Mimic Functions and Tractability

Peter Wayner
Department of Computer Science
Cornell University
Ithaca, New York 14850

Abstract

Mimic functions are designed to hide information by transforming it into another format. They may be as simple as converting the data into a file with a particular statistical profile or as complex as giving the new file any computable structure. This paper will analyze some of the theoretical results about the security of these functions and give a practical algorithm for producing more secure mimic functions.

Anyone who needs to deliver a message knows that there are two separate ways to ensure security. The first is to encrypt the message so that no one will be able to unscramble it if it is intercepted. The whole field of mathematical cryptography has evolved around this approach. The second method is to hide the message so it can't be found. The history of spies and the collection of spy novels and movies are filled with tales of microdots, hidden compartments and a number of other techniques. There have been many different foreys into hiding the existence of a message by cryptographers like the work by Simmons [Sim84] and many others.

Mimic functions were introduced in [Way90] as a general technique for converting information from one form into another seemingly benign and innocent format. The paper described three different types of functions which could generate results that fall into the standard heirarchy of language complexity: the regular languages, the context-free languages and the recursively-enumerable languages. The first group of functions converted data into strings from a regular set. These sets were described in terms of compression function because if a compression algorithm like Huffman coding converted a file with uneven statistical distributions into a small, more evenly distributed file, then its inverse would convert an evenly distributed file into one that would statistically mimic the file. The internal structure has many similarities to the homophonic ciphers developed by Massey and others. [JKM90]

The second type of function will convert data into strings from an unambiguous context-free language. The bits from the data file are used to choose which production of the grammar to use when producing the string. These bits can be recovered by parsing the string, finding the list of productions which generated the string and converting these productions into the bits. If a non-ambiguous method of parsing the productions was used, then data could easily be hidden in the structure of the parse tree.

Encoding the data in the strings accepted by a Turing machine is the next step in the progression. In the earlier paper, this level of sophistication was described in terms of *Van Wijngaarden* grammars. These grammars are essentially double context-free grammars where a meta-level CFG chooses the productions followed by the lower-level CFG. It is easy to see that these two levels of grammars encode all recursively enumerable sets by noting a well-known theorem that a finite automaton with two stacks is equivalent to a Turing machine. [AHU83] Since this machine has two stacks, it should be at least intuitively easy to see how VW-grammars are equivalent to the set of recursively enumerable languages.

The previous paper ([Way90]) described some of the practical aspects of producing mimic functions. This paper will examine some of the theoretical results that show that breaking CFG mimic functions can be a difficult proposition. Results from computational learning theory can be applied directly to show that "breaking" it could be as difficult as cracking RSA or factoring a Blum integer. The result shows that breaking a mimic function is in a class that contains RSA, but the result itself does not give any hints on how to construct "strong" grammars. The paper will show one technique for constructing grammars and argue that these grammars are secure against one straight-forward attack.

This paper has three major sections. The first discusses a theoretical result from computational learning theory that establishes that breaking a mimic function can be as difficult as many standard cryptographic systems like RSA. The next section will show how to construct grammars that should resist straight-forward attacks. The last section will discuss some of the possibilities of using some of the theoretical results about turing machines to assess the security of more complicated mimic functions.

Context-Free Mimic Functions

A Context-Free Grammar (CFG) is specified by a set of variables, V, a set of terminals, Σ, a set of productions, Π and a start symbol, S. The CFG is said to be *linear* if the right side of each production has no more than one variable. The CFG is *regular* if the right side of every production has either one terminal and one variable or one terminal. Figure 1 shows a few productions from simple context-free grammar designed to mimic the text of an announcer describing a baseball game over the radio.

The exact technique used to encode data in the strings of a grammar can be found in [Way90]. It is similar to the technique found in the appendix for efficiently mapping bits to choices. A section of text drawn from the complete grammar shown in part in Figure 1 can be found in Figure 2 where the string "Paul is Dead" is hidden in the voice-over to a baseball game.

Breaking a mimic function can be thought of as infering or learning a grammar, G, that generated a set of strings, $\{s_1, s_2 \ldots s_n\}$. If the plain text of the message is known, then the attacker might know the bits $\mathcal{B}_i$ which generated the individual strings, s_i. The abstraction is conservative because it would be hard to determine the ends of the individual strings in the example in figure 2. A successful attacker wants use the information in $\{s_1, \ldots, s_n\}$ and $\{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ to be able to determine the set of bits, $\{\mathcal{B}_{n+1}, \ldots, \mathcal{B}_p\}$ that produced a set of strings, $\{s_{n+1}, \ldots, s_p\}$ from the same grammar, *WhapperInfieldHit* $\rightarrow$ He tries to bunt, and *BlogsInfielder* *BuntResponseHit* | He knocks it down the line between the legs of *BlogsInfielder* . | He waps it into the short-stops glove, but he can't control it . Safe at first . | He lifts it over the head of *BlogsInfielder* *HitResult* | The batter gets a piece of it *HitResult* | It's contact time *HitResult* | Nice hit *HitResult* | Whoa ! That swing was on the money *HitResult* | Nice job *HitResult* | Great hit *HitResult* | Super looper for a hit . | He knocks a line-drive into the head of *BlogsInfielder* .

WhapperInfieldOut $\rightarrow$ grounds out to *BlogsInfielder* . | He pops it up to *BlogsInfielder* . | He tries to bunt, and *BlogsInfielder* *BuntResponseOut* | He knocks a line-drive into the glove of *BlogsInfielder* | He knocks an easy bouncer to *BlogsInfielder*. | He bounces one of the ground into the first-baseman's glove .

Figure 1: Several productions from a CFG for producing strings that mimic a baseball announcer covering a game between the Whappers and the Blogs. The text in italics (e.g. *WhapperInfieldOut*) represent variables.

G.

Breaking Grammars as "learning"

The problem of learning a grammar has been studied as part of a large body of knowledge developed by investigating some of the theoretical aspects of machine learning as posed by Valient's paper [Val84]. This view assumes that a computer is given a set of examples of a class and then it must "learn" to predict whether a new example is in the class. Many of the basic algorithms proposed are randomized and the algorithms succeed if and when they predict the membership correctly more than a pre-defined threshold.

Some of the recent work in the area proved many applicable results. Kearns and Valiant [KV89, Kea89] determined that learning boolean formulae, finite automata or constant-depth threshold circuits is at least as difficult as inverting RSA encryption ([RRA78]) or factoring Blum integers (x, such that $x = pq$ and $p, q = 3 mod 4$). Angluin and Karitonov extended this work along with the work of Naor and Yung to show that there aren't polynomial time prediction algorithms with membership queries for classes including finite unions or intersections of Finite Automata and Context-free Grammars. Rivest [Riv91] provides a good summary of how learning theory interacts with cryptography.

These bounds show that there are grammars that can't be approximated by a function that tries to "learn" the pattern– even if the algorithm is allowed to choose the cases and ask whether they are members of the language. The proof is a simple reduction that shows that it is possible to express the pairs of public-key and message in the

It's time for another game between the Whappers and the Blogs in scenic downtown Blovonia. I've just got to say that the Blog fans have come to support their team and rant and rave. Let's get going ! Another new inning. Ain't life great, Ted ? Yup. How about those players. The pitcher spits. Prince Albert von Carmicheal comes to the plate. He's trying the curveball . He pops it up to Harrison "Harry" Hanihan. One out against the Whappers. Now, Parry Posteriority swings the baseball bat to stretch and enters the batter's box. Here we go. OOOh, that's almost in the dirt . Definitely a ball. The next pitch is a bouncing knuckleball. Short and away. The umpire calls a ball . It's a change-up. and it's... La Bomba ! HomeRun ! Yup, got to love this stadium. Now, Mark Cloud swings the baseball bat to stretch and enters the batter's box. Yeah. He's uncorking a toaster. No contact in Mudsville ! It's a fastball with wings. No wood on that one. He's uncorking what looks like a spitball . Whoooooosh! Strike ! He's out of there. These are the times that make baseball special. Now, Parry Posteriority swings the baseball bat to stretch and enters the batter's box. Another fastball. No contact on that one. A full windup and it's a split-fingered fastball. He pops it up to Orville Baskethands. Well, that's the end of their chances in this inning, Rich.

Figure 2: Some text produced by the baseball announcer.

appropriate format (CFG, DFA etc...) for the learning algorithm. This is not exactly the same task than discovering the parse tree and the bits which generated another set of strings, but if an general algorithm exists for breaking mimic functions, then a similar one exists for learning a context-free grammar.

What is the difference? Attackers of the mimic function are faced with a set of strings from a given language $L(G)$ and they must find the parse tree that generated these strings from a set of examples. The learning proof, however, assumes that the algorithm must distinuish between strings that are in the language and those that are outside of it. A simple reduction splits $L(G)$ into two disjoint languages $L(G_1)$ and $L(G_2)$ that cover $L(G)$. If an algorithm existed which could reconstruct the parse trees of the strings from $L(G)$, then it could determine whether the string belonged to $L(G_1)$ or $L(G_2)$. This would be equivalent to an algorithm learning to predict $L(G_1)$ or $L(G_2)$.

This shows that if there is a general algorithm with the ability to learn to predict the parse trees of a Mimic Function, then it can be applied to learning how to test membership in the language defined by a finite automaton or context-free grammar. Breaking RSA or factoring Blum integers follows. This result does not, however, show how to construct these impermiable grammars short of encrypting the information with RSA and then reducing this to a grammar that must be learned.

A CFL that is Easy to Discover

The last section showed several theoretical results that indicated that there are plenty of CFLs that are difficult to discover. This section will describe a straight-forward attack on a mimic function using a very simple grammar where each terminal occurs on the right-hand side of one production. The next section after it will provide a straight-forward method for avoiding these weak grammars.

Assume that the attacker has the sets of strings $\{s_1, \ldots, s_n\}$ and the set of bits which generated each string, $\{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$. The implementation described in [Way91] will permute the mapping between the bits and the productions in order to introduce further complexity. Assume that this is not turned on and that the same bits will always specify the same sequence of productions and consequentally the same strings.

A straight-forward attack on the system can take be done by finding all sets of terminals that are repeated in some subset of the strings from $L(G)$. Since each terminal occurs in only one production, it is relatively easy to find the common bits. It is only necessary to have enough s_i so that a significant majority of the terminal strings appear at least twice. More duplication makes it easier to solve the problem. It may be possible to provide a good estimate of the "weakness" of this system using statistics, but it is probably not worth the effort. Although the size of $L(G)$ may grow exponentially with respect to the size of G, the number of terminals is bounded by the size of G. This means that the terminals will appear often and a linear number of strings is necessary to recover several instances of a terminal and discover the bits which generated it. The number of strings necessary is inversely proportional to the probability a given terminal will appear.

A CFL that may be Hard to Discover

If it is easy to discover the mapping between terminals and bits when each terminal appears only once, does it get harder when terminals appear more than once in a CFG? What if an attacker has access to enough strings from the language to determine that each terminal $t_i \in \Sigma$ could have been produced by one of several patterns of bits: $\beta_i^1, \ldots, \beta_i^n$? The attacker must convert each string of terminals into bits by choosing one of the β_i^j for each t_i in the string. If there are n choices for each terminal and m terminals in an average string, then there are m^n different possible interpretations of the string– a daunting number when m and n grow large.

But is this a good measure of the burden on the shoulders of the attacker? It only depends upon whether there are any correlations between the bits producing the terminals in a string. If, for instance, t_1 was generated by β_1^1 if and only if t_2 appears next to it and was generated by β_2^1, then there the assumption about exponential complexity does not hold. On the other hand, if t_1 is generated by β_1^1 50% of the time it appears next to t_2 and by β_1^2 the other 50% of the time, then there is no particular problem. The mutual correlations govern the strength of the system.

These mutual correlations, of course, can't be perfectly independent if the message is going to be read. There must be some pattern of the tokens which allows the string to be unambiguously parsed. That means that there must be some correlation between the bits and the tokens. But it can be infinitely more complex. A context-free grammar can encode very complex correlations between tokans as the examples and show.

It is possible to calculate the mutual dependencies for reasonable sized grammars.

Let $\rho(a_i)$ be the probability that a particular terminal a_i will appear in a string generated by a grammar and let $\rho(a_i)^{(j)}$ be the probability that the j-th duplicate of a particular terminal a_i will appear in a string. Also let $\rho(a_i^{(j)}|a_k^{(l)})$ stand for the probability that the j-th duplicate of the token a_i will occur in the same string with the l-th duplicate of the token a_k *when* some version of a_i and a_k appear in the string.

It turns out that information theory and the mathematical measure of information entropy are very applicable here. The theory, which was introduced by Shannon, measures the amount of bits of information contained in a string of events which happen with a certain probability. The process of breaking the system can be thought of infering the bits.

The real problem at this point is determining how much "information" is hidden in each choice of the different copies of a particular terminal a_i. This is a slightly different form of wondering just how many possible copies of a particular terminal a_i must be considered by the attacker at each point. Let $E(a_i)$ be this entropy. It is equal to:

$$\sum_{\forall j} \rho(a_i^{(j)}) \log_2\left(\frac{1}{\rho(a_i^{(j)})}\right) \tag{1}$$

Assume that the attacker is able to discern all of the possible terminals $\{a_1, \ldots, a_n\}$ and the set of possible bit patterns which could have produced each terminal:

$$\{\beta_1^1, \ldots, \beta_1^{B(a_1)}\}, \ldots, \{\beta_n^1, \ldots, \beta_1^{B(a_n)}\}$$

. If a particular string of terminals from a language is: $t_1, t_2, \ldots, t_m$, then the number of possible messages is:

$$\prod_{i=0}^{m} 2^{E(t_i)} \tag{2}$$

Is this a reasonable estimate for the number of possible combinations the attacker must consider? Yes, if we also believe that the size of $L(G)$ is so large that the attacker is not capable of keeping track of the substrings of terminals from the system. That is they cannot maintain any partial knowledge about a the probability that certain instances of the terminals may co-exist. The strength of the system against this sort of partial knowledge can also be estimated with information theory. Let $E(a_i, a_j)$ be:

$$\sum_{\forall k} \sum_{\forall l} \rho(a_i^{(k)}|a_j^{(l)}) \log_2\left(\frac{1}{\rho(a_i^{(k)}|a_j^{(l)})}\right) \tag{3}$$

It is possible to rework equation 2 to use this more complete entropy function by defining the function $\bar{E}(a_i, S)$ to be the minimum of $E(a_i, a_j)$ for all $a_j \in S$. Equation 2 becomes:

$$\prod_{i=0}^{m} 2^{\bar{E}(t_i, \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_m\})} \tag{4}$$

This formulation is more complete, but it is also much more difficult to compute. What is the average complexity of a particular grammar? One simple measure is:

$$\prod_{i=0}^{n} \rho(a_i) 2^{E(t_i)} \tag{5}$$

This equation represents the average number of possible solutions that may exist for a given string from the language. The larger it is, the more secure the system may be against probabilistic attacks.

Transformations for Changing CFLs

Unfortunately for general security, anyone creating a CFL intended to mimic some natural (or programmatic) language is more likely to use a system where each terminal occurs once. In the grammar in figure [1], the names of the outfielders for the Blogs are kept in one production. These grammars can be turned into more secure grammars through the application of two different operations on productions which are called here:*expansions* and *contractions*. These work similarily to the distributive property of algebras.

Given a set of productions, $\{A_i\}$ for a CFG, G, where $A_i \rightharpoonup \alpha_{i,1}|\ldots|\alpha_{i,n}$. The α_i are strings of both terminals and non-terminals. The grammar is *expanded* with these steps:

1. Given the productions from a variable, A_i. Take a subset K of these α_i.

2. For some of the production of the form $X \rightharpoonup \beta_1 A_i \beta_2$ where β_1 or $beta_2$ may be elements from $(\Sigma + V)^*$, create new productions $X \rightharpoonup \beta_1 \alpha_i \beta_2$ for each α_i in K.

3. If this expansion effects all productions containing A_i then elete the productions $A \rightharpoonup \alpha_i$ from G. If K contains all productions, $\alpha_1 \ldots \alpha_n$ then the variable A should not exist.

After a grammar has been expanded then a variable can be *duplicated* with these steps:

1. Find a variable A_i that is in a β_k from an earlier expansion. (Limiting the variable to these situations prevents grammar ambiguity.)

2. Make n copies of a variable A_i by creating n new variables, $A_i^{(1)}, \ldots, A_i^{(n)}$.

3. For each production $A_i \rightharpoonup \alpha_i$, create n new productions: $A_i^{(1)}, \ldots, A_i^{(n)}$.

4. Replace every occurance of A_i in the right side of an equation that was produced by an expansion with a β_k with one of the new variables: $A_i^{(1)}, \ldots, A_i^{(n)}$.

Notice that the duplication must take place after an expansion and it can only effect the productions effected by the expansion. Otherwise the grammar becomes ambiguous.A grammar is *contracted* with these steps:

1. Find some set of strings $\alpha_1 \dots \alpha_n \in (\Sigma + V)^+$ that such that there exist productions of the form $A \rightharpoondown \beta_1 \alpha_i \beta_2$ for each α_i in the set.

2. Create a new variable $B \notin V$.

3. Create the productions $B \rightharpoondown \alpha_1 | \dots \alpha_n$.

4. Replace the productions $A \rightharpoondown \beta_1 \alpha_i \beta_2$ with the one production $A \rightharpoondown \beta_1 B \beta_2$.

The two operations of expansion and contraction are inverses of each other. It is possible to convert any CFG G_1 into any other equivalent G_2 through a combination of these operations. A straight-forward proof of this fact can be constructed by noting that a sufficient number of expansions will convert any CFG, G_1 into a CFG, G_3, which has one variable, the start symbol S, and a set of productions $S \rightharpoondown s_1 \dots s_n$ for each $s_i \in L(G_1)$. If $L(G_1) = L(G_2)$ then a a different set of expansions will convert G_2 into G_3. This chain of expansions can be reversed with contractions to give a set of expansions and contractions that convert G_1 into G_2.

The process of duplication is the most powerful. If it is used generally, then it will create ambiguous grammars. When it is combined with expansions, then it forces the attacker to approach each string differently. It only increases the size of a grammar by a linear amount, but it can force the attacker to examine all the branches of the language. Here is an simple example that illustrates the result of combining several rounds of expansion and duplication.

H_3: S	$\rightharpoondown$	*who what where*
who	$\rightharpoondown$	Bill and Ted | Laverne and Shirley | Thelma and Louise | Bob and Ray
what	$\rightharpoondown$	went hunting bear | went surfing | decided to do nothing | slept late
where	$\rightharpoondown$	in Laguna | in Brooklyn | in Vienna | in Timbuktu

Here it is after the expansions:

$$
\begin{array}{rcl}
H_4\text{: S} & \rightharpoonup & \text{Bill and Ted } \mathit{what}_1 \ \ \mathit{where}_2 \ \mid \text{Laverne and Shirley } \mathit{what}_2 \ \ \mathit{where}_1 \ \mid \\
 & & \text{Thelma and Louise } \mathit{what}_1 \ \mathit{where}_1 \mid \text{Bob and Ray } \mathit{composite} \\
\mathit{what}_1 & \rightharpoonup & \text{went } \mathit{somewhere} \ \mid \text{em nada} \\
\mathit{somewhere} & \rightharpoonup & \text{hunting bear} \mid \text{surfing} \\
\mathit{nada} & \rightharpoonup & \text{decided to do nothing} \mid \text{slept late} \\
\mathit{what}_2 & \rightharpoonup & \text{went hunting} \mid \text{slept late} \mid \mathit{grabbag} \\
\mathit{grabbag} & \rightharpoonup & \text{decided to do nothing} \mid \text{went surfing} \\
\mathit{where}_1 & \rightharpoonup & \text{in Laguna} \mid \text{in Brooklyn} \mid \text{in Vienna} \mid \text{in Timbuktu} \\
\mathit{where}_2 & \rightharpoonup & \text{in } \mathit{place}_1 \mid \text{in } \mathit{place}_2 \\
\mathit{place}_1 & \rightharpoonup & \text{Laguna} \mid \text{Brooklyn} \\
\mathit{place}_2 & \rightharpoonup & \text{Vienna} \mid \text{Timbuktu} \\
\mathit{composite} & \rightharpoonup & \text{went } \mathit{somewhere} \text{ in } \mathit{place}_1 \mid \text{went } \mathit{somewhere} \text{ in } \mathit{place}_2 \mid \\
 & & \mathit{nada} \text{ in Brooklyn} \mid \text{decided to do nothing in Laguna} \mid \\
 & & \text{slept late in Brooklyn} \mid \\
 & & \text{slept late in } \mathit{place}_2 \mid \text{decided to do nothing in } \mathit{place}_2
\end{array}
$$

The practical effect of the duplication should be obvious The way that the verb and object phrase of the sentences in $L(H_4)$ depends upon the noun. Any information about the bits which produced the sentence "Bill and Ted went surfing in Laguna" will not help decode the sentence "Bob and Ray went surfing in Laguna." Straight-forward code-book attacks like the one described in a previous section are not possible.

Moreover, the size of the language $L(G)$ grows exponentially with the number of multi-production variables between the start symbol the terminal. This means that a well-designed grammar can grow linearly, but the size of its language will grow exponentially. A large grammar could produce very intractible problems for the attacker.

In practice, grammars like the ones in figure [1] can be expanded, duplicated and contracted several times to produce a new grammar where many of the terminals appear in two or more of the productions. Expansion can lead to exponential growth in the grammar, but this can be contained if the expansions only effect a few productions at a time.

Probability and Mimickry

The last section showed how to use three basic operations to construct grammars that are more secure. Naturally, weaknesses can emerge in some situations when probabilities are not preserved well. Ordinarily, the definition of mimic grammars includes a probability that governs how often a particular production will be chosen. The last section did not mention the effects of these for the sake of clarity. Occasionally, the effects of these probabilities can conspire to nullify many of the effects of expansions and duplications.

Consider this version of H_5:

$$H_5 : S \ \rightharpoonup \ aZ_1\{.999\}|bZ_1\{.001\}$$

$$Z_1 \rightharpoonup a\{.25\}|b\{.25\}|c\{.25\}|d\{.25\}$$

The values in brackets ($\{,|\}$) signify the probability that one production will be taken. If the production $S \rightharpoonup bZ_1$ is expanded, the language becomes:

$$\begin{aligned} H_61 : S &\rightharpoonup aZ_1\{.999\}|Z_2\{.001\} \\ Z_1 &\rightharpoonup a\{.25\}|b\{.25\}|c\{.25\}|d\{.25\} \\ Z_2 &\rightharpoonup ba\{.25\}|bb\{.25\}|Z3\{.5\} \\ Z_3 &\rightharpoonup bc\{.25\}|bd\{.25\} \end{aligned}$$

The expansion will only effect .1% of the sentences created by this grammar. It is probably an unwise use of the resources to modify the grammar by expansion. It follows that the best choices for expansion are the productions with the highest probabilities.

If there are good choices, then it also follows that there are some expansion/duplication choices that are better than others. The entire purpose of expanding and duplicating a grammar is to present an attacker with ambiguous values of each token. They can't be sure if it has one value or another. The result is that they must confront exponentially many strings.

A Practical Key System to Making a Grammar More Secure

The last two sections have discussed how to re-arrange a grammar so that each token can be produced by several different patterns of bits. This forces the attacker to choose between both interpretations when confronted with a token. When there are no cross-correlations between the tokens, then this may present a fairly formidable barrier to attack.

Here is an algorithm for converting a given grammar G into a strong grammar that will resist this form of brute-force attack:

Repeat this next step as often as desired:

1. Choose a random production and expand it.
2. The expansion may produce n different copies of the original production: $A_i \rightharpoonup \alpha_j$. For each variable A_k in α_j, choose a different version of the variable A_k to appear in each of the duplicated versions of α_j. Create these A_k via duplication.
3. Choose a random variable.
4. Contract some of the productions if this can be done. Choose the subset of productions at random.

The selection of a random permutation can be done by choosing bits from a pseudo-random source. Two parties can use this to create an identical version of a grammar that has undergone expansion and duplication by agreeing on the pseudo-random algorithm and grammar in advance. The seed of the random bit generator serves as a key.

If the probabilities are preserved, then this will produce a grammar that has a perfect distribution of cross-correlations.

Efficient Mimic Functions

Originally, mimic functions were conceived as part parlour trick and part sleight-of-hand encryption. The implementation is intended to encode information in English-like text. This is useful, but it isn't always efficient. Many of the experimental grammars would produce an "encrypted" file that was 20 to 30 times larger than the original. This isn't practical.

It is possible, however, to rely upon the theoretical results described here to construct efficient grammars with a set of tokens that is equivalent to the set of characters or words. In the ASCII world, this means using all 256 or 128 characters as tokens. The output of these grammars will not be readible, but they will inherit all of the theoretical security described in the last two sections.

For simplicity, assume that the data comes in blocks of one byte and there are 256 tokens in the grammar $\{a_0, \ldots, a_{255}\}$. The grammar consists of these tokens, a set of variables, $\{v_0, \ldots, v_n\}$ and a set of productions of the form:

$$v_i \rightharpoonup a_0V_0|\ldots a_{255}V_{255}$$

, where the V_i are possibly empty strings of variables. One of the v_i is designated the start symbol.

How is the composition of V_i's determined? It can either be generated at random or through the expansion and contraction operations. The average number of variables in the V_i's determines the average length of the strings in the language. If the average size is 1 or larger, then there will be non-trivial number of infinite strings in the language and it will not be practically useful. If δ is the average length of the V_i's, then δ^i is the probability that a given string will be i tokens long.

Turing Machines and Mimickry

The last sections discussed using CFG's for mimickry. In [Way90], the mimic functions that produced all recursively enumerable languages were presented as VW-grammars. This construct fitted nicely with the other items in the paper and it also allows us to apply all of the results in this paper to the larger classes of functions computable by Turing Machine. This section, though, will address the class of r.e. mimic functions as turing machines running backwards. Many of the classical facts about turing machines are directly applicable to this problem, but there a enough details that make the use of these general theorems problematic.

Let a turing machine consists of an infinite tape containing symbols from a tape language, Σ, and a finite control with a set of states, S. The infinite tape has a pointer to the symbol at one location and this head can write a new symbol at a location, and/or move left or right. At the beginning, the tape contains an initial string of symbols. A function, $\delta(s,t)$, governs the action of the turing machine based on the current state, $s \in S$, and symbol at the tape head, $t \in \Sigma$. It performs three actions. First it changes the finite control to a new state, second it may write a new symbol at the location of the tape head and third it may move the tape head left or right. A subset of the state set, S, are marked as either start states or end states. If a machine begins with a particular string on its tape and finds itself in an end state with an empty tape, then the turing machine is said to *accept* that string. $L(M)$ is the symbolic representation for the the set of strings accepted by the turing machine, M.

A *mimic* TM, M' can be built from a turing machine, M by running it backwards. Let $\delta(s_i,t) \rightharpoonup (s_j,u,\{left,right\})$ mean if the TM is at state, s_i, and the tape head is reading symbol t, then move to state, s_j, write symbol u on the tape and move the head to either the left or the right. Call the reverse of δ, δ'. If $\delta(s_i,t) \rightharpoonup (s_j,u,\{left,right\})$, then $\delta'(s_j,u) \rightharpoonup (s_i,t,\{right,left\})$, where the movement left or right is reversed. All the start states of M become the end states of M' and all the end states of M become the start states of M'.

The only real difference between a mimic TM, M' and its predicessor, M is the starting and ending condition of the tape. M starts with a string on the tape and ends when the tape is empty. M' starts with an empty tape and ends up with a string in $L(M)$ on the tape.

Naturally, there are some functions that are not invertable because there are several inputs to δ that produce the same result. This is where data can be encoded. One actual algorithm for converting a set of choices to bits is shown in the appendix, but there may be others. The data is recovered when the TM executes this one branch of the delta function.

One string in $L(M)$ can be produced with M' by starting in an end state of M with an empty tape and running M' with δ' until a start state of M (an end state of M') is reached. When a choice must be made by δ', then data can be encoded. An entire file can be produced by generating strings in $L(M)$ until all the data is used.

This data can be recovered by running M on the strings. If $\delta'(s_i,u) \rightharpoonup (s_j,t_j,L)|\ldots|(s_k,t_k,R)$ and the algorithm in the appendix is used to make the choices, then the bits can be recovered when the result of δ is state s_i. These bits will be produced in the opposite order.

Theoretical Bounds and Practical Questions

Classical computer science borrowed many results about turing machines from the realm of logic and philosophy. [HU69] provides a good discussion of this area. The central theorem of this theory states that it is undecidable whether a turing machine will accept a particular string. This result means that another turing machine cannot tell for certain whether a turing machine will ever stop in an end state when starting with a particular string. In some cases, the turing machine will enter an endless loop and never halt and this cannot be predicted by another TM.

Should this worry the attacker? No. If a message has been encoded in a string, s, from $L(M)$, then M must halt when it finds s on its tape. Can the sender place fake strings in the message that are not in $L(M)$? Not ones that will cause M to fail to halt because the sender itself would fail to decode them.

But, there are other features of undecidability that may prove vexing. Rice's theorem states that every non-trivial property of recursively enumerable sets is also undecidable. This means that it is undecidable whether two TM's accept the same languages. Theoretically, it means that breaking such a system with a machine is an undecidable proposition.

In reality, this problem can only arise when there are infinite strings in the language. These cases are certainly possible, but they can be avoided by modelling them with a finite approximation. The samples from real systems will all have finite strings. It is unclear if any of these results from computability have any application here. Never-the-less, the results from the section of the paper on Context-Free Mimic Functions also applies here because the recursively enumerable languages can be modeled as Van Wijngaarden grammars. And these are certainly more complex for an attacker.

Conclusion and Notes about Novelty

Most of the best work on cryptography has tried to guarantee security by relying on the assumptions that certain problems like factoring large integers [RRA78] or solving knapsacks [**?**] is a difficult problem. If these problems are hard, then the cipher system may be secure. This work tries to encode information in three different theoretical models of general computation and these models are much more general than the other specific cases. This is both a blessing and a curse. In one sense the classes of problems are larger and therefore must be at least more complicated to solve. Turing machines can model factoring, but factoring cannot model turing machines. On the other hand it may not be possible to find hard-to-break examples of the set. The first theoretical section showed that if a general method for breaking Mimic Functions could be discovered through a probabilistic learning theory, then it would be possible to break RSA and factor Blum integers.

But these theoretical reductions are only half of the problem. The Knapsack system of Merkle and Hellman taught the lesson that it may not be possible to identify significant classes of difficult to solve NP-complete problems. The same may be true for grammars. The construction here attempts to produce a set that may be difficult to crack and it offers a straight-forward method for estimating the brute-force effort that would be needed to mount an attack on a particular grammar. It is not certain, however, whether this classification of difficult to crack grammars is a solid one. This job is left to the readers who will, I hope, be able to devote some time to posing attacks on the system.

The last section of the paper discussed turing-complete mimic functions which are, in a sense, as big as all cryptography. All encryption systems can be modeled as Turing Machines and breaking a system is equivalent to infering the machine. Although this is technically an undecidable problem, it is unclear whether these results can add any real knowledge to the security of the system. It shows that that some theoretical results are not immediately valuable. Never-the-less, there is no reason not to expand the use of the system beyond context-free grammars to Turing Machines. All of the previous results hold and it should be obvious that the grammars are more robust, more realistic and at least as difficult to break.

Appendix: One Way of Encoding Data in Non-determanism

This is a general technique that converts data into a stream of choices that is based Huffman coding. [Huf51, Sto88]. This can be used to turn non-determanism into a technique for encoding information.

Start with a set of choices, $\{c_1, \ldots, c_m\}$. Let, $p(c_i)$ be the probability that choice, c_i, should be chosen. This algorithm will produce a tree, that can be converted into an encoding device:

1. Construct a node, n_i, for each choice, c_i. Each node will have a number, $P(n_i)$, associated with it. For nodes, $\{n_1, \ldots, n_m\}$, set $P(n_i)$ equal to $p(c_i)$. Each node can also have two descendents, which in the case of nodes, $\{n_1, \ldots, n_m\}$, are set to be empty.

2. Construct a set, S. Place all the nodes in S.

3. While S has more than one node in it, do:

   (a) Find the two nodes, n_i and n_j in S with the smallest values of P.

   (b) Remove these nodes from S.

   (c) Construct a new node, n_k, with n_i and n_j as its descendents. Let $P(n_k) = P(n_i) + P(n_j)$. Place n_k in S.

4. The one remaining node is the root of the tree. At the leaves are the choices.

Bits can be converted to choices with this algorithm:

1. Start at the root. Let N be the current node of the search, which at this point is the root.

2. While N is not a leaf node do:

(a) Sample one bit from the data stream.

(b) Use this one bit to choose one of the two descendents for the new value of N. In the simplest form this choice can be hard-coded. I.E. zero means go left, one means go right. But a complex machine could also be used to make the choice based upon the node, n_i and some pseudo-random state.

When the data is evenly distributed between 0's and 1's and the function that maps the bits to the choices is unbiased, then f $D(n_i)$ is the distance between n_i and the root, then the probability of a choice, c_i, being made is $2^{D(n_i)}$.

References

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

[HU69] J.E. Hopcroft and J.D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, Massachusetts, 1969.

[Huf51] D Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40:1098–1101, 1951.

[JKM90] H.N. Jendal, Y. J. B. Kuhn, and J. L. Massey. An information-theoretic treatment of homophonic substitution. In *Advances in Cryptology–Eurocrypt '89*, New York, 1990. Springer-Verlag, Lecture Notes in Computer Science.

[Kea89] Michael Kearns. *The Computational Complexity of Machine Learning*. PhD thesis, Harvard University Center for Research in Computing Technology, May 1989.

[KV89] Michael Kearns and Leslie Valient. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 433–444, Seattle,Washington, May 1989.

[Riv91] Ron Rivest. Cryptography and machine learning. In *Asia Crypt 91*, 1991.

[RRA78] Adi Shamir Ron Rivest and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(11), 1978.

[Sim84] Gus J. Simmons. The prisoner's problem and the subliminal channel. In David Chaum, editor, *Advances in Cryptology: Proceedings of Crypto '83*, pages 51–67, New York, 1984. Plenum.

[Sto88] James Storer. *Data Compression.* Computer Science Press, Rockville, Maryland, 1988.

[Val84] Leslie G. Valient. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[Way90] Peter Wayner. Mimic functions. Technical Report ???, Cornell University Department of Computer Science, December 1990.

[Way91] Peter Wayner. If sb266 wants plaintext, give them plaintext... *RISKS Digest*, 11(71), May 1991.