A Natural Language Approach to Automated Cryptanalysis of Two-time Pads

Joshua Mason
Johns Hopkins University
josh@cs.jhu.edu

Kathryn Watkins
Johns Hopkins University
kwatkins@jhu.edu

Jason Eisner
Johns Hopkins University
jason@cs.jhu.edu

Adam Stubblefield
Johns Hopkins University
astubble@cs.jhu.edu

ABSTRACT

While keystream reuse in stream ciphers and one-time pads has been a well known problem for several decades, the risk to real systems has been underappreciated. Previous techniques have relied on being able to accurately guess words and phrases that appear in one of the plaintext messages, making it far easier to claim that "an attacker would never be able to do *that*." In this paper, we show how an adversary can automatically recover messages encrypted under the same keystream if only the *type* of each message is known (e.g. an HTML page in English). Our method, which is related to HMMs, recovers the most probable plaintext of this type by using a statistical language model and a dynamic programming algorithm. It produces up to 99% accuracy on realistic data and can process ciphertexts at 200ms per byte on a $2,000 PC. To further demonstrate the practical effectiveness of the method, we show that our tool can recover documents encrypted by Microsoft Word 2002 [22].

Categories and Subject Descriptors

E.3 [**Data**]: Data Encryption

General Terms

Security

Keywords

Keystream reuse, one-time pad, stream cipher

1 Introduction

Since their discovery by Gilbert Vernam in 1917 [20], stream ciphers have been a popular method of encryption. In a stream cipher, the plaintext, p, is exclusive-ORed (XORed) with a keystream, k, to produce the ciphertext, $p \oplus k = c$. A special case arises when the keystream is truly random: the cipher is known as a one-time pad, proved unbreakable by Shannon [18].

CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.

It is well known that the security of stream ciphers rests on never reusing the keystream k [9]. For if k *is* reused to encrypt two different plaintexts, p and q, then the ciphertexts $p \oplus k$ and $q \oplus k$ can be XORed together to recover $p \oplus q$. The goal of this paper is to complete this attack by recovering p and q from $p \oplus q$. We call this the "two-time pad problem."

In this paper we present an automated method for recovering p and q given only the "type" of each file. More specifically, we assume that p and q are drawn from some known probability distributions. For example, p might be a Word document and q might be a HTML web page. The probability distributions can be built from a large corpus of examples of each type (e.g. by mining the Web for documents or web pages). Given the probability distributions, we then transform the problem of recovering p and q into a "decoding" problem that can be solved using some modified techniques from the natural language processing community. Our results show that the technique is extremely effective on realistic datasets (more than 99% accuracy on some file types) while remaining efficient (200ms per recovered byte).

Our attack on two-time pads has practical consequences. Proofs that keystream reuse leaks information hasn't stopped system designers from reusing keystreams. A small sampling of the systems so affected include Microsoft Office [22], 802.11 WEP [3], WinZip [11], PPTP [17], and Soviet diplomatic, military, and intelligence communications intercepted [2, 21]. We do not expect that this problem will disappear any time soon. Indeed, since NIST has endorsed the CTR mode for AES [7], effectively turning a block cipher into a stream cipher, future systems that might otherwise have used CBC with a constant IV may instead reuse keystreams. The WinZip vulnerability is already of this type.

To demonstrate this practicality more concretely, we show that our tool can be used to recover documents encrypted by Microsoft Word 2002. The vulnerability we focus on was known before this work [22], but could not be exploited effectively.

1.1 Prior Work

Perhaps the most famous attempt to recover plaintexts that have been encrypted with the same keystream is the National Security Agency's VENONA project [21]. The NSA's forerunner, the Army's Signal Intelligence Service, noticed that some encrypted Soviet telegraph traffic appeared to reuse keystream material. The program to reconstruct the messages' plaintext began in 1943 and did not end until 1980. Over 3,000 messages were at least partially recovered. The project was partially declassified in 1995, and many of the decryptions were released to the public [2]. However, the ciphertexts and cryptanalytic methods remain classified.

There is a "classical" method of recovering p and q from $p \oplus q$ when p and q are known to be English text. First guess a word likely to appear in the messages, say `the`. Then, attempt to XOR `the` with each length-3 substring of $p \oplus q$. Wherever the result is something that "looks like" English text, chances are that one of the messages has `the` in that position and the other message has the result of the XOR. By repeating this process many times, the cryptanalyst builds up portions of plaintext. This method was somewhat formalized by Rubin in 1978 [15].

In 1996, Dawson and Nielsen [5] created a program that uses a series of heuristic rules to automatically attempt this style of decryption. They simplified matters by assuming that the plaintexts used only 27 characters of the 256-character ASCII set: the 26 English uppercase letters and the space. Given $p \oplus q$, this assumption allowed them to unambiguously recover non-coinciding spaces in p and q, since in ASCII, an uppercase letter XORed with a space can not be equal to any two uppercase letters XORed together. They further assumed that two characters that XORed to 0 were both equal to space, the most common character. To decode the words between the recovered spaces, they employed lists of common words of various lengths (and a few "tricks"). They chose to test their system by running it on subsets of the same training data from which they had compiled their common-word lists (a preprocessed version of the first 600,000 characters of the English Bible). They continued adding new tricks and rules until they reached the results shown in Figure 1. It is important to note that the rules they added were *specifically designed to get good results on the examples they were using for testing* (hence are not guaranteed to work as well on other examples).

We were able to re-attempt Dawson and Nielsen's experiments on the King James Bible[1] using the new methodology described in this paper *without any special tuning or tricks*. Dawson and Nielsen even included portions of all three test passages they used, so our comparison is almost completely apples-to-apples. Our results are compared with theirs in Figure 1.

2 Our Method

Instead of layering on heuristic after heuristic to recover specific types of plaintext, we instead take a more principled and general approach. Let x be the known XOR of the two ciphertexts. A feasible solution to the two-time pad problem is a string pair (p, q) such that $p \oplus q = x$. We assume that p and q were independently drawn from known probability distributions $\Pr_1$ and $\Pr_2$, respectively. We then seek the most probable of the feasible solutions: the (p, q) that maximizes $\Pr_1(p) \cdot \Pr_2(q)$.

To define $\Pr_1$ and $\Pr_2$ in advance, we adopt a parametric model of distributions over plaintexts—known as a *language model*—and estimate its parameters from known plaintexts in each domain. For example, if p is known to be an English webpage, we use a distribution $\Pr_1$ that has previously been fit against a *corpus* (naturally occurring collection) of English webpages. The parametric form we adopt for $\Pr_1$ and $\Pr_2$ is such that an exact solution to our search problem is tractable.

This kind of approach is widely used in the speech and natural language processing community, where recovering the most probable plaintext p given a speech signal x is actually known as "decoding."[2] We borrow some well-known techniques from that community: smoothed n-gram language models, along with dynamic programming (the "Viterbi decoding" algorithm) to find the highest-probability path through a hidden Markov model [14].

2.1 Smoothed n-gram Language Models

If the plaintext string $p = (p_1, p_2, \ldots p_\ell)$ is known to have length ℓ, we wish $\Pr_1$ to specify a probability distribution over strings of length ℓ. In our experiments, we simply use an n-gram character language model (taking $n = 7$), which means defining

$$\Pr_1(p) = \prod_{i=1}^{\ell} \Pr_1(p_i \mid p_{i \dot{-} n+1}, \ldots p_{i-2}, p_{i-1}) \qquad (1)$$

where $i \dot{-} n$ denotes $\max(i - n, 0)$. In other words, the character p_i is assumed to have been chosen at random, where the random choice may be influenced arbitrarily by the previous $n - 1$ characters (or $i - 1$ characters if $i < n$), but is otherwise independent of previous history. This independence assumption is equivalent to saying that the string p is generated from an $(n-1)$st order Markov process.

Equation (1) is called an n-gram model because the numerical factors in the product are derived, as we will see, from statistics on substrings of length n. One obtains these statistics from a training corpus of relevant texts. Obviously, in practice (and in our experiments) one must select this corpus without knowing the plaintexts p and q.[3] However, one may have side information about the *type* of plaintext ("genre"). One can create a separate model for each type of plaintext that one wishes to recover (e.g. English corporate email, Russian military orders, Klingon poetry in Microsoft Word format). For example, our HTML language model was derived from a training corpus that we built by searching Google on common English words and crawling the search results.

[1] We used the Project Gutenberg edition which matches the excerpts from [5], available at `www.gutenberg.org/dirs/etext90/kjv10.txt`

[2] That problem also requires knowing the distribution $\Pr(x \mid p)$, which characterizes how text strings tend to be rendered as speech. Fortunately, in our situation, the comparable probability $\Pr(x \mid p, q)$ is simply 1, since the observed x is a deterministic function (namely XOR) of p, q. Our method could easily be generalized for imperfect (noisy) eavesdropping by modeling this probability differently.

[3] It would be quite possible in future work, however, to choose or build language models based on information about p and q that our methods themselves extract from x. A simple approach would try several choices of $(\Pr_1, \Pr_2)$ and use the pair that maximizes the probability of observing x. More sophisticated and rigorous approaches based on [1, 6] would use the imperfect decodings of p and q to *reestimate* the parameters of their respective language models, starting with a generic language model and optionally iterating until convergence. Informally, the insight here is that the initial decodings of p and q, particularly in portions of high confidence, carry useful information about (1) the genres of p and q (e.g., English email), (2) the particular topics covered in p and q (e.g., oil futures), and (3) the particular n-grams that tend to recur in p and q specifically. For example, for (2), one could use a search-engine query to retrieve a small corpus of documents that appear similar to the first-pass decodings of p and q, and use them to help build "story-specific" language models $\Pr_1$ and $\Pr_2$ [10] that better predict the n-grams of documents on these topics and hence can retrieve more accurate versions of p and q on a second pass.

(a)	Correct pair recovered		Incorrect pair recovered		Not decrypted	
	[5]	This work	[5]	This work	[5]	This work
$P_0 \oplus P_1$	62.7%	100.0%	17.8%	0%	20.5%	0%
$P_1 \oplus P_2$	61.5%	99.99%	17.6%	0.01%	20.9%	0%
$P_2 \oplus P_0$	62.6%	99.96%	17.9%	0.04%	19.5%	0%

(b)	Correct when keystream used three times		Incorrect when keystream used three times		Not decrypted	
	[5]	This work	[5]	This work	[5]	This work
P_0	75.2%	100.0%	12.3%	0%	12.5%	0%
P_1	76.3%	100.0%	11.4%	0%	12.3%	0%
P_2	75.4%	100.0%	11.8%	0%	12.8%	0%

Figure 1: These tables show a comparison between previous work [5] and this work. All results presented for previous work are directly from [5]. Both systems were trained on the exact same dataset (the first 600,000 characters of the King James Version of the Bible, specially formatted as in [5] — all punctuation other than spaces were removed and all letters converted to upper case) and were tested on the same three plaintexts (those used in [5], which were included in the training set). Unlike the prior work, our system was tuned *automatically* on the training set, and not tuned at all for the test set. (a) The first table shows the results of recovering the plaintexts from the listed xor combinations. The reported percentages show the recovery status for the pair of characters in each plaintext position, *without necessarily being in the correct plaintext*. For example, the recovered P_0 could contain parts of P_0 and parts of P_1. (b) The second table shows the results when the same keystream is used to encrypt all three files, and $P_0 \oplus P_1$ and $P_1 \oplus P_2$ are fed as inputs to the recovery program *simultaneously*. Here the percentages show whether a character was correctly recovered in the *correct* file.

It is tempting to define $\Pr_1(\texttt{s} \mid \texttt{h}, \texttt{o}, \texttt{b}, \texttt{n}, \texttt{o}, \texttt{b})$ as the fraction of occurrences of `hobnob` in the $\Pr_1$ training corpus that were followed by `s`: namely $c(\texttt{hobnobs})/c(\texttt{hobnob?})$, where $c(\ldots)$ denotes count in the training corpus and `?` is a wildcard. Unfortunately, even for a large training corpus, such a fraction is often zero (an underestimate!) or undefined. Even positive fractions are unreliable if the denominator is small. One should use standard "smoothing" techniques from natural language processing to obtain more robust estimates from corpora of finite size.

Specifically, we chose parametric Witten-Bell backoff smoothing, which is about the state of the art for n-gram models [4]. This method estimates the 7-gram probability by interpolating between the naive count ratio above and a recursively smoothed estimate of the 6-gram probability $\Pr_1(\texttt{s} \mid \texttt{o}, \texttt{b}, \texttt{n}, \texttt{o}, \texttt{b})$. The latter, known as a "backed-off" estimate, is less vulnerable to low counts because shorter contexts such as `obnob` pick up more (albeit less relevant) instances. The interpolation coefficient favors the backed-off estimate if observed 7-grams of the form `hobnob?` have a low count on average, indicating that the longer context `hobnob` is insufficiently observed.

Notice that the first factor in equation (1) is simply $\Pr(p_1)$, which considers no contextual information at all. This is appropriate if p is an arbitrary packet that might come from the middle of a message. If we know that p starts at the beginning of a message, we prepend a special character BOM to it, so that $p_1 = \text{BOM}$. Since $p_2, \ldots p_n$ are all conditioned on p_1 (among other things), their choice will reflect this beginning-of-message context. Similarly, if we know that p ends at the end of a message, we append a special character EOM, which will help us correctly reconstruct the final characters of an unknown plaintext p. Of course, for these steps to be useful, the messages in the training corpus must also contain BOM and EOM characters. Our experiments only used the BOM character.

2.2 Finite-State Language Models

Having estimated our probabilities, we can regard the 7-gram language model $\Pr_1$ defined by equation (1) as a very large edge-labeled directed graph, G_1, which is illustrated in Figure 2d, Figure 2a. Each vertex or "state" of G_1 represents a context—not necessarily observed in training data—such as the 6-gram `not␣se`.

Sampling a string of length ℓ from $\Pr_1$ corresponds to a random walk on G_1. When the random walk reaches some state, such as `hobnob`, it next randomly follows an outgoing edge; for instance, it chooses the edge labeled `s` with independent probability $\Pr_1(\texttt{s} \mid \texttt{h}, \texttt{o}, \texttt{b}, \texttt{n}, \texttt{o}, \texttt{b})$. Following this edge generates the character `s` and arrives at a new 6-gram context state `obnobs`. Note that the `h` has been safely forgotten since, by assumption, the choice of the *next* edge depends only on the 6 most recently generated characters. Our random walk is defined to start at the empty, 0-gram context, representing ignorance; it proceeds immediately through 1-gram, 2-gram, ... contexts until it enters the 6-gram contexts and continues to move among those.

The probability of sampling a particular string p by this process, $\Pr_1(p)$, is the probability of the (unique) path labeled with p. (A path's label is defined as the concatenation of its edges' labels, and its probability is defined as the product of its edges' probabilities.)

In effect, we have defined $\Pr_1$ using a probabilistic finite-state automaton.[4] In fact, our attack would work for any language models $\Pr_1, \Pr_2$ defined in this way, not just n-gram language models. In the general finite-state case, different states could remember different amounts of context—or non-local context such as a "region" in a document. For example, n-gram probabilities might be significantly differ-

[4]Except that G_1 does not have final states; we simply stop after generating ℓ characters, where ℓ is given. This is related to our treatment of BOM and EOM.

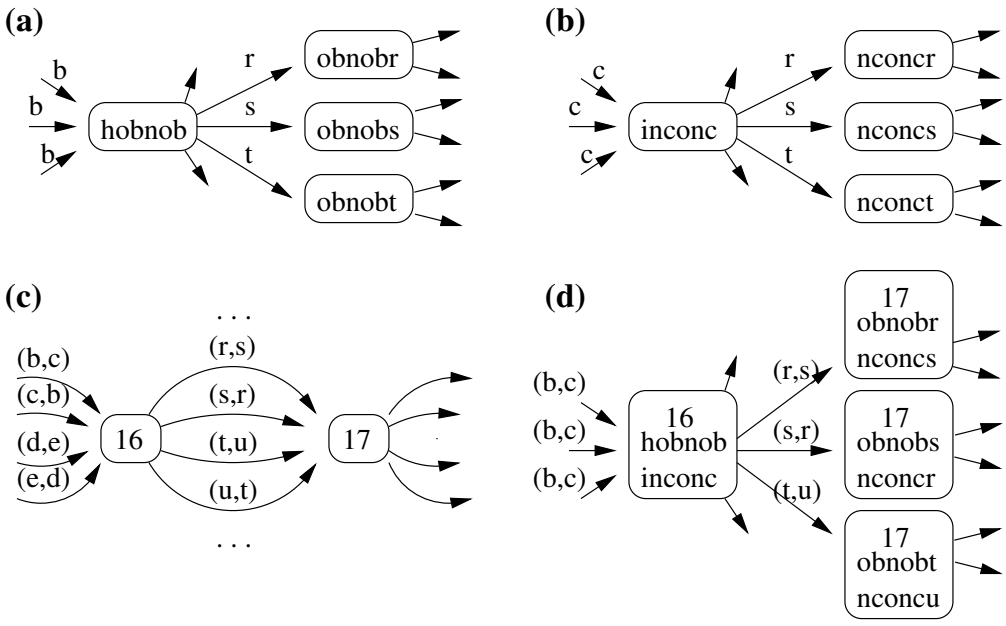

Figure 2: Fragments of the graphs built lazily by our algorithm. (a) shows G_1, which defines $\mathbf{Pr}_1$. *If* we are ever in the state `hobnob` (a matter that is yet to be determined), then the next character is most likely to be `b`, `s`, space, or punctuation—as reflected in arc probabilities not shown—though it could be anything. (b) similarly shows G_2. `inconc` is most likely to be followed by `e`, `i`, `l`, `o`, `r`, or `u`. (c) shows X, a straight-line automaton that encodes the observed stream $x = p\mathbf{xor}q$. The figure shows the unlikely case where $x = (\ldots, 1, 1, 1, 1, 1, 1, 1, \ldots)$: thus all arcs in X are labeled with (p_i, q_i) such that $p_i \oplus q_i = x_i = 1$. All paths have length $|x|$. (d) shows G_x. This produces exactly the same pair sequences of length $|x|$ as X does, but the arc probabilities now reflect the product of the two language models, requiring more and richer states. $(16, \texttt{hobnob}, \texttt{inconc})$ is a reachable state in our example since $\texttt{hobnob} \oplus \texttt{inconc} = 111111$. Of the 256 arcs (p_{17}, q_{17}) leaving this state, the only reasonably probable one is `s`,`r`, since both factors of its probability $\mathbf{Pr}_1(\texttt{s} \mid \texttt{hobnob}) \cdot \mathbf{Pr}_2(\texttt{r} \mid \texttt{inconc})$ are reasonably large. Note, however, that our algorithm might choose a less probable arc (from this state or from a competing state also at time 16) in order to find the *globally* best path of G_x that it seeks.

ent in a message header vs. the rest of the message, or an HTML table vs. the rest of the HTML document. Beyond remembering the previous $n-1$ characters of context, a state can remember whether the previous context includes a `<table>` tag that has not yet been closed with `</table>`. Useful non-local properties of the context can be manually hard-coded into the FSA, or learned automatically from a corpus [1].

2.3 Cross Product of Language Models

We now move closer to our goal by consructing the joint distribution $\Pr(p, q)$. Recall our assumption that p and q are sampled *independently* from the genre-specific probability distributions $\Pr_1$ and $\Pr_2$. It follows that $\Pr(p, q) = \Pr_1(p) \cdot \Pr_2(q)$. Replacing $\Pr_1(p)$ by its definition (1) and $\Pr_2(q)$ by its similar definition, and rearranging the factors, it follows that

$$\Pr(p, q) = \prod_{i=1}^{\ell} \Pr(p_i, q_i \mid \quad p_{i \doteq n+1}, \ldots p_{i-2}, p_{i-1}, \quad q_{i \doteq n+1}, \ldots q_{i-2}, q_{i-1}) \qquad (2)$$

where

$$\Pr(p_i, q_i \mid p_{i \doteq n+1}, \ldots, p_{i-1}, q_{i \doteq n+1}, \ldots, q_{i-1}) = \Pr_1(p_i \mid p_{i \doteq n+1}, \ldots, p_{i-1}) \cdot \Pr_2(q_i \mid q_{i \doteq n+1}, \ldots, q_{i-1}) \qquad (3)$$

We can regard equation (2) as defining an even larger graph, G (similar to Figure 2d), which may be constructed as the cross product of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. That is, $G = (V_1 \times V_2, E)$, where E contains the labeled edge $(u_1, u_2) \xrightarrow{(char_1, char_2) \,:\, prob_1 \cdot prob_2} (v_1, v_2)$ iff E_1 contains $u_1 \xrightarrow{char_1 \,:\, prob_1} v_1$ and E_2 contains $u_2 \xrightarrow{char_2 \,:\, prob_2} v_2$. The weight $prob_1 \cdot prob_2$ of this edge is justified by (3). Again, we never *explicitly* construct this enormous graph, which has more than 256^{14} edges (for our situation of $n = 7$ and a character set of size 256).

This construction of G is similar to the usual construction for intersecting finite-state automata [8], the difference being that we obtain a (weighted) automaton over character *pairs* would still apply even if, as suggested at the end of the previous section, we used finite-state language models other than n-gram models. It is known as the "same-length cross product construction."

2.4 Constructing and Searching the Space of Feasible Solutions

Given x of length ℓ, the feasible solutions (p, q) correspond to the paths through G that are compatible with x. A path $e_1 e_2 \ldots e_\ell$ is compatible with x if for each $1 \le i \le \ell$, the edge e_i is labeled with some (p_i, q_i) such that $p_i \oplus q_i = x_i$. As a special case, if p_i and/or q_i is known to be the special character BOM or EOM, then $p_i \oplus q_i$ is unconstrained (indeed undefined).

We now construct a new weighted graph, G_x, that represents just the feasible paths through G. All these paths have length ℓ, so G_x will be acyclic. We will then find the most probable path in G_x and read off its label (p, q).

The construction is simple. G_x, shown in Figure 2d, contains precisely all edges of the form

$$(i-1,\ \ (p_{i \dot{-} n+1} \ldots, p_{i-1}),\ \ (q_{i \dot{-} n+1} \ldots, q_{i-1})) \xrightarrow{(p_i, q_i)\ :\ prob} (i,\ \ (p_{i \dot{-} n+2} \ldots, p_i),\ \ (q_{i \dot{-} n+2} \ldots, q_i)) \quad (4)$$

such that $p_j \oplus q_j = x_j$ for each $j \in [i \dot{-} n + 1, i]$ and $prob = \Pr_1(p_i \mid p_{i \dot{-} n+1}, \ldots p_{i-2}, p_{i-1}) \cdot \Pr_2(q_i \mid q_{i \dot{-} n+1}, \ldots q_{i-2}, q_{i-1})$.

G_x may also be obtained in finite-state terms as follows. We represent x as a graph X (Figure 2c) with vertices 0, 1, $\ldots \ell$. From vertex $i-1$ to vertex i, we draw 256 edges,[5] labeled with the 256 (p_i, q_i) pairs that are compatible with x_i, namely $(0, 0 \oplus x_i), \ldots (255, 255 \oplus x_i)$. We then compute $G_x = (V_x, E_x)$ by intersecting X with the language-pair model G as one would intersect finite-state automata. This is like the cross-product construction of the previous section, except that here, the edge set E_x contains $(i-1, u) \xrightarrow{(char_1, char_2)\ :\ 1 \cdot prob} (i, v)$ iff the edge set of X contains $(i-1) \xrightarrow{(char_1, char_2)\ :\ 1} i$ and E contains $u \xrightarrow{(char_1, char_2)\ :\ prob} v$.

Using dynamic programming, it is now possible in $O(\ell)$ time to obtain our decoding by finding the best length-ℓ path of G_x from the initial state (0,(),()). Simply run a single-source shortest-path algorithm to find the shortest path to any state of the form $(\ell, \ldots)$, taking the length of each edge to be the negative logarithm of its probability, so that minimizing the sum of lengths is equivalent to maximizing the product of probabilities.[6] It is not even necessary to use the full Dijkstra's algorithm with a priority queue, since G_x is acyclic. Simply iterate over the vertices of G_x in increasing order of i, and compute the shortest path to each vertex $(i, \ldots)$ by considering its incoming arcs from vertices $(i-1, \ldots)$ and the shortest paths to *those* vertices. This is known as the Viterbi algorithm; it is guaranteed to find the optimal path.

The trouble is the size of G_x. On the upside, because x_j constrains the pair (p_j, q_j) in equation (4), there are at most $\ell \cdot 256^6$ states and $\ell \cdot 256^7$ edges in G_x (not $\ell \cdot 256^{12}$ and $\ell \cdot 256^{14}$). Unfortunately, this is still an astronomical number. It can be reduced somewhat if $\Pr_1$ or $\Pr_2$ places hard restrictions on characters or character sequences in p and q, so that some edges have probability 0 and can be omitted. As a simple example, perhaps it is known that each (p_j, q_j) must be a pair of *printable* (or even alphanumeric) characters for which $p_j \oplus q_j = x_j$. However, additional techniques are usually needed.

Our principal technique at present is to prune G_x drastically, sacrificing the optimality guarantee of the Viterbi algorithm. In practice, as soon as we construct the states $(i, \ldots)$ at time i, we determine the shortest path from the initial state to each, just as above. But we then keep only the 100 best of these time-i states according to this metric (less greedy than keeping only the 1 best!), so that we need to construct at most $100 \cdot 256$ states at time $i+1$. These are then evaluated and pruned again, and the decoding proceeds. More sophisticated multi-pass or A* techniques are also possible, although we have not implemented them.[7]

2.5 Multiple Reuse

If a keystream k is used more than twice, the method works even better. Assume we now have three plaintexts to recover, p, q, and r, and are given $p \oplus q$ and $p \oplus r$ (note that $q \oplus r$ adds no further information). A state of G or G_x now includes a triple of language model states, and an edge probability is a product of 3 language model probabilities.

The Viterbi algorithm can be used as before to find the best path through this graph given a pair of outputs (those corresponding to $p \oplus q$ and $p \oplus r$). Of course, this technique can be extended beyond three plaintexts in a similar fashion.

3 Implementation

Our implementation of the probabilistic plaintext recovery can be separated cleanly into two distinct phases. First, language models are built for each of the types of plaintext that will be recovered. This process only needs to occur once per type of plaintext since the resulting model can be reused whenever a new plaintext of that type needs to be recovered. The second phase is the actual plaintext recovery.

All our model building and cracking experiments were run on a commodity Dell server (Dual Xeon 3.0 GHz, 8GB RAM) that cost under $2,000. The server runs a Linux kernel that supports the Xeon's 64-bit extensions to the x86 instruction set. The JVM used is BEA's freely available JRockit since Sun's JVM does not currently support 64-bit memory spaces on x86.

3.1 Building the Language Models

To build the models, we used an open source natural language processing (NLP) package called LingPipe [4].[8] LingPipe is a Java package that provides an API for many common NLP tasks such as clustering, spelling correction, and part-of-speech tagging. We only used it to build a character based n-gram model based on a large corpus of documents (see section 4 for details of the corpora used in our experiments). Internally, LingPipe stores the model as a trie with greater length n-grams nearer the leaves. We had LingPipe "compile" the model down to a simple lookup table based representation of the trie. Each row of the table, which corresponds to a single n-gram, takes 18 bytes except for the rows which correspond to leaf nodes (maximal length n-grams) which take only 10 bytes. If an n-gram is never seen in the training data, it will not appear in the table; instead, the longest substring of the n-gram that does appear in the table will be used. The extra 8 bytes in these nodes specify how to compute the probability in this "backed-off" case. All probabilities in both LingPipe and our Viterbi implentation are computed and stored in log-space to avoid issues with integer underflow. All of the language models used in this paper have $n = 7$. The language models take several

[5] Each edge has weight 1 for purposes of weighted intersection or weighted cross-product. This is directly related to footnote 2.

[6] Using logarithms also prevents underflow.

[7] If we used our metric to prioritize exploration of G_x instead of pruning it, we would obtain A* algorithms (known in the speech and language processing community as "stack decoders"). In the same A* vein, the metric's accuracy can be improved by considering right context as well as left: one can add an estimate of the shortest path from $(i, \ldots)$ through the remaining ciphertext to the final state. Such estimates can be batch-computed quickly by decoding x from end-to-beginning using smaller, m-gram language models ($m < n$).

[8] Available at: `http://www.alias-i.com/lingpipe`

hours to build, and the non-compiled trie representation can use many gigabytes of memory for large corpora. For example, our biggest compiled model is 872 MB and took over 8 hours to generate. It contains the results of looking at more than 7 billion training characters from 300,000 HTML files.

Language modeling has a rich literature, and there are many options that could be useful in particular scenarios, especially for non-natural-language or unknown-genre plaintexts. For example, if we had used less training data, we would have been able to take far more context into account without exceeding the available RAM. LingPipe can build practical 32-gram character language models when the training data is limited to 10 million words [4]. An intermediate strategy would be to include long contexts (e.g., 31-grams) in the language model only when they are very frequent, otherwise backing off to shorter contexts (e.g., 6-grams). There also exist modeling techniques for considering discontiguous or long-distance contexts; for adapting to input properties (cf. Lempel-Ziv); and for training effectively on heterogeneous corpora that consist of several (labeled or unlabeled) genres.

3.2 Recovering the Plaintext

The optimized Viterbi search represents the bulk of our implementation. Ideally, we would first generate the full automaton from the LingPipe language model tables. Unfortunately, this creates a state explosion since each state in one model can be paired with every state in the other model. Instead, we generate states on the fly from the tables as they are needed. This is not quite as expensive as it seems.

Because of the way that our automaton is constructed, each state transition represents adding a single character to each of the n-grams in the current state. The number of single characters that can be added to an n-gram (given x) is usually relatively small. If we assume the underlying plaintext can be modeled using only non-binary characters, there are only 128 possible choices. In our experiments, the actual number of observed 1-grams when modeling HTML, e-mail, or other plain-text protocols hovers around the number of printable characters. This means each combination state only has approximately 96 transitions.

Unfortunately, the number of possible states still grows exponentially in the length of the plaintexts being recovered. To deal with this, we use the "beam search" heuristic optimization: low probability partial paths are pruned early so as to limit the number of states that we must keep in memory. This path pruning means that the search is no longer guaranteed to end up with the best path (it might have been pruned), but the technique seemed to work in practice in our experiments. The pruning is implemented as a binary tree of the current states sorted by the probability of the path; the tree is pruned after each output byte is considered. Processing each byte in our implementation takes approximately 200ms.

After the newly created tree is sorted and pruned, the surviving nodes' state numbers, along with their parents' state numbers, are written to disk. The program writes a file containing this information for each byte in the input file. At the end of the Viterbi search, the program reassembles the final Viterbi crack path by parsing these files. The result is the pair of plaintext messages that represent the best path through the graph.

4 Results

In this section, we present the results of a variety of experiments we performed using our implementation. We examined three different types of files: unstructured English text files (emails, with headers), English text files with text-based structure (HTML documents), and English text files with binary structure (Microsoft Word documents). We examine the effect of such factors as the amount of training data available, the number of times that a keystream was reused, and whether having the plaintexts be of different types affects the reconstruction. We always assumed that both plaintexts started with BOM, and we took ℓ to be the length of the XOR stream, determined by the shorter of the plaintexts. We did not use EOM.

4.1 Data Collection and Testing Methodology

All of the data that we used in our experiments is publicly available. Our HTML dataset was the easiest to gather. We searched Google for common English words and used wget to crawl the results. Our largest HTML training corpus consists of 300,000 different files (over 7 billion characters).

Emails were slightly harder to come by — we didn't find anyone willing to allow us to experiment on their inboxes. Fortunately, the Federal Energy Regulatory Commission (FERC) has made available the emails sent by the senior managers of Enron. These emails were collected during FERC's investigation into Enron's business practices.[9] The emails do not include attachments, and some of the emails have been redacted or removed due to requests from the employees involved. However, it is, to our knowledge, the best email corpus publicly available. Our largest email training corpus consists of 500,000 emails (more than 4 billion characters).

To collect Microsoft Word documents we again turned to Google, this time filtering so that only `.doc` files would be returned. The largest Word training corpus we use is 90,000 files (more than 450 million characters). We only train on the first 5,000 bytes of each file as Word files are typically larger than HTML documents or email messages; this is, however, more than sufficient to get past the Word header information.

In all cases, we randomly reserved some of the files we collected for experimentation or evaluation. All of the results in this section are reported on documents that were not used in the training of the model or design of the method.

4.2 Basic Results

We begin by examining how our reconstruction works on each type of plaintext we consider (i.e. HTML, email, Word). We randomly selected 100 files of each type and XORed pairs of the same type together to create 50 different XORed streams for each type. This corresponds to a likely real-world case: when a system or protocol that exhibits keystream reuse is generally used with a single type of file. We tried to recover the plaintexts from each of these streams under models built using training corpora of varying size. The results are shown in Figure 3. It is first worth noting that increasing the training corpus size has a relatively small effect on the results. It turns out that at the corpus sizes we consider, the most important factor is the "variety" of documents in the corpus. When we were initially experimenting, we failed to randomize our choices of which documents from our full corpus would be included in each training set. This led to

[9]The 400MB zip file is available from `http://www.cs.cmu.edu/~enron/`.

the excellent results on test files that happened to be related to those that were trained on and terrible results on all other files. Randomizing the selection of the training set fixed this problem in the HTML and Word training sets. The email corpus consists of messages from only around 150 users. This provides a low degree of diversity and so the email models seem to more easily become "unbalanced." The 200K email corpus appears to do much better than the other email corpora on many tests (leading to its high median recovery percentage). However, unlike the other email models, it can recover only 50% of characters in some files.

The HTML results are by far the best, with more than 99% of characters correctly being decoded. The Word results are the worst at 44%, likely due to Word files having less predictable structure and more possible byte values than the other two data sets. The email results fall in the middle, with 82% of the characters correctly recovered on average. We examine why the email reconstruction is worse than the HTML reconstruction in the following sections and show some techniques that can be used to improve it.

4.3 The Switching Streams

The email results seem to be far worse than those for HTML. This is only true because of the particular way we decided to measure success. If we instead said that a reconstruction succeeded on a particular byte when the two bytes that are returned are correct, regardless of whether they are in the correct file, the results would change dramatically as shown in Figure 4. This disparity is due to an artifact of the way we recover the plaintexts. Our n-gram models only look at the last few characters that are recovered. Usually this allows for the streams to be reconstructed sequentially — the next character recovered in each plaintext is likely to complete a word already begun in that plaintext. However, assume that the reconstruction fails to recover correct bytes for a short stretch. When the reconstruction gets back on track, it has no idea to which stream the following correct bytes belong. Therefore, it will sometimes choose incorrectly, and correct bytes will be added to the incorrect file until another such switch occurs. We now consider two methods by which this problem can be ameliorated.

4.4 Improving the Results

Our first attempt at improving the results simply assumes that the keystream was used more than twice. Here, the model should prevent switching as each plaintext is now matched with two other plaintexts (i.e. two of the plaintexts can no longer be switched because their XOR differences with the third will no longer be valid). This technique is quite effective as shown in Figure 5.

There are times when an attacker may be unwilling to wait for multiple keystream reuses; perhaps they are very rare events. However, if the two plaintexts are of two different types, as could be the case in encrypted protocols that transport multiple types of documents (e.g. WEP), a similar effect occurs. After a period of errors, the model can recover correctly since the distributions of the two plaintexts are different and thus the probabilities of the switched stream paths will be lower than the probabilities of the true path. The results of recovering HTML documents XORed with email messages is shown in Figure 6.

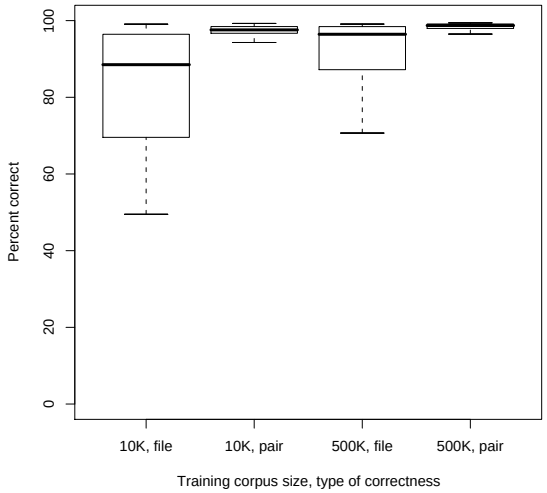

Figure 4: This graph illustrates the major problem that is encountered during the reconstruction. After a short string of errors occurs and the reconstruction recovers, it's no longer clear to which recovered plaintext the correct characters should be added. This creates reconstructions where parts of each original plaintext occur between short errors. Here, we show the difference between computing correctness based on whether a character was assigned the correct *file* vs. simply whether each *pair* of recovered characters was correct. All plots are of 100 randomly chosen email messages xored in pairs.

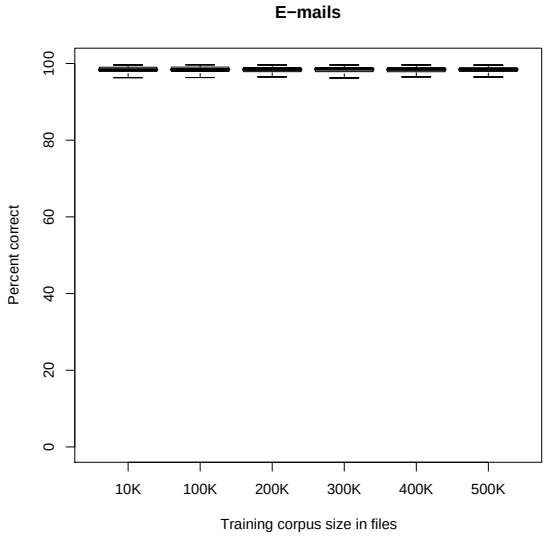

Figure 5: This graph shows the difference between having a keystream reused once, yielding $p \oplus q$, and having it reused twice, giving $p \oplus q$ and $p \oplus r$. Each scenario was evaluated using 50 randomly chosen instances from the email corpus.

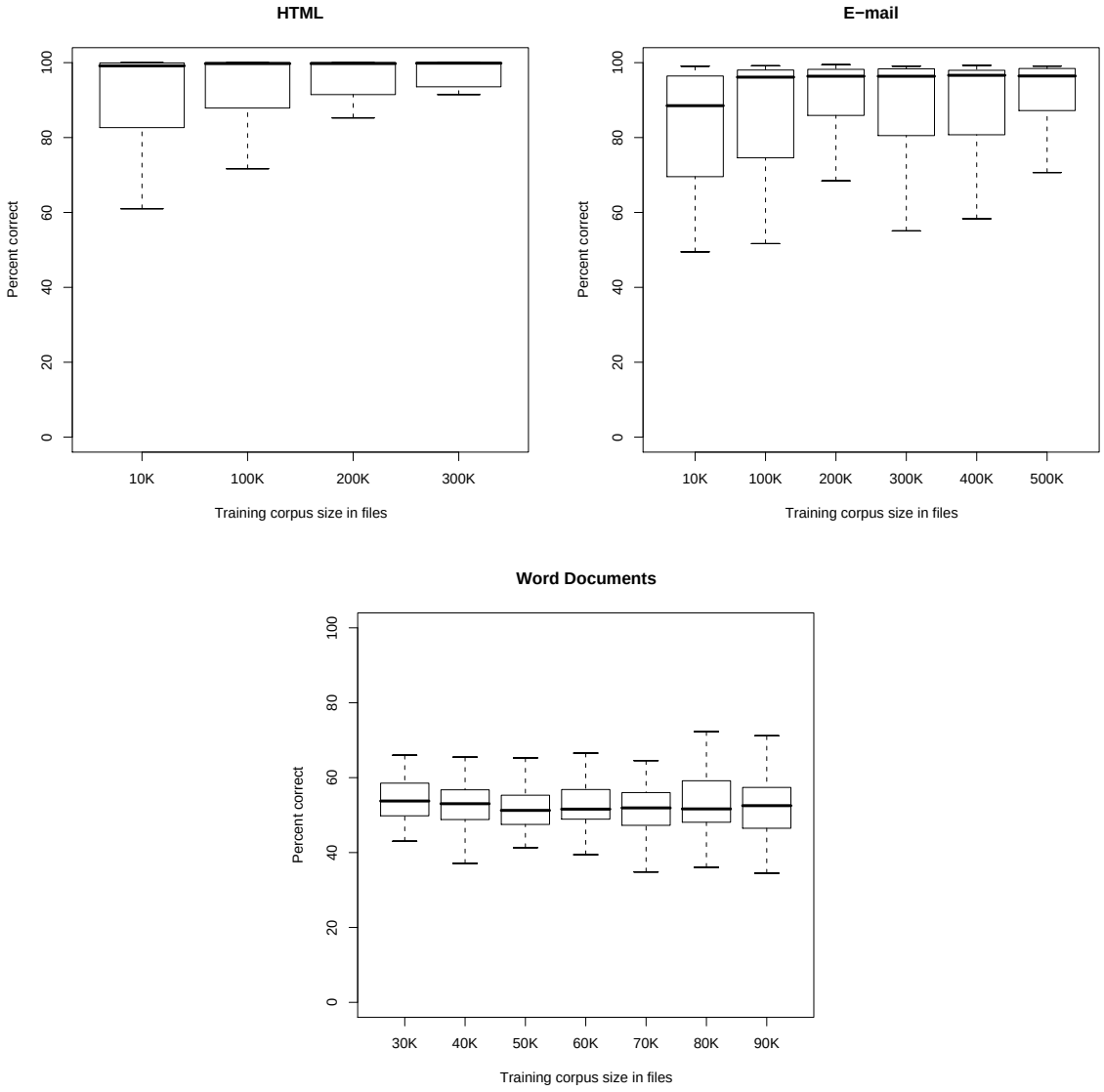

Figure 3: **These graphs show the basic results when cracking two files of the same type xored together at different training corpus sizes. Fifty pairs of files were cracked at each training corpus size. The models should not be directly compared to one another: different types of files have different lengths and so influence the model differently. Plotting based on the number of characters read for each model would also be deceptive as it would not indicate the differing number of start bytes following bom (which we find to be important given our heavy pruning). See section 4.1 for information on the number of bytes in the models.**

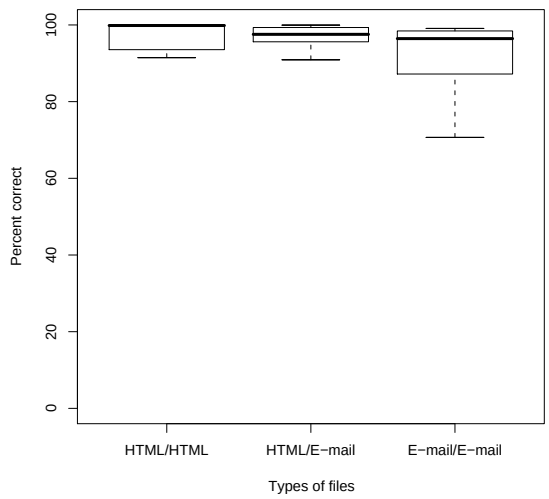

Figure 6: Here, the results show the effect of using reusing a keystream on two different types of documents. Fifty streams were recovered first for two xored HTML files, then for an HTML file xored with an email message, and finally with two email messages xored together.

November 13, 2002#ATA/ATAPI Host Adapters Standard (ATF;#h Packet)#This is no internal working document of T13, a Technical Committee of Accredited Standards Committee INCITS. The T13 Technical Committee may modify the contents. This document is made available and has not been approved. The contents may be modified by the T13 Technical technical committees, and their associated task groups to reproduce this document for the purposes of

November 13, 2002#ATA/ATAPI Host Adapters Standard (ATA # Adapter)#This is an internal working document of T13, a Technical Committee of Accredited Standards Committee INCITS. The T13 Technical Committee may modify the contents. This document is made available for review and comment only.#Permission is granted to members of INCITS, its technical committees, and their associated task groups to reproduce this document for the purposes of

Figure 7: At top, the cracked file resulting from adding a character to an encrypted Word document. Characters that are underlined were recovered incorrectly while characters that are wavy underlined were recovered into the wrong file (see section 4.3). Hash characters (#) represent unprintable ASCII values (i.e. formatting). Underneath, the original corresponding plaintext is shown.

5 Attacking Word 2002

This section shows how our technique can be used to attack a real system: Microsoft Word 2002 RC4 encryption as shown vulnerable in [22]. Microsoft Word 2002 allows users to encrypt their documents with a password. The user can select from one of many cipher suites and simply enter his password, encrypting the document using the chosen cipher. Among the choices for cipher suite is a popular stream cipher, RC4. RC4 [16] takes a key, k, and uses it to generate a keystream, $RC4(k)$, that is then xored with the plaintext.

When Microsoft Word encrypts a document with RC4, the document is assigned a randomly generated initialization vector, IV. The initialization vector is then concatenated with the user's encryption password and cryptographically hashed forming the key array for RC4:

$$k = H(IV||password)$$

The problem arises when the document is edited and saved. The initialization vector is not regenerated after editing. This means the original document and the edited document both use the same keying material. Depending which editing changes are made, an attacker possessing both the original encrypted document, $p \oplus RC4(k)$ and its edited version, $p' \oplus RC4(k)$, can use our method to gain portions of the original plaintexts.

Not all changes to the document allow recovery, however. For example, if only a single character is changed in p, $p \oplus p'$ will be almost completely zeros. While this is interesting and useful information to an attacker, it is not a full recovery. Fortunately, most edits do not affect only a few bytes of the file: inserting a single character near the beginning of the document is sufficient as all of the other characters will be offset.

Aside from adding characters, there are several features that a user can change that may or may not yield portions of the plaintext. For instance, making a character at the top of a Word document bold yields no results. Adding a footnote, though, follows a similar pattern as adding a character. If a footnote is added at the top of a document, a large portion of the original document can be obtained. However, a footnote at the end of a document is the same as appending a character to the end of a document. The track changes feature follows the same pattern, but yields slightly more information should the editor append a character to the end of a document. Deleting and re-adding a paragraph with the exact same formatting, as well as double spacing an entire document, yields no useful results.

In order to test our method, we used Google to search for a Word document with two available revisions. This models a real set of changes that could occur between two saved versions of a document. We were not able to find such a pair that was encrypted, so we used Word 2002 to encrypt the pair ourselves so that the IV was reused. We then applied our tool using the Word corpus build from 90,000 other Word documents. The recovery was 54% accurate (84% pairwise accurate), which agreed with the experiments from section 4.2. A portion of the recovered text is shown in figure 7.

6 Related Work

Markov models (hidden or otherwise) have been previously used for several purposes in security and cryptography such as improving dictionary attacks [13], mounting side channel attacks on protocols [19], recoving keystrokes based on the way they sound [23] and solving simple substitution ciphers [12], among others.

7 Conclusions

We have shown that keystream reuse is a real problem—allows a practical attack—when the data being encrypted comes from a known, non-uniform distribution. Our attack is general and can be easily applied to new types of files as the need arises. We have achieved over 99% accurate recovery in some instances and shown how to improve our results for other types of files under specific conditions such as a

keystream being reused multiple times. Finally, footnotes 3 and 7 outlined opportunities for future improvements in accuracy and speed.

The technique does not directly apply for plaintexts that have a near uniform distribution, such as files that have been compressed. In theory, a language modeling attack could still be used in this case—one simply searches for p, q, such that $p \oplus q = x$ and $\Pr_1(decompress(p)) \cdot \Pr_2(decompress(q))$ is maximized. However, dynamic programming can no longer be used to render this brute-force attack tractable.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant No. 0347822 to the third author. We thank Yoshi Kohno, David Molnar, Kevin Fu, Erika McCallister, Fabian Monrose, and Charles Wright for their helpful comments on this work.

7.1 References

[1] L. E. Baum. An inequality and associated maximization technique in statistical estimation of probabilistic functions of a Markov process. *Inequalities*, 3, 1972.

[2] R. L. Benson and M. Warner. *Venona: Soviet Espionage and the American Response 1939-1957*. Central Intelligence Agency, Washington, D.C., 1996.

[3] N. Borisov, I. Goldberg, and D. Wagner. Intercepting mobile communications: The insecurity of 802.11. In *MOBICOM 2001*, 2001.

[4] B. Carpenter. Scaling high-order character language models to gigabytes. In *Association for Computational Linguistics Workshop on Software*, Ann Arbor, MI, 2005.

[5] E. Dawson and L. Nielsen. Automated cryptanalysis of xor plaintext strings. *Cryptologia*, 20(2):165–181, April 1996.

[6] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statist. Soc. Ser. B*, 39(1):1–38, 1977. With discussion.

[7] M. Dworkin. Recommendation for block cipher modes of operation. NIST Special Publication 800-38A, 2001.

[8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, 1979.

[9] D. Kahn. *The Codebreakers*. Scribner, New York, NY, 1996.

[10] S. Khudanpur and W. Kim. Contemporaneous text as side information in statistical language modeling. *Computer Speech and Language*, 18(2):143–162, 2004.

[11] T. Kohno. Attacking and repairing the winzip encryption scheme. In *11th ACM Conference on Computer and Communications Security*, pages 72–81, Oct 2004.

[12] D. Lee. Substitution deciphering based on hmms with applications to compressed document processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(12):1661–1666, Dec 2002.

[13] A. Narayanan and V. Shmatikov. Fast dictionary attacks on human-memorable passwords using time-space tradeoff. In *12th ACM Conference on Computer and Communications Security*, pages 364–372, Washington, D.C., Nov 2005.

[14] L. R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[15] F. Rubin. Computer methods for decrypting random stream ciphers. *Cryptologia*, 2(3):215–231, July 1978.

[16] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[17] B. Schneier, Mudge, and D. Wagner. Cryptanalysis of microsoft's pptp authentication extensions (ms-chapv2). In *CQRE '99*, 1999.

[18] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379—423, July 1948.

[19] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *10th USENIX Security Symposium*, Aug 2001.

[20] G. Vernam. Secret signaling system. U.S. Patent 1310719, July 1919.

[21] P. Wright. *Spy Catcher*. Viking, New York, NY, 1987.

[22] H. Wu. The misuse of rc4 in microsoft word and excel. Cryptology ePrint Archive, Report 2005/007, 2005. `http://eprint.iacr.org/`.

[23] L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *12th ACM Conference on Computer and Communications Security*, pages 373–382, Washington, D.C., Nov 2005.