card message space, we focus on utilizing the Issuer Identification Number (IIN) and the card number length as sources of non-uniformity to map card numbers to probabilities. This is because these first six digits vary based on card issuer, whereas the individual account identifier numbers are randomly generated and distributed, and the remaining checksum digit is completely deterministic. We will refer to the IIN as a *prefix*.

The checksum digit is used as a form of verification regarding the validity of a given credit card number. We use it in our implementation to ensure that any credit card numbers used anywhere in our system is at all times valid. It is calculated according to the Luhn algorithm:

Definition. The Luhn algorithm states that the checksum digit is equivalent to nine times the sum all previous digits, taken modulo 10.

A specific card issuer may be assigned either a single prefix, or a range of prefixes. For example, Visa is given the domain of prefixes ranging from 400000 to 499999, which it can then in turn redistribute to smaller card distributers, e.g. MetaBank, which is assigned the prefix 460005. We will denote a range of prefixes using the asterisk * notation, e.g. 4*****, indicating that those digits can be any digit from 0-9.

4.2.3 Message Space Construction

To construct our message space, we first gathered data on valid credit card numbers and scraped 2039 prefixes from the web [12]. Then, we compared these prefixes against tables of credit card lengths based on major issuer (e.g. Visa, Mastercard) and assigned each prefix a card length ranging from 12 to 19 [13]. The most common card length is 16 digits [13].

As described in section 4.2.2, larger card issuers are often assigned a range of prefixes. If more specific prefixes exist within such ranges, we utilize the specific prefix instead such that our message space is as granular as possible and represents the widest range of individual card issuers.

The data was stored as a Python dictionary with entries in the following form:

prefix: [degree of range, card length, weight]

The degree of range indicates the number of asterisks in a prefix, i.e. the number of unique prefixes defined by the designated range. For instance, a prefix of 6428** is given a degree of range of 2, as there are 10^2 prefixes in its range from 642800 to 642899. The card length is the length of the credit card number designated by the card provider, and is thus entirely dependent on the prefix. Finally, the weight is the number of representations that the prefix holds - a prefix of 6428** has a weight of 100, whereas a prefix of 642854 has a weight of 1.

4.2.4 Credit Card Probability Functions API

Having researched the properties of the credit card message space, we then implemented the probability functions required by the DTE to map messages to seeds in the seed space.

PDF
In this implementation, we assign each unique 6-digit prefix uniform probability. Thus, the probability of a given message is dependent only on the length of the postfix. If the postfix has d digits (including a final check digit), then there are $d-1$ random digits and the message has probability $10^{-(d-1)}$, scaled by the

sum of all prefix weights such that the total sum of all message probabilities is 1.

CDF
Using an arbitrary ordering of the scraped prefixes, we first precalculate the cumulative probailities for each prefix. Then, to calculate the cumulative probability of a given message, we first extract its prefix and look up the cumulative range for this prefix. Within each prefix range, we order all messages numerically, and since each message within this prefix range is of the same length and thus equally likely, we can easily calculate the offset probability from the prefix cumulative distribution.

Next message
As mentioned previously, within each prefix range we order all messages numerically. The next function, then, is easy; we simply strip the check digit from our message, increment by one, and then append the recalculated check digit. Note that here we make the simplifying assumption that we will never call next message on the last message in a prefix range, which we can enforce by creating a sufficiently large inverse sampling table.

4.2.5 Credit Card Results

Using the described functions, we implemented a symmetric key honey encryption scheme for credit cards. Using a secret key, our implementation is able to both encode valid credit cards into ciphertexts and decode ciphertexts back to the correct plaintext messages. Furthermore, using an incorrectly guessed key, our implementation produces a incorrect plaintext message which is a valid message in our credit card space.

Note that we make an important assumption in this implementation: that the frequency of specific prefixes is dependent on the range of prefixes that card issuer is assigned. In reality, we do not have knowledge of how accurate this assumption is with the real-world distribution. Credit card information is highly sensitive, and thus little-to-no public information is available about distributional trends in card numbers.

4.3 Application - Simple Messaging

Generating a message space that covers the entire spectrum of the English language is infeasible, mainly because of two factors - size and complexity. Even when sentences or phrases are limited to a 140 character tweet, there exists roughly $2^{154} = 2 * 10^{46}$ different meaningful messages [8]. At this magnitude, honey encryption becomes impractical due to the sheer number of possible messages, and implementing this algorithm would take an incredibly long time. Furthermore, the English language requires a lot of dependencies and must follow many grammatical rules. Generating a complete set of contextually and grammatically correct messages is very difficult for the scope of our project.

Therefore, we turn to a subset of the messaging space that follows a simple structure, namely questions and answers.

4.3.1 Defining a Question and Answer

Questions are an ideal subset of messaging to implement honey encryption on because people tend to ask them using very similar structures. We focus on simple yes-no questions: questions that don't require answers other than a "yes" or a "no". This simplifies the message space of the answer to just those two responses. Most simple questions begin with a verb, a subject, a predicate, and optionally, a direct object. A simple question may be: "Is he eating the potato?" Listed below are several question structures that we use [9].

1. [is/was] + [he/she] + [verb (prog. tense)] + [d. object]
2. [are/were] + [you/we/they] + [verb (prog. tense)] + [d. object]
3. [does/did/will/would] + [he/she] + [verb (present tense)]+ [d. object]
4. [do/did/will/would] + [you/we/they] + [verb (present tense)]+ [d. object]
5. [has/have/had] + [he/she/you/we/they] + [verb (past tense)] + [d. object]

We may convert these simple questions into interrogative questions that require a more detailed response by inserting one of the five W's as the first word of every sentence. An example would be: "Why is he eating the potato?"

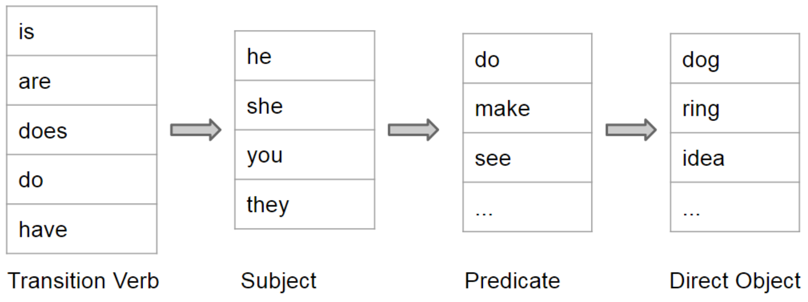

Figure 4: Basic structure of a question

4.3.2 Generation of Message Space

With the structures of the questions laid out, we proceed to fill up the message space by automatically parsing through and generating our questions. The Natural Language Toolkit (nltk) provides a module that generates all possible sentences given a context-free grammar. We use a list of the 300 most common nouns [10] and verbs [11] to generate the message space. The python packages Nodebox Linguistics and Inflect both automatically convert verbs and nouns to their correct form.

While it is possible to implement honey encryptions in previous schemes without using data tables due to the rigid structure of numbers and letters, it is very difficult to implement English messages without data tables. The structure of language is just too complex. Therefore, we generate every single message with its probability distribution in a csv file. Although this trivializes the process of defining the four functions in the probability function API, it is limited by the amount of time and space needed to generate such a file.

4.3.3 Creation of Probability Distribution Function

The list of nouns and verbs that we use to generate the messages also contains the relative frequency of each word, generally ranging from 0 to 1,000. We create a weight function based on those two frequencies to determine the probability of a sentence. First, we normalize both sets of data such that each list has a frequency range from exactly 0 to 1,000. Second, because nouns tend to be more numerous and more specific than verbs, common verbs tend to be used more frequently than common nouns. We multiply the frequency of the verb by 3, giving it more weight. Finally, we take the sum of the value of the verb and noun used in each sentence to give the sentence a final score.

4.3.4 Messaging Limitations

We are able to generate over a hundred million different sentences using context-free grammar and our list of nouns and verbs. Although these questions, with the help of Nodebox Linguistics and Inflect, are all grammatically correct, a large portion of these questions may not make sense in the context of our language. Because our probability distribution function is independent to the noun-verb pair that we use, we do not account for lowering the probability of unusual sentences. For example, although the sentence ”did you eat the potato?” is common, the sentence ”did you eat the professor?” is asked a lot more rarely, even though the word ”professor” is used more often than the word ”potato”. Mapping common verbs towards their associative nouns would improve the validity of our message space. However, this is a complicated problem that is outside the scope of this project.

5 Extension to Public-key Encryption Scheme

Thus far in our discussion of honey encryption, we have used a symmetric encyption scheme to produce a ciphertext C from a given seed S. As always with symmetric key schemes, this assumes that both parties have the architecture to store secret keys securely and exchange keys without risk of interception.

To this end, we introduce an extension to the current honey encryption construction which utilizes public key encryption instead of this symmetric encryption. We will first describe how PKE can be applied to the HE process, and we will then show that the security bounds proved by Juels and Ristenpart [4] as the basis of the HE scheme still hold.

Just as before, we let `DTE = (encode, decode)` be a DTE scheme which outputs seeds in the space $S = \{0,1\}^s$, and let `PKE = (enc, dec)` be a conventional public key encryption scheme with message space S and some ciphertext space C. For our purposes, we can assume a canonical PKE such as RSA.

We can now define `HE[DTE,PKE] = (HEnc, HDec)` as a DTE-then-Encrypt scheme, which first applies the DTE encoding and then encrypts the seed under PKE. To decrypt, simply reverse this order, first decrypting under PKE and then decoding under the DTE. Just as with symmetric encyption, it is easy to see that this scheme is semantically secure if the PKE used is semantically secure.

We claim that this PKE-based honey encryption scheme shares exactly the same security bounds as shown by Juels and Ristenpart for symmetric based honey encryption [4]. To show why, note that the only assumption Juels and Ristenpart make about the `SE` scheme is that encrypting uniform messages gives uniform ciphertexts. More precisely, we assume that $s \leftarrow S; c \leftarrow \texttt{enc}(k, s)$ and $c \leftarrow C; s \leftarrow \texttt{dec}(k, c)$ yield identical distributions for all $k \in K$. In addition to holding for block ciphers in CTR and CBC mode, as described by Juels and Ristenpart, this assumption also holds for many PKE schemes, and so we can use any of these and arrive at identical security bounds as honey encryption schemes with symmetric keys.

This extension to standard honey encryption schemes allows clients to much more easily pass and store encrypted information, without the need for a secure channel of key communication or key storage mechanisms, at the cost of slower and more complex implementations, a tradeoff that could prove useful in a variety of applications.

6 Application Results

We were able to build working implementations of the generic alphabet, credit card, and messaging applications, in accordance to the methodology discussed throughout this paper, as well as demos of the first two.

Our implementation of these message spaces resides at `https://github.com/danielzuot/honeyencryption`.

7 Future Work

7.1 Credit Cards

The credit card message space could be further refined by including information about the last four digits of cards, which is considered semi-public information. For example, many online applications can reveal these digits as a form of identity verification. Utilizing these digits allows us to more accurately map specific credit card numbers to probabilities. However, obtaining this information poses a significant challenge, as large public databases of such information likely do not exist due to the sensitive nature of credit card numbers as a whole.

7.2 Messaging

Although it is very easy to generate a large number of grammatically correct sentences, our message space was limited by the number of contextually accurate sentences that we could generate. In order for this messaging system to work at its highest level, more work needs to be done in creating a probability distribution function between all possible verb/noun pairs. A first step would be to give classifications to nouns that describe them, such as "food", "abstract", "concrete", etc. and assign each verb a probability distribution function when it is referring to different classes of nouns. However, this would require an immense amount of manual work in order to tag the thousands of nouns and verbs commonly used in the English language. Once this is established, we can explore more complex question and statement structures and expand our message space.

8 Conclusion

The recent development of honey encryption offers many password based security schemes resilience to brute force offline attacks by yielding plausible plaintexts under decryption by invalid keys. In this paper, we have presented our implementation of a honey encryption scheme and its application to a variety of use cases, ranging from generic alphabets to credit card numbers to text messaging. Specifically, we addressed the key challenge of generating plausible honey messages for each of these spaces by researching the probabilistic distribution of the message spaces and constructing good DTEs for each. We also extended the base honey encryption scheme to support public key encryption, and showed that this extension preserves honey encryption's security features.

References

[1] Bellare, Mihir, Thomas Ristenpart, Phillip Rogaway, and Till Stegers. "Format-preserving encryption." In Selected Areas in Cryptography, pp. 295-312. Springer Berlin Heidelberg, 2009.

[2] Dyer, Kevin P., Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. "Protocol misidentification made easy with format-transforming encryption." InProceedings of the 2013 ACM SIGSAC conference on Computer communications security, pp. 61-72. ACM, 2013.

[3] Juels, A.; Ristenpart, T., "Honey Encryption: Encryption beyond the Brute-Force Barrier," Security Privacy, IEEE , vol.12, no.4, pp.59,62, July-Aug. 2014

[4] Juels, Ari, and Thomas Ristenpart. "Honey encryption: Security beyond the brute-force bound." Advances in Cryptology–EUROCRYPT 2014. Springer Berlin Heidelberg, 2014. 293-310.

[5] Juels, A., Rivest, R.: Honeywords: Making password-cracking detectable. In: ACM Conference on Computer and Communications Security, CCS 2013, pp. 145–160. ACM (2013)

[6] Spitzner, Lance. Honeypots: tracking hackers. Vol. 1. Reading: Addison-Wesley, 2003.

[7] Spitzner, Lance. "Honeytokens: The other honeypot." (2003).

[8] Monroe, Randall. "Twitter." Twitter. Xkcd, n.d. Web. 13 May 2015.

[9] Zamora, Antonio. "Interrogative Sentences." Basic English Structure. Scientific Psychic, n.d. Web. 13 May 2015. ¡http://www.scientificpsychic.com/grammar/enggram7.html¿.

[10] "Top 1500 Nouns." TalkEnglish, n.d. Web. 13 May 2015. ¡http://www.talkenglish.com/Vocabulary/Top-1500-Nouns.aspx¿.

[11] Albright, Adam, and Bruce Hayes. "English Verbs." Materials on Automated Phonology Learning. UCLA, n.d. Web. 13 May 2015. ¡http://www.linguistics.ucla.edu/people/hayes/learning/english.xls¿.

[12] "List of Bank Identification Numbers" Web. 13 May 2015. ¡http://www.stevemorse.org/ssn/List_of_Bank_Identification_Numbers.html¿.

[13] ISO, "ISO/IEC 7812-1:2006 Identification cards – Identification of issuers – Part 1: Numbering system", October 2006, ¡http://www.iso.org/iso/iso$_{c}atalogue/catalogue_{t}c/catalogue_{d}etail.htm?csnumber = 39698 >$.