CSTEG: TALKING IN C CODE
Steganography of C Source Code in Text

Jorge Blasco Alís, Julio Cesar Hernandez-Castro, Juan M. E. Tapiador
and Arturo Ribagorda Garnacho
Computer Science Department, University Carlos III of Madrid, Av. Universidad 30, 28911, Leganés, Spain
{jbalis, jcesar, jestevez, arturo}@inf.uc3m.es

Keywords: Export restrictions, Steganography, Information Hiding.

Abstract: Cryptographic software has suffered in many ocassions from export restrictions. Governments might claim that cryptographic algorithms are equivalent to military equipment to justify and maintain these restrictions. Sometimes, these laws are approved under dictatorial rules or even by democratric goverments which exploit and overstimate a terrorist menace to restrict civil rights. Citizens have evaded these restrictions in many ways: handwriting the program's source code and then typing it again, printing the source code in a t-shirt, using some kind of steganographic technique, etc. In this paper, we present a system called CSteg that hides source code into plain text by using context-free grammars. This presents the additional advantage that under some laws plain text is protected (and its exportation allowed) by free-speech and/or intellectual property legislation.

1 INTRODUCTION

Governments have frequently restricted the export and even the use of cryptographic software and source code by civilians. Use of cryptography in military environments has made it a weapon, instead of a tool to protect confidentiality. During the 90s these restrictions were applied when some civilians tried to export source code of cryptographic tools outside the United States (R.Karn, 1994; Bernstein, 1992). All these people found that the United States government denied the required free export application.

As a justification, the government claimed that the electronic version of the code was easy to compile to produce a fully working program. After years of trials, in 2000 the Sixth Circuit Court sentenced that electronic source code was protected by freedom of speech.

In 1995 a group of 40 countries, signed the Wassenaar Agreement[1]. This regulates the transfer of weapons, and dual-use goods and technologies. Cryptography was included as a dual-use technology.

Now, most of the cryptographic software can be obtained without export license, but these laws could be used in the future to restrict its free distribution.

With this work we try to offer a suitable method to export source code evading any restriction further those that applied to printed literature or speech. Export of some kinds of source code may be restricted by law, so we will have to transform the source code into an exportable object. It is obvious that we cannot use cryptography to solve this problem. In this scenario, steganography (not included in Wassenar's agreement or in most export laws) becomes the best solution, as stego-objects have the property of going unnoticed if properly created.

The remainder of this document is structured as follows. Section 2 describes the basics of steganography. Section 3 briefly presents the solution proposed to solve the problem. Section 4 describes the experiments carried out to verify the proposed solution. Section 5 shows and analyzes the results obtained in the experiments. Section 6 discusses the conclusions gathered in this work.

2 STEGANOGRAPHY

First well-documented usage of steganography was made by the Greeks. On 480 B.C, Demaratus wanted to warn the Spartans against an invasion by Xerxes, the Persian leader. Demaratus sent a message written on a wooden table covered by wax, so it could pass all the guard controls and arrive to Sparta.

Since those days, steganography has developed as a science, and many different approaches have been used to cover contents of any kind. Image Steganography (Neil F. Johnson, 1998) is one of the most used

[1]http://www.wassenaar.com

techniques. Most simple techniques hide information on the least significant bits (LSB) of each pixel. Another widely used cover are digital audio files. Audio steganography also includes techniques such as LSB (similar to image LSB steganography). Audio steganography can be performed also in compressed audio files like MP3s. Some tools like MP3Stego (Petitcolas, 2006) can hide information during the *inner loop* step, by modifiying the DCT values.

More steganographic techniques can be found in the literature, including subliminal channels (Simmons, 1996), SMS (Mohammad Shirali-Shahreza, 2007), TCP/IP packets (Murdoch and Lewis, 2005), executable files (El-Khalil, 2003) and games (Castro et al., 2006).

3 HIDING SOURCE CODE

We have developed a stego-system (*Csteg*) based on context-free grammars. Our design allows transforming a source code file into a plaintext file (stego-text). A grammar describing the source code structure is used to produce the stego-texts. The stego-text generated has no export restrictions [2]. Stego-text can be recovered at its destiny applying the reverse grammar. Recovered source code keeps all the functionality of the original source code.

Other text steganography systems based on context-free grammars have been developed in the past: *Spammimic*[3] can convert a short text message into an email Spam message. Another tool is *C to English to C* (Schwarz, 2001) which translates C source code to the English explanation of it. In 2001, Marttila (Marttila, 2001) conceived a system to hide source code inside a text.
Martilla designed a tool called *c2txt2c* that, by using context-free grammars, was able to produce an English text from the source code of the *Blowfish* cipher. Martilla's *c2txt2c* is very limited as it was not able to hide another cryptographic algorithm except for *Blowfish*. We have continued Marttilia's research and designed and implemented a tool that hides consistently any source code into plain text. On the other hand, using context-free grammars to produce stego-text has its drawbacks. Producing meaningful texts is difficult and, additionally, the used grammar should be able to parse any source code provided. Finally, the use of the same grammar to hide all the input files may generate similar stego-texts which might ease an attack. To improve the security of the system it could be advantegeous to have the possibility of generating very different stego-texts, even from the same source file.

[2]at least in the reviewed legislations

[3]http://www.spammimic.com

Our system uses a plain text file to produce context-free grammars. The aim is to be able to produce different grammars just by changing the input file. This will also help to build meaningful stego-texts. These plaintext files should have special characteristics.

- The content of the text file should have sense and meaning.
- Text file should have the maximum possible length. Our system will extract portions of the file to generate the grammar.
- Files used by the system should not be restricted by any copyright.

To perform the recovery process, our stego-system will need the stego-text and the same input text file that was used to hide the source code (Figure 1). In this case a reverse grammar will be generated, producing the original source code as output. The text file used to hide and restore the source code can be considered as the key of the stego-system, as it is needed to restore the source code and a different file will produce different stego-texts.

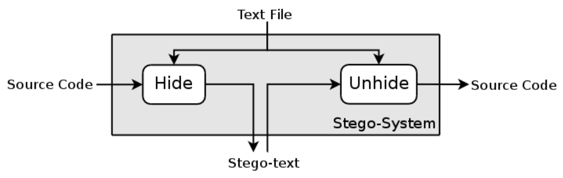

Figure 1: Stego-system scheme.

3.1 Grammar Generation

Grammar generation can be divided in two different steps. First step builds the grammar to hide the source code. Second step builds the grammar to restore the source code (recovery grammar). Both grammars must be generated with the same cover-text as input.

3.1.1 Creating the Hiding Grammar

The first step in creating the hiding grammar is to read the source code files. The resulting grammar is closely related to the programming language to be hidden. The cover-text file is used to generate the output of the parsing process. Output for each rule (P) of the grammar (G) is extracted from the cover-text file (Figure 2). To avoid conflicts in the recovery process, fragments (f_i) used to produce output cannot be repeated. Each of the fragments can be attached to a terminal symbol (t_i) so it will be its output in the stego-text. Nevertheless, is not necessary to have only one fragment per terminal symbol. A non terminal (S, A, B) derivation with just one terminal symbol may use more than one text fragment to produce the output. In this case, the output for that derivation will be the concatenation of these fragments.

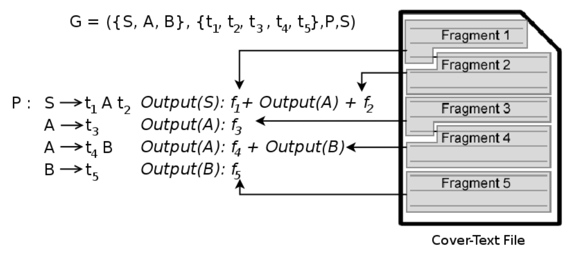
G = ({S, A, B}, {t1, t2, t3, t4, t5},P,S)
P : S → t1 A t2 Output(S): f1 + Output(A) + f2
A → t3 Output(A): f3
A → t4 B Output(A): f4 + Output(B)
B → t5 Output(B): f5
Fragment 1
Fragment 2
Fragment 3
Fragment 4
Fragment 5
Cover-Text File

Figure 2: Hiding grammar construction.

Programming languages have a high redundancy. This can be exploited to mount a steganographic attack based on frequency analysis. To reduce the feasibility of these attacks, some rules of the grammar may have different number of outputs from the text file. This output should be selected randomly between the random fragments selected from the cover-text file (Figure3).

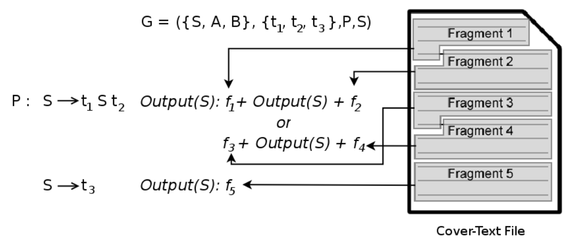

Figure 3: Random fragments in hiding grammar.

Programming languages also have specific kinds of tokens, like identifiers and literals. A specific hiding strategy must be used for each. Strategies used in *Csteg* are described in section 4.1.

3.1.2 Creating the Recovery Grammar

This grammar is able to parse stego-text files to produce source code files. In this case, fragments extracted from the plain text file are used as terminal symbols of the grammar. The programming language grammar is used as a template of the recovery grammar. Terminal symbols in the programming language grammar are substituted by fragments of the cover-text file. The output produced by each of the grammar rules includes the original programming language terminal symbols.

One derivation can have more than one output. This output is selected randomly between the group of outputs for that derivation. In the case of the recovery grammar, this translates to multiple derivations producing the same output. Each of the derivations parses one of the different outputs produced by the stego-text grammar.

Hiding methods for special tokens (identifiers, etc.) have their special recovery mechanism included inside the description of the recovery grammar.

3.1.3 Specific Programming Languages Issues

A context-free grammar describing the programming language is not enough to build stego-text files. Source code is usually split into different files. Some programming languages, like C and C++, may include preprocessor directives in their code. All these must be processed before the parsing process starts. Decisions taken to solve this issue in *Csteg* are described in section 4.

3.2 Stego-text Generation and Recovery Process

The stego-text is generated following a typical compiler/parser process. Source code is read and a derivation tree is built. Each of the activated derivations generates its output which includes fragments of the cover-text file. The output is concatenated using the derivation tree. The file produced is the stego-text file.

The cover-text file works as the key of the stego-system. This file is necessary to recover the original source code. The recovery grammar parses the stego-text file to produce the source code files. Generated source code files have the same functionality of the original source code.

4 EXPERIMENTS

The system previously described has been tested with an implementation that hides ANSI C source code. This implementation has been called *Csteg*. The stego-system has been implemented using Ruby and C. Parsers are produced through Flex [4] and Bison [5] parser generators. Cover-text files have been retrieved from Project Gutenberg[6]. Both grammars share non-terminal symbols. Terminal symbols are different in each grammar. The C grammar includes the common

[4]http://flex.sourceforge.net

[5]http://www.gnu.org/software/bison/

[6]http://www.gutenberg.org

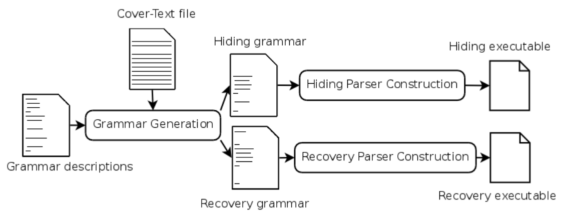

Figure 4: Parser generation process.

terminal symbols from the C programming language (*if, else, identifiers, literals,* etc.). Terminal symbols for the stego-text grammar are extracted from the Project Gutenberg's book used as cover-text. In *Csteg*, terminal symbols of the stego-text grammar can be words, phrases or paragraphs of the cover-text. Parser generation process is described in Figure 4.

4.1 Extraction of Stego-text Terminal Symbols

This phase extracts terminal symbols from the cover-text file provided. Terminals are extracted from Project Gutenberg cover-texts and attached to the stego-text grammar. The text file used to generate the stego-text grammar should be able to produce at least one terminal for each terminal in the grammar. As in other programming languages, C has some structures that may be more used than others. The frequency of appearance of this structures may lead to a specific frequency appearance of cover-text terminal symbols.

To complicate frequency-based steganalysis, we have added random stego-text derivations for the most used C structures. We have analyzed the mostly used structures and tokens of a group of C source code files with cryptographic purposes. Files have been gathered from different sources including eStream (ECRYPT, 2008), Applied Cryptography (Schneier, 1996) and an implementation of the Anubis cipher (Vincent Rijmen, 2008).

Table 1: Frequency of C tokens in cryptographic software.

Token type	Appearance in %
Punctuator	51.59
Identifier	30.02
Numerical literal	11.63
Reserved word	4.77
String literal	1.29
Preprocessor directive	0.7

Measures have been made with tools taken from (Jones, 2003). Comments have not been accounted. Frequency distribution of C tokens gathered in our tests is described in Table 1.

Table 2: Freq. of punctuator tokens in analyzed software.

Token	Frequency	Token	Frequency
,	21.52	->	2.05
;	13.21	.	1.82
(	12	*	1.73
)	12	|	1.34
=	5.41	#	1.19
]	4.8	v++	1.11
[	4.8	+	1
{	2.21	*v	0.92
}	2.21	Other	11.68

Table 3: Freq. of reserved words in cryptographic software.

Word	Frequency	Word	Frequency
if	14.84	static	2.93
int	13.79	register	2.84
unsigned	9.25	case	2.83
char	8.84	while	2.60
for	8.30	break	2.54
void	5.85	sizeof	1.54
else	5.09	extern	1.21
return	5.02	short	1.14
long	3.74	struct	0.98
const	3.49	Other	3.15

Most used punctuator tokens are described in Table 2. Reserved words frequency is more homogeneous (Table 3).

We have found that inside each group of possible tokens (punctuators and reserved words) there are only a few tokens which are commonly used. The rest of the tokens are used rarely compared with the common tokens (Figure 5).

To reduce the difference in the redundancy of fragments linked to most used programming language tokens, we have introduced random derivations. Each time a symbol from this group is derived, his output will be chosen randomly from a group of cover-text fragments.

Table 4: Random derivations per punctuator.

#	Additions	Gap	Tokens
1	20	100%-14%	,
2	14	14%-6%	; ()
3	4	6%-2%	= [ ] { } ->
4	1	2%-0%	Other

Most used tokens in each of the analyzed sets have been grouped defining boundaries on their frequency (Tables 2 and 3). For each group, a number of random cover-text derivations have been added. The number of derivations added in a group is directly related with

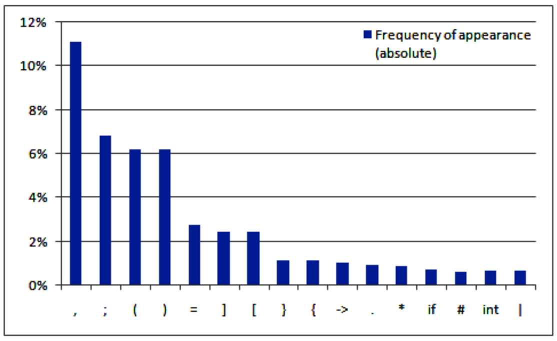

Figure 5: Most used tokens.

boundaries selected for that group. Four groups have been created for each set (punctuator tokens and reserved words). The highest appearance group will correspond to the one with more random derivations. The last group will not have any random derivation on it. Groups deduced for punctuators are shown in Table 4.

The same procedure has been used for reserved words. The results are shown in Table 5.

Table 5: Random cover-text derivations per reserved word.

#	Additions	Gap	Tokens
1	10	100%-9%	if,int,unsigned
2	8	9%-5%	char,for,void, else,return
3	3	4%-1%	long,const,case,short static,register,extern while,break,sizeof
4	1	1%-0%	Other

There are elements inside the source code that can not be hidden with substitutions from the cover-text. These elements have been hidden by using special techniques.

- Identifiers are replaced by names. Each time an identifier is found, a symbol table is looked up. If the identifier is not in the symbol table, it is stored on it and associated with an English name. The output in this case is the identifier concatenated with the associated name. If the identifier is in the symbol table, nothing is added to the table and the output produced is just the English name associated with that identifier.

- A String literal can use any kind of character. We have implemented a simple Caesar cipher to hide the content of this kind of literals. The ciphered string will be imperceptible inside the stego-text.

- Numerical literals that appear in the code are translated into "poems" with a substitution algorithm. The digits of the literal are substituted by poem words. Each of the different digits has ten different words to choose randomly for its substitution.

4.2 Parser Generation

Terminal symbols extracted from the cover-text must be introduced into the grammar description file. We have used Flex and Bison to produce our parsers. Each time an operation is performed only one parser is generated. A hide operation will generate a C parser producing a stego-text as output. A recovery operation will generate a stego-text parser that produces C source code as output. To produce the parsers the source code generated by Flex and Bison is compiled.

4.3 Covering Process

The C parser is able to parse any kind of C source code. The output of this parser is the stego-text. C preprocessor directives have not been hidden. Source code files that are going to be hidden have to pass through a C preprocessor.

Our preprocessor treats the file inclusion directives differently than an usual C preprocessor. When the preprocessor finds an *#include* directive it checks if the file included is a system library (with <>) or a user library (with " "). System libraries are not included in the auxiliary file. If system libraries were included the lines of code to be hidden would significantly increase. Reference to system libraries is hidden like a string literal. On the other hand, user libraries source code is included in the auxiliary file.

The use of a C preprocessor means that most of the preprocessor directives included in the code are lost in the recovery process. The hiding process will make that the recovered source code will not be exactly the same source code that was hidden, but it will have exactly the same functionality.

4.4 Recovering Process

Stego-text parser reads the stego-text files and generates C source code files.

In order to regain the functionality of the original source code, a post processing is needed. If the preprocessed file was the result of some file inclusions, the source code files which were included are created. This restores the original source code file structure.

All experiments have been performed over cryptographic C source code files. Twelve experiments have been performed. Each consisted on the concealment and the recovery of the source code of a cryptographic algorithm. Four different algorithms have been chosen: Anubis, 3DES, IDEA and DECIM. Each of the algorithms has been hidden using three different types of terminal symbols from the cover-text file (words, phrases and paragraphs). Different books from Project Gutenberg's repositories have been used as cover-texts.

5 RESULTS

The size of stego-texts generated (Figure 6) strongly depends on the kind of terminal symbol selected. Stego-texts generated with words from the cover-text are naturally smaller than stego-texts generated with paragraphs. Obviously, the average size of the terminal symbols in the second case is bigger. The hiding process has removed all preprocessor directives except file inclusions, which remain unchanged. Comments have been lost.

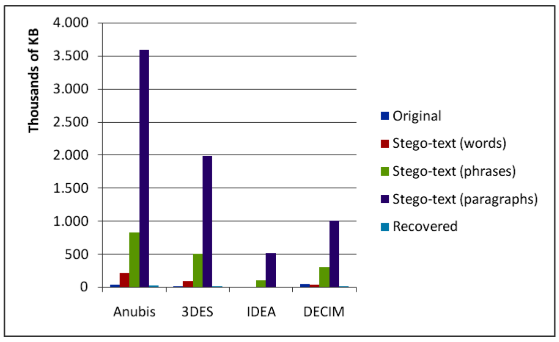

Figure 6: File size chart.

We have computed the number of bytes of the stego-object produced per byte of hidden information (Table 6). This can be used as a redundancy estimation. To perform this calculation we have divided the size of the stego-object by the size of the recovered source code. Our stego-system, as expected, has introduced a lot of redundancy into the stego-object.

Table 6: Bytes of stego-text per byte of source code.

	Anubis	3Des	Idea	Decim
Words	8.32	6.83	5.49	4.32
Phrases	31.42	36.95	66.64	32.72
Paragraphs	136.45	147.08	338.56	108.508

Stego-text compression ratio (using *bzip2*) compared with original and recovered source code files (Figure 7) indicates again, that stego-text files have a considerable redundancy. Stego-text files generated with words have much less redundancy than those generated with phrases and paragraphs. Differences on the compression ratio between phrases and paragraphs are not significant. A redundancy check by an attacker of the stego-text would rise suspicions. This security drawback could be reduced by adding more random stego-text derivations. In the ideal situation our stego-text would not have any repetition. In this case, the compression ratio would not be the same as the source code but it would be less suspicious.Word generated stego-texts have the lower compression ratio, but they only are a conjunction of generally meaningless words. Due to the minimum differences between phrases and paragraphs stego-texts, it should be a better choice the use of paragraphs to build the stego-texts.

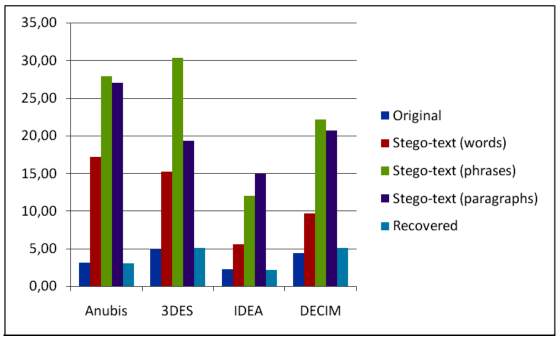

Figure 7: Compression ratio chart.

Stego-text generated should comply with the frequency distribution of characters in English. We have measured the frequency of appearance of characters and digraphs in all stego-texts generated. Unfortunately, distribution of letters in texts depends on the length, language, and the text itself.

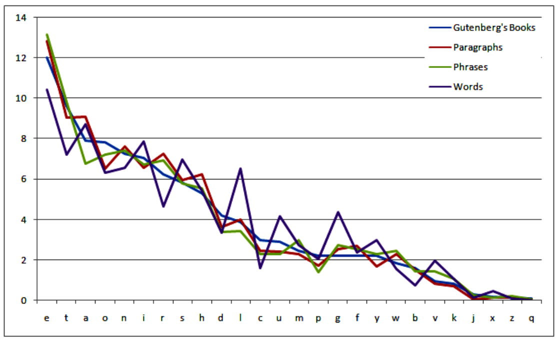

Figure 8: Comparison of character distribution.

Frequency of appearance of characters usually is similar between text of the same language and length, but it can be easily manipulated, as in the case of lipograms[7]. Aside from these rare cases we have build

[7]A lipogram is a text which does not use a specific letter.

frequency distributions for characters and digraphs based on 235 different books from Project Gutenberg. This frequency distribution will give a good approximation of the characteristic frequency distribution for texts in English.

Frequency distribution of characters (Figure 8) in stego-texts generated with words from a Project Gutenberg book differs a lot from the reference distribution. Stego-texts with paragraphs and phrases are much closer to the reference distribution. Digraphs frequency (Figure 9) is consistent with this observation. Stego-texts generated with phrases and paragraphs fit better the reference digraph distribution.

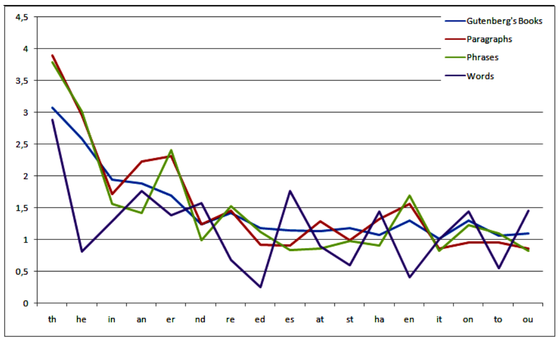

Figure 9: Comparison of digrams distribution.

Finally, we have compared the main propierties of *CSteg* against other text steganography tools described in this paper (Table 7).

Table 7: Text steganography tools comparision.

	Input	Variability	Key
CSteg	C source.	Yes	Yes
c2txt2c	Blowfish.	No	No
C to English to C	C source	No	No
Spammimic	Text	No	Yes

6 CONCLUSIONS AND FUTURE WORK

Our system can hide any C files independently of its length. As the size of the embedded data increases, the redundancy of the stego-text also increases, reducing the security of the system against redundancy-based steganalysis. Our Stego-texts carry the same functionality of the original source code files.

A notable example is "La Disparition" (Perec, 1969). None of the 300 pages contained the letter "e".

Stego-texts generated can avoid cryptographic export restrictions. Our system is able to produce meaningful stego-texts. A human can find common structures of the human language. This is an important issue that will help the stego-text to pass unnoticed by a passive attacker. On the other hand, grammar parsers are very sensitive to changes. A change in the stego-text by an active attacker will probably cause the embedded data to be lost.

Future lines of work will try to improve the most important weakness of the system. Robustness of the system should be improved. This could be achieved by means of a synonyms system. Redundancy may be reduced adding random derivations. Others lines of work may include the use of this kind of stego-system to hide any kind of information, not only source code.

The software resulting from this research (*Csteg*) has been uploaded to SourceForge.net[8].

REFERENCES

Bernstein, D. (1992). Bernstein case web page. http://cr.yp.to/export.html.

Castro, J. C. H., Lopez, I. B., Tapiador, J. M. E., and Garnacho, A. R. (2006). Steganography in games. *Computers and Security*, 25(1):64–71.

ECRYPT (2008). eSTREAM Project. http://www.ecrypt.eu.org/stream/.

El-Khalil, R. (2003). Hydan. http://crazyboy.com/hydan/.

Jones, D. M. (2003). The New C Standard: A Cultural and Economic Commentary.

Marttila, L. (2001). Accurate language to inaccurate language (and back) translator, c2txt2c v0.2.1. http://www.verkkotieto.fi/∼lm/c2txt2c/.

Mohammad Shirali-Shahreza, M. H. S.-S. (2007). Text steganography in sms. *Int. Conference on Convergence Information Technology*, pages 2260–2265.

Murdoch, S. J. and Lewis, S. (2005). Embedding covert channels into tcp/ip. *Information Hiding*, pages 247–261.

Neil F. Johnson, S. J. (1998). Exploring steganography. *IEEE Computer*, 31(2):26–34.

Perec, G. (1969). *La Disparition*. Gallimard, Paris.

Petitcolas, F. A. P. (2006). MP3Stego. http://www.petitcolas.net/fabien/steganography.

R.Karn, P. (1994). Applied cryptography case web page. Web. http://people.qualcomm.com/karn/export/history.html.

Schneier, B. (1996). *Applied Cryptography*. John Wiley and Sons, 2nd edition.

Schwarz, O. (2001). C to English to C. http://www.mit.edu/∼ocschwar/.

[8]http://www.sourceforge.net

Simmons, G. J. (1996). The history of subliminal channels. In *Proceedings of the First International Workshop on Information Hiding*, pages 237–256.

Vincent Rijmen, P. S. L. M. B. (2008). The Anubis Block Cipher. http://paginas.terra.com.br/ informatica/paulobarreto.