GIT-CERCS-04-24

Authentication Protocols making use of Context free Grammar: Guessing Strings

Abhishek Singh, David Dagon
Georgia Tech. Information Security Center (GTICS)
Center for Experimental Research in Computer Science (CERCS)
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{abhi,ddagon}@cc.gatech.edu

Abstract

Until now context free grammars (CFG) have been used mainly for the design of programming languages. There have been several studies, which demonstrate the relationship between learning theory and number theory. This paper presents protocols, which use context free languages for an authentication protocol using one-time authentication information. This authentication information can be used for the generation of one-time passwords. This paper also analyses of the difficulty of guessing strings in such a language. The paper further discusses structured and unstructured techniques for generating strings, which is a part of ongoing work. We present here our approach and some of our early results.

1.0 Introduction

Reusable passwords have traditionally been the weakest link in host security. Often they are poorly chosen and therefore easy to guess, or their storage is poorly protected. In recent years the reusable password has also become a favorite target for network eavesdroppers [14]. Since passwords must often be sent ``in the clear," they can be easily captured by eavesdroppers and used later for intrusions. Intruders often leave behind automated ``sniffer" programs in order to harvest passwords for later use.

Systems like Kerberos encrypt all sensitive network traffic, foiling both active and passive attacks. But the user is required to use only ``kerberized" hosts to participate; remote logins over an unprotected link or from a dumb terminal are still vulnerable. This problem also exists with other application-level encryption protocols like SSH. They are great tools and should be used when available, but in many situations they are either unavailable or inconvenient.

“One-time” passwords provide one solution for fixed static password, as they allow users to authenticate over insecure links without revealing any sensitive information in the process. As the name implies, a one-time password is only good for one authentication session, after which it automatically expires, and a new (previously unused) password becomes valid.

Design of the schemes to generate one time passwords has been one of the interesting areas of research. Leslie [13] in his paper proposed the use of hash functions for the generation of one-time password. Symmetric keys, asymmetric keys and challenge response are also being used for the design of one-time passwords. Learning theory till now has been used for the design of programming languages. This paper explores the usage of learning theory for the design of authentication protocol generation one-time authention information. We present here some of our initial results.

Section 2.0 discusses the hardness of learning the grammars. Section 3.0 discusses the hardness of guessing a string. Section 4.0 presents the protocol making use of mutually independent grammar for the generation of one-time authentication information. In section 5.0 we present the structured and unstructured manner of generation of strings. Section 6.0 presents a method to prevent replay of previously generated strings. In section 7.0 we present a protocol using distinct CFGs. Section 8.0 notes the need for future work.

2.0 Learning Grammars

In [2], a *representation class* was defined as a class of objects that can be represented by strings over some alphabet. The set of all context free grammars is a representation class. A *learning algorithm* is one that tries to learn a representation class from positive and negative instances. [2] presents a formal definition of the learning process. We let *c* and *h* be two instances of some representation class, where *c* is the target class (the class to be learned), while *h* is the output class. We let *A* be the learning algorithm, which takes positive and negative instances as an input. *A* outputs class *h.* We use $e_c^+(h_A)$ to denote that algorithm *A* errors on a positive instance of *c* and classifies them as a negative instance of *h*. Similarly, $e_c^-(h_A)$ represents the probability that *A* errors on negative instances of *c*. When $e_c^+(h_A) = e_c^-(h_A) = 0$ then the classes *c* and *h* are identical. This is the ideal case of complete learning, with zero errors, and can be classified as strong learning. There can be another case where the error is not zero, but it is a small, negligible value, called "weak learnable". [2] presents a representation class *c* as weakly learnable if there exists a probabilistic polynomial time algorithm *A* that on access to a set of positive and negative instances of a target representation $c \varepsilon C$, generates an output representation *h*, so that $e_c^+(h_A) < ½ - 1/O(|c|^K)$ and $e_c^-(h_A) < ½ - 1/O(|c|^K)$ and for some constant k. |c| is usually taken as some polynomial in n, the length of the sample instance

If A_{CFG} is a learning algorithm for context free grammars, then for any given context free grammar G_1, with access to a small subset of positive and negative instances (i.e., strings that belong to G_1 and are outside of G_1 respectively), A_{CFG} can implicitly construct a grammar G_2 which is *almost identical* to G_1. Almost identical can either be a case of strong learnability or weak learnability. Strong learnability lies in the undecidable domain. We present a proof.

Given a set of strings $S_1, S_2, S_3 \ldots S_n, S_{n+1} \ldots S_m$ there can be an infinite number of grammars that can generate these strings. The exact context free grammar is a grammar which can be generated by using *S1, S2, S3…. Sn* and which successfully accepts $S_{n+1}, S_{n+2} \ldots S_m$. Let's assume that there exist an learning algorithm *A* which on an input set of string $S_1, S_2, \ldots S_n$ can predict the exact context free Grammar *G* which generated it. Now let us pick any context free grammar G_1 and generate strings $S_1, S_2 .. S_n$ by randomly selecting rules from the grammar. These strings are passed through the learning algorithm *A*, which gives grammar G_2 as the exact context free grammar. However, it is known that given two context free grammars G_1 and G_2, whether the languages are equal, $L(G_1) = L(G_2)$, is an undecidable problem [1]. Therefore there exists no way to verify if the language generated by G_1 and G_2 are equivalent. Hence there cannot be any algorithm, which can give an exact context free grammar. So the strong learnability of context free grammars lies in the undecidable domain.

Strong learnability implies learning the whole CFG. Since strong learnability requires false positives and false negatives to be zero, the output class is the same as the target class. Hence it classifies the instances with probability one. This makes strong learning a very strong assumption for cryptographic purposes. Randomly, we can guess whether a target class and an output class are identical with probability ½. For cryptographic purposes must define how much better a learning algorithm can learn a grammar compared to random guessing. A representation class is *weakly learnable* if there exists an algorithm that can do a little better as a classifier than random guessing. Weak learnability expects a learning algorithm to classify the output class just *slightly* better than random guessing.

According to results of Kearns and Valiant [2], if ADFA n p(n) denotes the class of deterministic finite automata of size at most p(n) that only accepts strings of length n, and ADFA p(n) = $\cup_{n>1}$ ADFA n p(n) , then for some polynomial p(n), the problem of inverting the RSA encryption function, recognizing quadratic residues and factoring blum integers are probabilistic polynomial-time reducible to weakly learning ADFAp(n). They further state that any representation class whose computational power subsumes that of NC^1 is not weakly learnable. Since CFGs contain the computational complexity class NC^1, they are also not weakly learnable under the similar cryptographic assumptions as that of ADFA. Factoring RSA, recognizing quadratic residues and factoring Blum integers are reducible to weakly leaning CFG's. So it can be concluded that it is hard to learn CFGs. **We use the term "guessing a string" throughout this paper to refer to the problem where, given a set of strings in a language, one must guess another string which belongs to the same language**. This problem of "guessing a string" still lacks a strong theoretical foundation. This leads to an interesting research challenge: If a context free grammar is hard to learn can it be used to construct any useful security protocols?

3.0 Guessing a String

Even though theoretically for an average case it is hard to learn the CFG, there have been several studies using artificial neural networks [9] and genetic algorithms [10] to learn grammars. Singh et. al.[4] detail studies of learning context free grammars and the design consideration of an algorithm to make learning the grammar hard. Note that there can be many algorithms that make learning hard. One of them is presented in [4]. Even though it is tough to learn a grammar, the difficulty of guessing a string is still an interesting research problem. This paper presents some of our initial results on string guessing. The main purpose of the research is to present some of our initial research results to address the issue of guessing a string.

To assess the hardness of guessing the string, several experiments (presented in the appendix) were conducted. The algorithm in [4] was used to generate random strings. Sample strings were collected, broken into different combinations and parsed through the CFG. One percent of the combinations were accepted.

This test demonstrates that the acceptance rate is too high for cryptographic purposes. In [3] a protocol making use of CFG for the generation of one-time authentication was proposed. In the proposed protocol only one string was sent at a time. However our experiments show that it might be easier for an adversary to guess the string belonging to a language than to learn the grammar. It can be seen from the above experiments that, given a set of strings from a grammar, guessing a string is easier than learning the grammar. Therefore in order to get a stronger resistance to guessing we propose the use of mutually independent grammars to generate "n" strings. The intuition is this: if an adversary has a measurable probability of guessing a string successfully, use many such independent grammars to force the adversary to repeat their success.

Two events are said to be independent if the knowledge about one event does not affect the probability of the other event. So if the grammars follow an independent distribution then, the probability of guessing a string S_1, will have no effect on the probability of guessing a string from another grammar. Hence under the assumption that independent grammars can be generated, we define an adversary's chance of success as:

$$\Pr[\textit{Guess } S] = \prod_{i=1}^{n} \Pr[\textit{Guess } S_i]$$

Here, we let S be a concatenation of strings in independent grammars, $S = S_1 | S_2 | S_3 | \ldots S_n$. Hence to minimize the adversary's chance of success, it should be ensured that the probability of guessing of a string $\Pr[\textit{Guess } S_i] < ½$. By enforcing this rule it can be ensured that as we add more and more grammars the chances of guessing the string S ($= \prod_{i=1}^{n} S_i$) reduces. This results from the use of independent grammars and minimizes the guessing ability of an adversary. For the modified version of the protocol, in order to successfully guess a string S, an adversary will have to guess all the substrings $S_1, S_2 \ldots S_n$. Even if there is a one percept acceptance rate for randomly generated strings, as demonstrated by our simple test, the use multiple independent grammars provides a promising way to use such grammars in a security protocol.

Hence the grammar for the proposed protocol should follow two characteristics.

- The grammar should be hard to learn. This makes guessing the grammar a tough task, and minimizes Pr [*Guess* S_i].
- It should be independent. This makes guessing the string difficult. Instead of guessing one string, an adversary will have to guess all the substrings.

4.0 The Protocol making use of mutually independent context free grammar

The version of the protocol proposed in this section is making use of mutually independent context free grammars. In the sub section 4.1 we present the protocol and address the issue of guessing of strings. In Section 6.0, we address the issue of replay attacks.

For this discussion, we use the following terms:

- The entity, which generates the one time authentication information, is called as the *generator*.

- The entity, which validates the one time authentication information, is called as the *verifier*.
- Output of the grammar G_l is called a string S_l
- The output of the protocol, a concatenation of the strings, is called an authentication token $S = S_1| S_2 |.... |S_n$.

The generation of the string has to be performed on the generator's end and the verification of the string has to be performed on the verifiers end.

4.1 Authentication protocols using mutually independent grammars.

The generator and the verifier both share the same grammar $\{G_1, G_2, G_3.... G_k\}$. The number of grammars "*k*" and the grammars themselves are kept secret. We also choose a fixed "n" where n is the number of strings to be concatenated. It has to be ensured that k > n otherwise an adversary can make an easy guess about the number of grammars. If k < n then a substring from the same grammar can appear in the string, and independence is lost. **For this subsection it is assumed that the replay of the previously generated strings is not possible**. Prevention of replay attacks is discussed in the section 6.0.

4.1.1 Generation of the Strings

Figure 1 shows the pseudo code for the generation of the string. The number *l* is the length of the string generated by the grammar. The grammar at the position *l* mod *k* is chosen to generate a string of length *l*. This process is repeated "n" times and the strings generated after "n" operations are concatenated and sent to the verifier.

Procedure Generation of n strings
Input: n, number of strings.
Begin
Step 1. Start for i equal to 1 to n
Step 2. Generate a random *l*.
Step 2. Pick up a grammar at position *l* mod *k* = G (*l* mod *k*)
Step 3. Generate a string of length *l*. = S_l.
Step 4. Repeat the process n times ensuring that every time a different independent grammar is used to generate each string.
Step 5. Concatenate to get the strings to get authentication token $S = S_1| S_2|... |S_n$
Step 6. Send the S to the verifier
End

Figure 1.0 Pseudo code for the Generation of the "n" Strings.

4.1.2 Verification of the Strings:

Figure 2 shows the pseudo code for the verification of the string. The verifier takes the authentication token S and breaks it up into strings $S_1.... S_n$. Verifier then calculates the length l_l of the string S_1. The string S_1 is parsed through the grammar at the position *l*. The process is repeated for the "n" strings. If all the "n" strings are successfully parsed then an authentication procedure is complete.

Procedure: Verification of the "n" strings
Begin
Step 1. Get the authentication token $S = S_1|S_2....|S_n$
Step 2 Start for i= 1 to n
Step 3. Take a string S_i
Step 4. Calculate the length of the substring $S_i = l_i$
Step 5. Parse the string at grammar at position l_i .
End for
If all the strings are accepted then authenticate else reject.
End

Figure 2.0 Pseudo code for the verification of the "n" strings

4.2 Difficulty of guessing the string

The protocol to generate one-time authentication information is still susceptible to man in the middle attacks. Our protocol does not address these, and so other existing solutions should be used. For example, we assume that the generator is using a trusted platform to create the strings and the communication infrastructure between the generator and the verifier is using a VPN to prevent monitoring.

The proposed adversary is given all the terminals, which can appear in a language, and a string with the last missing terminal. An adversary will only have to successfully guess the last terminal, which can occur in a string. If the number of terminals is 25, and out of these 25 terminals if 12 can occur at the last position then the random guessing probability of an adversary becomes 12/25.

Since the output of the protocol consists of "n" strings from mutually independent grammars at each instance, an adversary will have to perform the *guessing of string* "n" times. Hence the probability of guessing for an instant reduces to $[12/25]^n$. Here we introduce the notion of a confidence factor.

Confidence factor = 1 – probability of successful guess.

$$\Pr[\text{Guess } S_1] = 12/25 = 0.48$$
$$\Pr[\text{Guess } S_1 \mid S_2] = [12/25]^2 = [0.48]^2$$
$$\Pr[\text{Guess } S_1 \mid S_2 \mid S_3] = [12/25]^3 = [0.48]^3$$
$$\vdots$$
$$\Pr[\text{Guess } S_1 \mid S_2 \mid \ldots \mid S_n] = [12/25]^n = [0.48]^n$$

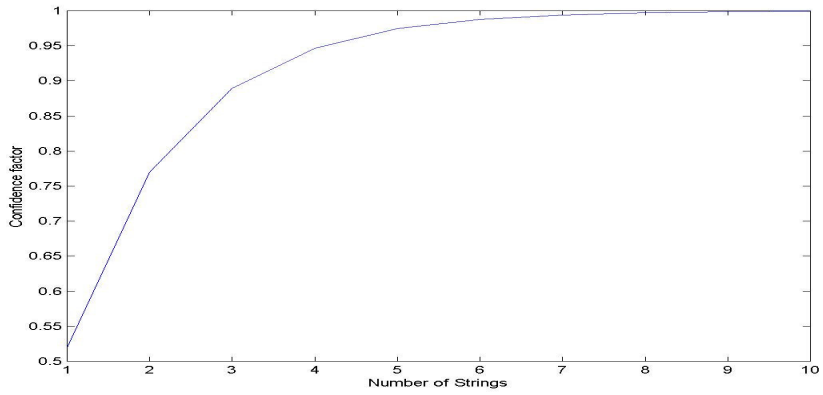

Figure 3.0 Graph of confidence factor vs. Number of strings in a grammar.

Hence it can be seen that as the number of strings increases, the confidence factor also increases. This increase in confidence is exponential in nature. Even though it might be simpler for an adversary to make a guess about the one string, since the protocol is making use of "*n*" strings at an instant, the adversary will have to make a successful guess about all "*n*" strings. The usage of "*n*" mutually independent strings at each instant reduces the guessing probability of an adversary. In a practical situation an adversary will get only a few limited attempts before the authentication mechanism detects the attack. For example, the cost of each guess might take time, or a user might

have a limited number of authentication attempts. So with a very high confidence factor and a limited number of attempts it might be tough for an adversary to guess the output of the protocol.

5.0 Structured and Unstructured manner of generating string.

There have been many studies to determine patterns in grammars and learn CFGs. The earliest method for CFG learning was proposed by Solomonoff [6]. In this approach the learner is given a positive sample S^+ from a language L and has access to a membership oracle for L. Solomonoff in his work proposed to find repeated patterns in a string: for every string $w \in S^+$, delete sub-strings from *w* to create new string *w'* and ask the oracle if *w'* is in the language. If it is, then insert repetitions of the deleted sub-strings into the string and ask if the new string is also in the language. If so, then there must be a recursive rule. For example, if there are several strings of type $a^n b^n$, then we can infer that $A \rightarrow aAb$ is in the grammar. This method is inefficient and does not include all the context free grammars. It has been shown in [8] that this method fails if we have sequential embedding. For the same example, if we have $A \rightarrow aAb \mid cAd$ then this method will not work.

Another attempt for predicting context free grammars is reported in [7]. It is assumed in [7] that there is a teacher that gives a set of strings to a learner or a program. The program first checks if the sample is already in the language. If it is, then the program decides in consultation with the teacher if it is finished or if it should continue with more samples. If the string is not in the machine's language, the program adds to the grammar a production that adds this sentence to the language, attempts to verify that this production does not lead to illegal strings, and requests the next sample from the teacher. This method heavily depends upon the order in which samples are presented. This algorithm may choose a grammar rule that is too general. At a later stage a new sample may come up and even the predicted grammar may generate text that is not in the language. An example is when predicting the language of arbitrarily long strings of *a*'s or *b*'s, but not both *a* and *b*. If samples *a* and *aa* are given the algorithm will produce a partial grammar with the following rules

$S \rightarrow a \mid SS$

Now if sample *b* is given it produces $S \rightarrow b$. This grammar can produce the string *ab,* which is not in the language. The algorithm has made use of only positive instances to predict the grammar. Neural networks were used to learn context free grammars in [9]. Neural networks were trained with both positive and negative instances. As in [9], the order of input was important. The network was trained first with shorter strings and then with longer instances. This scheme proved good only for small context free grammars having around four to five terminals. It failed to scale for larger context free grammars.

Genetic algorithms are used to learn stochastic context free grammars from a finite sample in [10]. Genetic algorithms [11] are a family of robust, probabilistic optimization techniques that offer advantages over specialized procedures for automated grammatical inference. A stochastic context free grammar is a variant of ordinary context free grammars in which grammar rules are associated with a probability, a real number from the range of [0,1]. A genetic algorithm was used to identify the context free grammar. This algorithm took a corpus *C* as an input. This corpus *C* for a language *L* was a finite set of strings drawn from *L* where each strings $\alpha \in C$ is associated with an integer f_α representing its frequency. As in [9] this only can be used for small grammars. The chief limitation of this approach is the cost involved in evaluating the fitness of each candidate solution, which required parsing every string in each possible way. The number of parsing operations increased exponentially with the number of non-terminals. Hence, this scheme proved prohibitively costly for more than 8 non-terminals. As it can be seen in the functional approach, patterns can be found. The main problem of such an approach is the visibility of the patterns which can be inferred in the strings. The current way to generate the grammar is called a functional approach. For functional approaches there is random generation of productions. The main problem in the functional approach is that if the grammar is composed of functional units, after seeing the patterns, functional units can be easily guessed. For example: $A \rightarrow$ nsew, (or $A \rightarrow$ a) ($B \rightarrow$b) ($C \rightarrow$c). Over the time, an adversary will find these basis productions rules:

cb == CB
cc == CC

Over time, an adversary can make a good guess about production rules. We have not quantified the power of an adversary guessing production rules in a functionally-generated grammar. However, we believe it is trivial to guess many such rules, depending on the grammar.

There are two tiers of difficulties that we want to create for an adversary. The first one is predicated upon the hardness of learning context-free grammars. Earlier works showed that learning context free grammars from example strings is a hard problem [2] [12]. Taking inspiration from those results, we conjecture that if a generator authenticates himself with a verifier using authentication tokens generated by a CFG (which is not known to the adversary), then it may be hard as such for the adversary to present a false acceptable token which must be a string from the grammar. However after getting certain strings it might be simple for an adversary to guess the pattern which appears in the string.

Here we introduce the notion of structured and unstructured manner of generation of strings. We call the approach of generating the random strings from a grammar as an "*unstructured*" manner of generating the string. As a second measure of hardness, we also propose to generate the strings in a "*structured*" way, so that the verifier always expects the generator to present the authentication tokens in a specific order. The knowledge of generating string in a structured manner will be shared by both the verifier and the generator. Therefore, for the adversary, it does not suffice to just learn the grammar and present 'any' string. The adversary needs to learn the 'order' as well.

5.1 Unstructured manner of generating Strings

For an unstructured manner, it can be assumed that the strings are generated randomly. If the strings are generated in an unstructured manner, then given a set of strings, an adversary can learn much about a language. A simple example appears in figure 4.0: for a context free language $L = \{a^n b^m c^p: m, n, p \geq 0, \text{ and } m \neq n \text{ or } m \neq p\}$ if one random string, ab^2c, is generated, an adversary learns that (1) the language contains a, b, c., and (2) b follows a and c follows b. Now, given a second string "a^4 b?", with the last terminal missing, an adversary can make an better guess about the last eight bits in the string.

5.2 Structured manner of Generating Strings for N mutually independent grammars.

A structured generation is defined as the production of strings in a grammar following certain rules and regulation. For example, we might require that the generated strings follow a certain sequence, or are produced in-order, based on length. This shared knowledge has to be kept secret by both the generator and the verifier. So for structured generation of strings the total number of terminals, the grammar and the structured manner of generating string will have to be kept as a secret. One of the simplest structured manners of generating strings can be with each string a new terminal is introduced. So even though an adversary has the information about all the previously generated strings he will still have to guess the last eight bits. Explaining it with the same example which was used for the unstructured manner of the context free language $L = \{a^n b^m c^p: m, n, p \geq 0, \text{ and } m \neq n \text{ or } m \neq p\}$ is used to generate only two strings in a controlled fashion in the order of $a^2 b$, $a^3 bc^2$ then given $a^2 b$ and **$a^3 b$?** it might be hard for an adversary to guess the last eight bits of the given string since with each new string a new terminal is being introduced and the loops follow particular pattern.

The intuition is this: a structured manner of string generation requires a specific string at a particular instant. Apparently it might be hard to generate the strings that are specifically expected at every instant without learning the CFG. However according to the results of Kearns and valiant, weakly learning is a hard problem. It has been shown that factoring RSA, recognizing quadratic residue, and factoring blum integers is equivalent to weakly learning CFG. Hence we believe that the scheme of generating strings in a structured manner is secure. We have not provided a formal proof, and leave for future work the investigation of this intuition.

If all the terminals are generated randomly, and for the structured manner of generation of strings a new terminal is introduced after each instant. For an adversary he will be inferring these terminals as random terminals. In order to successfully guess the new terminals, which have been generated, randomly adversary will have to make a random guess or learn the grammar. For n bits probability of random guess is $1/2^n$. From the collected set of terminals, he will try to guess the next missing terminal by learning the grammar. However according to the results of Kerns and Valiant, weakly learning is a hard problem. It has been shown that factoring RSA, recognizing quadratic residue and factoring blum integers is equivalent to weakly learning CFG. Hence if the strings are generated in a structured manner it is secure.

As per the proposed protocol in section 4.0, at a particular instant S (= $\prod_{i=1}^{n} S_i$) strings generated from the mutually independent grammars are sent at each instance of authentication. So given "n" strings along with the previously generated strings with the last missing terminal, if the strings are generated in an structured manner an adversary will have to make a guess about 8*n bits. If a new terminal is introduced after every instant then the structured manner of generation of the grammars requires that the grammar will have to be replaced after all the terminals have been used.

CFG $L = \{a^n b^m c^p : m, n, p \geq 0 \text{ and } m \neq n \text{ or } m \neq p\}$ The language L is used to generate only two strings	
Unstructured Manner	**Structured Manner**
Generator and Verifier will have to keep secret • CFG • Terminals a, b, c	**Generator and Verifier will have to keep secret** • CFG • Terminals a, b, c • Structured manner of generation of string.
Adversary is given $a\ b^2\ c$	Adversary is given $a^2\ b$
Adversary learns • b follows a, c follows b • Language contains a, b, c	Adversary learns • b follows a • Language contains a, b
Easy to guess the last terminal **$a^4 b$?**	Hard to guess the last terminal **$a^3 b$?**

Figure 4.0 Table showing the hardness of guessing the last 8 bits for structured and unstructured string generation.

5.3 Protocol making use of structured manner of generating strings with a Single Grammar.

A structured manner of generation of string can also be used for the design an authentication protocol making use of a single grammar. For this version of the protocol, the generator and the verifier will share a secret grammar and the knowledge of how to structure the generation of strings. One of the ways to generate the string in a structured manner is to introduce "n" new terminals belonging to the language at each instant. If the number of terminals, the grammar and the structured manner of generation of strings is kept as a secret, then given all the previously generated strings to an adversary, at each instant he will still have to make a random guess about the 8n bits which results in 2^{8n} combinations. It can be seen that as the number of new terminals "*n*" increases, the number of combinations increases exponentially. If *k* is the total number of terminals then it can be seen that the grammar will have to be replaced after *k/n* strings are generated.

All the previously generated strings do not contain the new valid terminals introduced in subsequent strings. Generator and the verifier both share the knowledge of structured manner of the generation of the string. So at each instant the generator and verifier will share the knowledge about the occurrence of terminals in a string. So not only it makes it hard for an adversary to guess the terminal but also the knowledge of occurrence of particular strings at an instant can be used to prevent replay attacks using previously generated strings.

The context free nature of the language can also be used to generate the strings in a structured manner. From [8] it can be inferred that if the strings is in the $A \rightarrow aAb$, $A \rightarrow \lambda$ is present; it is easy to infer the pattern. However for the language, $A \rightarrow aAb \mid cAd \mid eAf$, it becomes difficult to infer a pattern. For a "structured" manner to generate the strings the context free property can again be used to prevent the easy guessing of the string. Explaining it with an example for a language L_1: $A \rightarrow aAb$, $A \rightarrow \lambda$, by using unstructured manner to generate the strings given aaa bbb, aabb, ab, it becomes easier for an adversary to guess the next string. For a language L_2 : $A \rightarrow aAb \mid cAd \mid eAf$, $A \rightarrow \lambda$, which has more sequential embedding as compared to the language L_1, when generating a string in an *structured* manner, adversary will witness only acbd, aebf, eabf . Thus, it becomes hard for an adversary to guess the next string.

6.0 Prevention of Replay attack

One way to prevent the replay attack is to make sure that once a string in a grammar is used, it must be recorded to prevent its usage for the second time. However storing these strings and checking the inputs may require considerable resources. An efficient way to handle this problem is to use a Bloom filter [5]. For each grammar the generator and the verifier maintain identical bloom filters. A Bloom filter can be implemented as a vector of bits, all initially set to zero. When a string is used, it is hashed *k* times using independent hash operations, and the corresponding bit in the filter is set to one. To check whether a string has been seen before, one merely hashes the string *k* times to make sure each bit is already set.

Thus, when a generator creates a string, it is hashed *k* times with independent hash operations. These *k* operations each point to a bit in the vector, which is set to one. Thus, for the grammar *G,* if the string S, we hash the string, and set the corresponding bit entry in the filter *F*.

$F [H_1(l)] = 1$
$F [H_2(l)] = 1$
..
$F [H_k (l)] = 1$

To check whether a string has been used, a similar operation is used. One merely checks that all of the corresponding *k* bits in the filter are set. In such a case, the string would be rejected, since it can only be used once. Rules in the string are called again to generate the string.

With any Bloom filter, there's no chance of a false negative. However, there is a possibility of a false positive, particularly when the filter is near ``full" and many bits have been set. With *k* hash operations and n strings already in the hash table, sized *m*, the probability that any particular bit remains zero is:
$(1 - 1/m)^{km}$

Thus, the chance of a false positive can be approximated as:

$$(1 - (1 - 1/m)^{km})^k \approx (1 - e ({}^{kn/m)})^k$$

Since a false positive in the Bloom filter leads to the rejection of a valid string, we can think of this as a false negative for the grammar making use of the filter. The proposed implementation of a grammar-based authentication scheme therefore has a false negative (incorrectly rejecting a valid string) and a false positive rate (having an adversary guess a string). As before, we merely have to adjust parameters to make the risk acceptably low.

Key parameters for Bloom filter operations include table size, *m,* and the number of hashes, *k.* One could also reduce the number items hashed into the filter, *n* but for efficiency reasons, we don't want to restrict this. That is, *n* is effectively the number of strings in a grammar *G*, and we may wish this to be large. So, one merely has to expand the size of the bit table *m*, in relation to *k* and *n* to make the false positive rate acceptable.

7.0 Authentication protocols using distinct CFG.

In the earlier version of the protocol strings are being used for authentication. For this version of the protocol we present the exchange of context free grammars for authentication between the generator and the verifier. We

present here the initial approach and the intuition behind the approach. This protocol will be investigated further in detail.

For this version of protocol, the generator and the verifier both share the same grammar $G = \bigcup_{i=1}^{n} G_i$. Each of the CFG's $G_1, G_2, \ldots, G_n$, which forms the grammar G, are called sub CFG's. Each of these sub context free grammars contains distinct terminals. The generator and verifier keep the grammar G as a secret. At each instant "*i*" the generator selects a sub grammar G_i, and sends it to the verifier. Once sent, the sub grammar G_i is deleted from the generator's end, the verifier receives G_i, validates it with the stored grammar at its end. Upon successful validation generator is authenticated.

Let us assume that replay of the previously used sub grammar is prohibited by the protocol. An adversary who is listening to the communication between the generator and the verifier will collect the sub grammars $G_1, G_2, \ldots, G_n$ where G_i is the sub grammar generated at i^{th} instant. From these collected set of sub grammars, generated between the first and the n^{th} instant an adversary will have to guess the grammar, which will be used at the $n+1^{th}$ instant. One way of guessing the output sub grammar at $n+1^{th}$ instant is to guess the grammar G. From the grammar G an adversary will try to guess the output at $n+1^{th}$ instant. The context free languages are closed under union, concatenation and Kleene Star [1]. The sub-grammars $G_1, G_2 \ldots\ldots G_n$ belong to the grammar G, hence an adversary can collect the positive and the negative instances of the grammar G. From these instances an adversary will try to learn the grammar G so that he can predict the output at $n+1^{th}$ instant. However it can be seen from the result of Kearns and Valiant that factoring RSA, recognizing quadratic residues and factoring Blum integers are reducible to weakly leaning CFG's. Since it is hard to learn a CFG, so it is hard to successfully guess the output at $n+1^{th}$ instant by learning the grammar G.

For the proposed protocol there can be other forms of attacks like concatenating the previously generated grammar and sending it, deleting some terminals from the previously generated grammar and sending it. These attacks can be prevented by ensuring that the grammar used at each particular instant contains a distinct terminal. Since this information of the terminal/(terminals) is not present in the previously generated grammars attacks like concatenation of the previously generated grammars or deleting some terminals from the previously generated grammars and sending to the verifier will not be successful.

This protocol to generate one-time authentication information can give only a fixed number of attempts. After all the grammars have been used, the generator and the verifier will have to use fresh set of grammars.

8.0 Conclusion and Focus of future work.

Generation of one-time authentication protocols has been an area of active research. Current authentication protocols make use of symmetric keys, public private keys and hash functions. Our research attempts to use context free grammars for the generation of one-time authentication information. We presented some of our early work in this direction, including three protocols:

- In the first version of the protocol mutually independent grammars are being used. For the protocol making use of mutually independent grammars the adversary is confined to guessing the string. As a part of future work more powerful adversary will be considered. The powerful adversary will be given set of previously generated strings and the next string with the last missing terminal / (terminals). He will have to successfully guess the last missing terminal.

  The current adversary in Section 4.2 is making a random guess. Instead of making a random guess an adversary can use other stronger guessing algorithms. As discussed in the section 2.0 some tests to guess a string were performed. For the worst case the acceptance probability was observed to be one percent. For the above-mentioned experiments strings were generated in an unstructured manner. These experiments show that a single grammar is not acceptable for a security application. However, mutually independent grammars can be used to make guessing hard. As a part of future work further rigorous experiments will be performed by using other more powerful guessing algorithms to estimate the guessing of next string, given a set of strings.

  To make it hard for a powerful adversary to guess the string, rules to generate strings are investigated. One of the structured manners of string generation is the introduction of new terminals at each instant. Future work will also explore the guessing probability of other algorithms. Input to these algorithms will be strings generated in a structured manner and unstructured manner.

- In the second version of the protocol, proposed in section 5.3 a single CFG is shared between generator and verifier, which make use of structured string generation. As shown in figure 6.0, given the information from 1 to n-1 instant, structured manner of generation of string makes it hard to guess the string at n^{th} instant. We believe that the structured manner of generating strings can increase the hardness of guessing the string since if the strings are generated in a structured manner, guessing algorithm will have not only have to guess the string belonging to a language but also to guess the order as well. We present here a simple case of structured manner of generation of strings. Again there can be many ways of generation of string in a structured manner. We present here one of the manner of the generation of string in a structured way where in at each instant "*new terminals*" are being introduced. It may happen that for the structured manner of generation of string that a string is generated and never used by him. Structured manner of generation of string will require that the information has to be shard with the verifier. Resynchronization process has not been discussed here due to the space limitations.
- For the third version of the protocol, instead of string, context free grammar is being exchanged. For this version of the protocol we present here some of our initial intuition behind the approach. Future work will involve further detailed investigation and validating the protocol.

We presented here some of our initial research directions. Many details have been omitted due to space limitations. Further analysis of each of these protocols along with performance results is left to future work. An algorithm to generate the context free grammar was proposed in [4]. Ongoing work is focused on finding and enforcing structured patterns on the rules generation such that it might be hard for a powerful adversary to guess the new terminals, which are introduced at each instant. This complexity can further be increased by mutually independent grammars. Mutually independent grammars provides the inherent advantage that even in the case that the adversary can learn a grammar, from a set of grammars, confidence factor will not be reduced to zero. Necessary modifications in the protocol discussed in the section 3.0 and to the algorithm [4] to generate context free grammar may be proposed based upon the *structured* string generation pattern. *Structured* manner of generating strings will require that the grammars should be disposed after certain number of strings is generated. Guessing probability of the more powerful guessing algorithms is also a focus of future research. Comparison of the CFG based authentication protocol with the existing protocol in different domain will be an interesting study.

Acknowledgement

We would like to thank Dr. Mostafa H. Ammar for his discussion about statistics.

References

[1] Lewis H.R., Papadimitriou C. H., "Elements of the Theory of Computation", Prentice – Hall, 1998.

[2] Kearns Michael and Valiant Leslie, "Cryptographic limitations on learning Boolean formulae and finite Automata", Journal of ACM, 41(1): 67 – 95, January 1994.

[3] Previously published result making use of one CFG.

[4] Published result for an algorithm to generate CFG which is hard to learn.

[5] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors", Communications of the ACM, 13(7):422-426, July 1970.

[6] Solomonoff R. J., "A method for discovering the grammars of phase structure language", Information processing, New York: UNESCO, 1959.

[7] Gold E Mark, "Language identification in a Limit", Information and Control, 10(5), pp 447 – 474, 1967.

[8] Knobe Bruce and Knobe Kathleen, “A method for inferring context free grammars”, Information Control 2(2), pp 129 – 146, 1976.

[9] Das Sreeupa, Giles C. Lee, Sun Guo- Zheng, “ Learning context free grammars : Capabilities and Limitations of a Recurrent Neural Networks with an External Stack Memory” , Fourteen Annual Conference of the Cognitive Science Society, Morgan Kaufmann, San Mateo, CA, P 791 – 795, 1992.

[10] Keller, B. and Lutz R., “Learning Stochastic Context free grammars from examples from corpora using a genetic algorithm”, in ICANNGA 97.

[11] Fu King-Sun and Booth Taylor R., “ Grammatical Inference: Introduction and survey”, Parts I and II, IEEE Transaction Systems, Man and Cybernetics, SMC- 5(1) and (4), pp. 95 – 111 and pp. 409-423, 1975.

[12]L. Pitt and M. Warmuth. Reductions among prediction problems: On the difficulty of predicting automata. In *3rd Conference on Structure in Complexity Theory*, pages 60-69, 1988

[13] Leslie Lamport , “ Password Authentication over Insecure Communication”, Communications of ACM, 24 (11), pp.770 – 772, November 1981.

[14] CERT Advisory CA – 1994- 01, Ongoing Network Monitoring Attacks, http://www.cert.org/advisories/CA-1994-01.html

Appendix

Several experiments were conducted to get a feel about the toughness of guessing a string from a given set of strings. Input to these guessing tests were the strings generated in a random manner. These tests were named *frontbreaking*, *backbreaking, allfrontbreakimg, allbackbreaking*. The *frontbreaking* test selects a string and breaks into different combinations such that the first terminal remains same. For example, different combinations of the string "abcdefg" by the *frontbreaking* rule are abc, abcd, abcde, and abcdf. The b*ackbreaking* rule selects a string and breaks it into different combinations such that the last terminal remains the same. Different combinations for the same example by back breaking rule are efg, defg,cdefg, bdefg. These strings were passed through the grammar. The *allfrontbreaking* type of rule involves collection of all the terminals, which start a string, followed by the selection of a string and parsing it to find out if any of the start terminals appears in it. In case of appearance of any of the start terminals at any position except at the starting of the string, the string is broken such that start symbol occupies the first position and the *frontbreaking* rule is applied. The *allbackbreaking* type of rule involves collection of all the terminals, which ends the strings, followed by the selection of a string and parsing it. In case of appearance of any of the end terminals at any position except at the end, string is broken such that the end terminal occupies the last position and the *backbreaking* rule is applied.

As a simple example, if there are two strings "abcdef" and "crafgd" we collect all the terminals which start the strings and all the terminals which end the string. For starting terminals we get "a" (starting terminals for "abcdef") and "c" (starting terminal for "crafgd"). A string is chosen and it is parsed to find out if any of the start symbols appear in it. In the current example "c" appears in "abcdef" so by the *allfrontbreaking* rule the strings "cdef" and "cde" are generated. Similarly all the end terminals are collected. For the given example the end terminals are "f" and "d". Since the end terminal "f" appears in "crafgd" strings "craf" and "raf" are generated using the *allbackbreaking* rule.

. The strings were generated by randomly calling the grammar's production rules. 674 strings of length ranging from 3 – 25 were generated. By applying *frontbreaking* rule, the strings resulted in 4902 combinations. Out of these 4902 combinations 390 strings were accepted by the grammar. So for strings of length between 3 – 25, around 90% of time it can be ensured that the strings generated by applying *frontbreaking* rule will not be accepted by the grammar. 956 strings of length between 3-150, resulted in 20925 combinations by *frontbreaking* rule, out of which only 398 strings got accepted by the grammar. This gives an acceptance rate of 1.9%. For each range, a different set of strings was generated by randomly expanding the start symbol and nonterminals. This means that the 674 strings generated for the length-range 3-25 are totally different from 956 strings generated for the length-range 3-150. As the string-length increases, the *front-breaking* rule results in increasing number of output combinations. And at the same time, the number of strings accepted by the grammar decreases. By applying *backbreaking* rule, 674 strings of length between 3- 25 resulted in 4902 combinations, out of which 195 got accepted by the grammar. 956 strings of length ranging from 3 – 150 resulted in 20925 combinations, out of which 212 got accepted. For the *backbreaking* rule, string-length ranging from 3 – 25 resulted in 3.9% acceptance and of size between 3 – 150 results in 1.013% acceptance by the grammar. By applying the *allfrontbreaking* rule, 724 strings of length between 3-25, resulted in 4250 combinations out of which 169 got accepted. 1203 strings of size ranging from 3 – 150 resulted in 19991 combinations by using *allfrontbreaking* rule. Out of 19991 strings, 213 got accepted giving an acceptance rate of 1.06%. *Allbackbreaking* rule for 724 strings of length ranging from 3 – 25 resulted in 3369 combinations out of which 125 strings got accepted by the grammar. This gives an acceptance rate of 3.7%. 1023 strings of length between 3- 150, resulted in 16232 combinations by *allbackbreaking* rule. Out of 16232 combinations, 209 strings got accepted by the grammar. This gives an acceptance rate of 1.28%. From these tests it can be concluded that given a set of strings it might be simple to guess the next string.