Steganalysis of Hydan

Jorge Blasco, Julio C. Hernandez-Castro, Juan M.E. Tapiador, Arturo Ribagorda and Miguel A. Orellana-Quiros

Abstract *Hydan* is a steganographic tool which can be used to hide any kind of information inside executable files. In this work, we present an efficient distinguisher for it: We have developed a system that is able to detect executable files with embedded information through *Hydan*. Our system uses statistical analysis of instruction set distribution to distinguish between files with no hidden information and files that have been modified with *Hydan*. We have tested our algorithm against a mix of *clean* and stego-executable files. The proposed distinguisher is able to tell apart these files with a 0 ratio of false positives and negatives, thus detecting all files with hidden information through *Hydan*.

1 Introduction

Steganography is the art and science that tries to hide the existence of messages [4]. The objectives of steganography are not the same that those of cryptography, which main aim is to conceal the message contents by performing different transformations so only authorized persons can read it. At first, one may think that cryptography is enough to ensure the security of the communications between two parties, but there are scenarios where the knowledge of the existence of a communication between two parties may be critical. These scenarios all have something in common with that described by Simmons and known as *the Prisoners problem* [12]. In this, two prisoners (Alice and Bob) want to plot an escape plan. As they are not in the same cell they must communicate through a warden (Willie). If Willie ever suspects that Alice and Bob are planning to escape or are engaging in any kind of secret com-

Jorge Blasco · Julio C. Hernandez-Castro · Juan M.E. Tapiador · Arturo Ribagorda
Carlos III University of Madrid, Av. de la Universidad 30, 28911 Leganés, e-mail: jbalis@inf.uc3m.es, jcesar@inf.uc3m.es, jestevez@inf.uc3m.es, arturo@inf.uc3m.es

Miguel A. Orellana-Quiros
Ministry of Economy, Cl. Alcala,5, 28071 Madrid e-mail: mangel.orellana@meh.es

munication he will put them into isolation cells. In this scenario, Alice and Bob can not simply use cryptography because Willie will recognize encrypted messages and infer they are communicating secretly, so he will stop this channel. Alice and Bob should hide their messages into seemingly innocuous ones, so Willie will not notice the covert communication. Additionally, Willie can behave in different ways: If Willie just checks the messages and forwards them to its recipient, then Willie is a *passive warden*. On the other hand, if Willie has high suspicions of Alice and Bob planning an escape, but he does not have a proof, it is possible that he will modify slightly the message contents trying to perturb any hidden information. In this case, Willie is an *active warden*. Both possible scenarios must be considered when designing stego-systems, so the quality of a stego-system can be measured (in addittion to other properties) by means of the difficulty to detect its content and the possibility that hidden information is not lost even if the stego-object suffers some modifications.

The first documented use of steganography [5] was made by *Demaratus*, who wanted to warn the Greeks about a Persian invasion leaded by *Xerxes*. *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 [9]. Image Steganography [4] is one of the most used techniques. Covering contents into images can be done in many different ways. Most simple techniques hide information on the least significant bits (LSB) of each pixel. Other techniques use image compression algorithms. For example, the JPEG image compression algorithm is based on the parameters of the discrete cosine transform (DCT). Using different parameters in the DCT calculation allows hiding information in the image file. Another widely used cover are digital audio files. Audio steganography also includes techniques such as LSB (similar to image LSB steganography).

Changing the last significant bit on each audio sample produces slight modifications on audio files that can not generally be distinguished by humans, specially if the redundancy ratio is high. Audio steganography can be performed also in compressed audio files like MP3s. Some tools like MP3Stego [10] can hide information during the *inner loop* step, by modifying the DCT values. Much more steganographic techniques can be found in the literature such as subliminal channels [12], SMS [11], TCP/IP [6] and games [3].

All security requirements for cryptographic systems are usually (or should be) applied to steganographic systems. This means that the security of a steganographic algorithm should not rely itself on the secrecy of the algorithm, which should be public, but on the knowledge of the key. In steganography, it should not be possible to distinguish a *clean* object from a stego-object if the key is unknown. In this work, we prove that it is possible to distinguish a *clean* executable file from a stego-object created through *Hydan* without the possession of the key. The remainder of this document is structured as follows. Section 2 introduces previous work done in executable files steganography. Section 3 describes the basics of *Hydan* and how it works. Section 4 shows the steganalysis performed on *Hydan* and the resulting distinguisher. This section also performs a discussion on possible ways to overcome the steganalysis presented. Section 6 presents the gathered conclusions and possible lines of future work.

2 Previous Work

Hydan [2] is the first documented tool and scheme that uses directly executable files as a cover. During years, other techniques have been used to insert hidden information into source files, but for copyright protection purposes only. These involve access to source code, where programmers insert copyright marks and integrity checks right inside their code. Information inserted in this way can be used to prove the integrity and authorship of the program [13]. Outside *Hydan*, other authors [1] have later described different techniques to introduce information in executable files. Authors describe four different techniques. *Instruction Selection* replaces some of the instructions in the executable file for others with the same functionality. *Register Allocation* encodes embedded information in changes on the registers used by some instructions. *Instruction Scheduling* changes the order of non-dependant instructions. Finally, *Code Layout* uses the order of big blocks.

Authors have implemented all the proposed techniques in a more advanced tool called *Stilo*. A steganalysis of *Stilo* is proposed in the same paper based on a concept named *Code Transformation Signature*, which is defined as the set of characteristics that can be used to detect the presence of hidden information into *Stilo* executable files. Authors describe the *Code Transformation Signatures* for *Stilo* and propose a group of countermeasures to avoid them. Authors also mention *Hydan*, but they do not perform any steganalysis nor reveal the corresponding *Code Transformation Signatures* for *Hydan*. Apart from this work, no other techniques have been proposed to hide information on executable files. In this paper we describe the main properties (its *Code Transformation Signatures*) that can be used to detect executable-files with hidden information through *Hydan*. Based on those properties, a very efficient distinguisher is proposed.

3 Hydan

Hydan is a steganographic tool which covers messages in executable files. It does not change the functionality of the executable neither the size of it. A detailed description on how *Hydan* works can be found on [2].

Hydan uses the “redundancy” on the instructions sets of executable files to introduce hidden information. Specifically, *Hydan* uses the concept of *functionality-equivalent instructions*. A set of *functionality-equivalent instructions* is a group of instructions in which any instruction of the group can be replaced for other without loss of functionality. For example, to add a certain amount to a specific register it is possible to use *add, r1, 8* or , equivalently, use *sub, r1, -8*. In this case, the *add* instruction could encode the bit value 0, and the *sub* instruction may encode the bit value 1. Depending on the size of the *functionality-equivalent instructions* sets it is possible to encode more than one bit with one instruction. A set of four *functionality-equivalent instructions* would allow codifying 2 bits (00, 01, 10 and 11). Generally, with a set of n equivalent instructions it would be possible to encode $\lfloor \log_2(n) \rfloor$ bits. Table 1 describes the *functionality-equivalent instructions* groups and number of instructions in each of the groups for the *x86* set, which is the most common and the one used by *Hydan*.

Table 1 Groups of *functionality-equivalent instructions* used in *Hydan*

Group	Inst.	Group	Inst.	Group	Inst.
toac8	5	*toac32*	5	*rrcmp8*	2
rrcmp32	2	*toasxc8*	7	*toasxc32*	6
addsub8	2	*addsub8-2*	2	*addsub32-1*	2
addsub32-2	2	*addsub32-3*	2	*xorsub8*	4
xorsub32	4	*add8*	2	*add32*	2
adc8	2	*adc32*	2	*and8*	2
cmp8	2	*cmp32*	2	*mov8*	2
mov32	2	*or8*	2	*or32*	2
sbb8	2	*sbb32*	2	*sub8*	2
sub32	2	*xor8*	2	*xor32*	2
and32	2				

Embedding process of *Hydan* is done in two steps. First step encrypts the message to be hidden using *AES* or *Blowfish* with the password given by the user. In the second step, the encrypted message is embedded into the executable file. Specifically, *Hydan* works as follows: Once the message has been encrypted, *Hydan* searches for possible places to introduce information. Then, *Hydan* generates a random number seeded with the password entered by the user. This number is used to select which of the selected places of the executable file will be used to hide the information. With this mechanism, the password will be needed to recover the data and different passwords will lead to different placements of the embedded information. Recovery process first extracts the encrypted message from the executable file. Then, the message is decrypted using the provided password.

With *Hydan*, it is possible to embed (on average) 1 bit of information per 110 bits of executable code. In fact, it is possible to embed different ratios of information, but El-Khalil proposed the specified one as the better trade-off between security and capacity [2].

Hydan changes perceptibly the content of the executable files with hidden information. Therefore, if these changes lead to a specific signature, it is possible to build a system that is able to distinguish a *Hydan* executable file from any other executable file. This signature may show in many different ways. Next section discusses the possible methods to detect a *Hydan* modified executable and proposes a very efficient distinguisher to detect a *Hydan* covert-channel.

4 Steganalysis of Hydan

Changes introduced by *Hydan* into assembler code can modify different properties of the original executable file. *Hydan* does not change the size of the stego-object, but it changes the code itself. If the original program is available it will be possible to check through integrity checks (CRCs [8], hash functions [7], etc.) if the executable file has been modified, but these are not proof of embedded information. Other properties such as execution time, flag activation and copyright marks checks, can prove that executable code has been modified, but will not be proof of embedded information.

Most compilers often produce similar sets of instructions. Thus, if a compiler has to select between two instructions with the same functionality it will usually select the same instruction. This property of most compilers allows building a profile of *clean* applications based on the probability distribution of instructions inside *clean* programs. Changes made by *Hydan* may lead to another probability distribution of instructions. If these changes can be profiled and generalized, it would be possible to detect if an executable file has hidden information. Steganalysis performed on this paper is based on this approach.

We have built a distinguisher that is able to detect executable files with embedded information through *Hydan*. To construct this distinguisher, first we have built a statistical model of *clean* executable files. Then, we have performed different concealment operations in a variety of executable files. We have analyzed the main differences between the set of *clean* executables and the set of *Hydan* modified executables. In this paper, we also describe possible countermeasures and the maximum capacity of *Hydan* steganographic files to overcome this steganalysis.

4.1 Statistical Analysis of Clean Executable Files

The distinguisher proposed is based on the presence of unusual sets of instructions on executable files. We have performed a statistical analysis of a set of 1261 *clean* executable files retrieved from */usr/bin* and */usr/sbin* of an *Ubuntu x86* distribution. Figure 1 shows the frequency distribution of the *functionality-equivalent instructions* sets for our set of files. This distribution tells the probability that a random instruction belongs to a *functionality-equivalent instruction* set. Depending on this distribution, the bandwidth of the covert channel offered by an executable may differ a lot. The bigger is the proportion of instructions belonging to a big set of *functionality-equivalent instructions*, the bigger will be the information *Hydan* is able to hide.

Our analysis has shown that all the *functionality-equivalent sets* of instructions are present in our test files. Nevertheless, most of the instructions found on the analyzed files belong to a small group of *functionality-equivalent instructions* sets. Therefore, the capacity of the covert channel depends on the capacity of these commonly used sets (Fig. 1). In order to build our statistical model, we have analyzed distribution of instructions inside each of the most frequent *functionality-equivalent instructions* sets.

One of the most used *functionality-equivalent instructions* sets is *toac32*. This set includes five different instructions. Thus, it can encode $\lfloor \log_2(5) \rfloor = \lfloor 2.32 \rfloor = 2$ bits. Frequency distribution of instructions inside the set is shown in Fig. 2.

Results obtained in the frequency analysis of this instruction set have been gathered in Table 2.

Table 2 Frequency distribution of instructions on *toac32* set

Instruction	Frequency
test r/m32, r32	100.0%
or r/m32, r32	0.0%
or r32, r/m32	0.0%
and r/m32, r32	0.0%
and r32, r/m32	0.0%

In all analyzed files, only one instruction of this set was used. In this case, a variation of the distribution of instructions within this set would be detected easily.

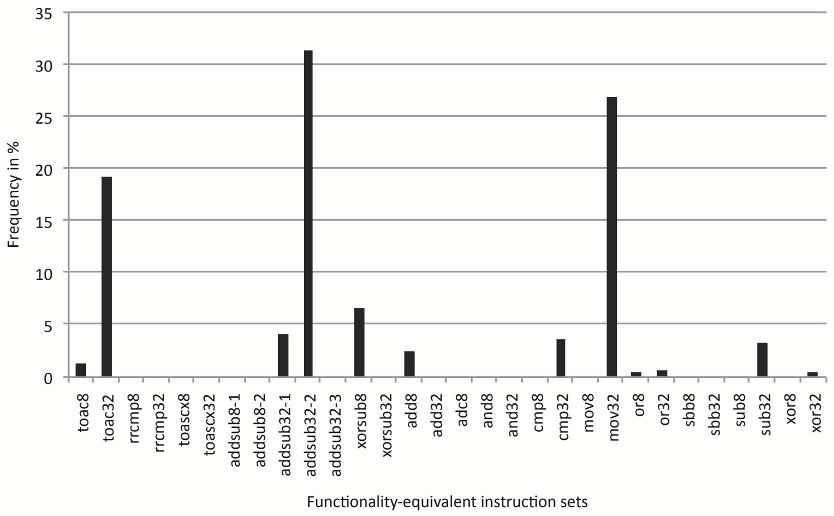

Fig. 1 Frequency distribution of *functionality-equivalent instructions* sets

For each of the remaining sets of equivalent functions, we have computed the frequency distribution of its instructions based on our set of executable files, as in the *toac32* set. Once we have constructed a frequency distribution model for each of the sets, we have also computed the proportion of instructions per set in each of the executable files. Each of the proportions computed for each file and *functionality-equivalent instructions* set has been compared using a chi-square statistic (χ^2) against the frequency distribution of that *functionality-equivalent instructions* set calculated for all the files. For each of the *functionality-equivalent instructions* sets we have calculated the average χ^2 statistic (Equation 1).

$$Average_{set_j} = \sum_{i=0}^{n} \frac{\chi^2_{file_i}}{n} \tag{1}$$

Where set_j is a *functionality-equivalent instructions* set, and $file_i$ is the ith file on our set of files. Figure 3 shows the average χ^2 for all the *functionality-equivalent instructions* sets. For most of the equivalent instructions sets, the distribution of its instructions has remained constant in all the executable files. Thus, its averaged chi-square is 0. *Functionality-equivalent instructions* sets with higher average value indicate that the frequency distribution of that sets has more variability between executable files. Figure 3 shows how six of the *functionality-equivalent instructions* sets suffer lots of variability on the distribution of its instructions depending on the executable file.

Differences introduced by *Hydan* will change the frequency distribution of instructions inside each of the *functionality-equivalent instructions* sets. Comparing the new instruction distributions obtained against the reference distributions for each of the *functionality-equivalent instructions* sets will allow to determine if information has been embedded into the executable file.

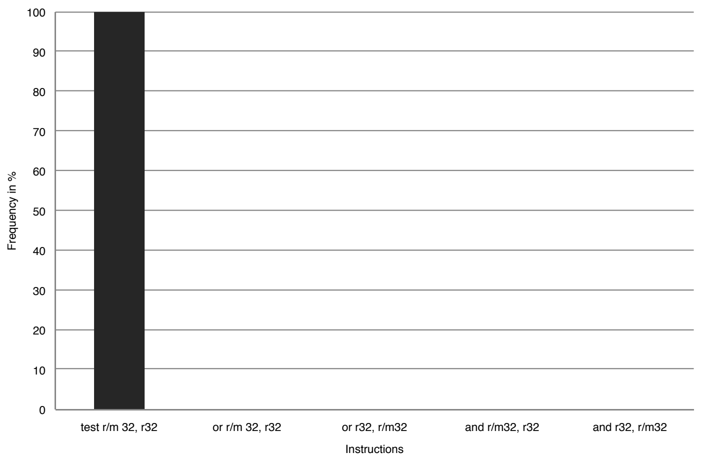

Fig. 2 Frequency distribution of instructions on *toac32* set

This can be easily seen through an example. Figure 4 represents the differences, in terms of a χ^2 statistic, on the frequency distribution of each *functionality-equivalent instruction* set of the *apt-get* executable file with no embedded information. Differences obtained are consistent with the average shown on Fig.3.

Inserting information into this executable file will modify the frequency distribution of instructions inside some of the sets of equivalent instructions. Figure 5 represents differences, in terms of a χ^2 statistic, on the distribution of instructions inside each of the equivalent instructions sets of the *apt-get* executable with embedded information.

Frequency distribution of instructions inside the highly variable functionality equivalent instruction sets has also offered high chi-square values, as in the reference (Fig. 3) and clean file comparison (Fig. 4). Nevertheless, distributions of some *functionality-equivalent instructions* sets have changed and its chi-square has increased comparing it with the reference comparison (Fig. 3) and the previous chi-square value (Fig. 4), which was 0.

The same procedure has been performed with all the executable files, obtaining for each set a model of the frequency distribution of that set. This has allowed us to establish which distributions of instructions inside *functionality-equivalent* instruction sets remain constant between different *clean* executable files.

These results have been used to build our distinguisher which is explained in the next section.

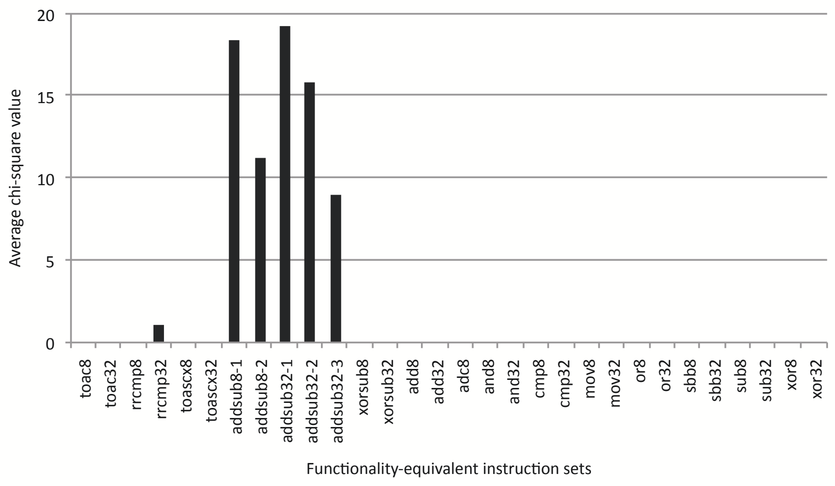

Fig. 3 Average chi-square statistic for each of the *functionality-equivalent instructions* sets

5 Distinguisher Design

The proposed distinguisher measures the changes on the distribution of instructions inside a selection of *functionality-equivalent instructions* sets. These measures have been made in terms of a χ^2 statistic against the reference distribution for each of the selected *functionality-equivalent instructions* sets. *Functionality-equivalent instructions* sets with high variability of instruction distribution between *clean* files have not been selected in the calculations of our distinguisher value. High variability may elevate the result offered by the distinguisher, marking some *clean* files as stego-objects. Our distinguisher only uses the *functionality-equivalent instructions* sets which its average chi-square value is 0, as calculated in 1. Therefore, 8 sets of *functionality-equivalent instructions* are not used: *toac8*, *rrcmp32*, *addsub8*, *addsub8-2*, *addsub32-1*, *addsub32-2*, *addsub32-3* and *xorsub8*. Mathematically, the value obtained with our distinguisher is expressed as follows:

$$D(file) = \sum_{i=0}^{n} \chi^2_{instruction\ set_i} \tag{2}$$

Where n is the number of sets of *functionality-equivalent instructions* whose average chi-square value is 0. To obtain the threshold of our distinguisher we have calculated all the results the distinguisher offers from three set files: a set of clean files, a set of files with embedded information using a 40 % of its capacity and a set of files with embedded information using an 80 % of its capacity. We have calculated the mean and standard deviation of values obtained by the distinguisher for the three sets. Results obtained are shown in Table 3.

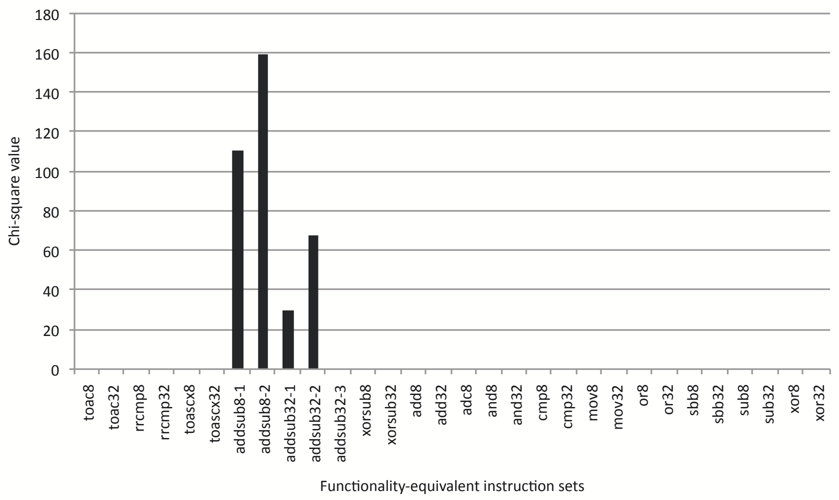

Fig. 4 Chi-square statistics for each of the equivalent instructions sets in *apt-get*

Table 3 Distinguisher results for different sets of executable files

Distinguisher	Clean	Hidden at 40%	Hidden at 80%
Mean	0.000604	151.254608	299.039886
Standard Deviation	0.024571	12.298561	17.292770

We have selected the threshold of our distinguisher as the addition of the mean and the standard deviation of the clean files set. When a file offers a value above the expected mean and typical deviation it is marked as a stego-object. Threshold of our distinguisher is described be as follows.

$$T = Mean_{clean} + T.Deviation_{clean} = 0.000604 + 0.24571 = 0.025175 \quad (3)$$

5.1 Results

With the selected threshold we have performed a test over three sets of files, each having 1063 files. The first set of files is a selection of *clean* files from the *Ubuntu 8.10 x86* distribution. Second set of files is the set of *clean* files with embedded information up to 40% of the capacity of each file. Last set is composed by the first

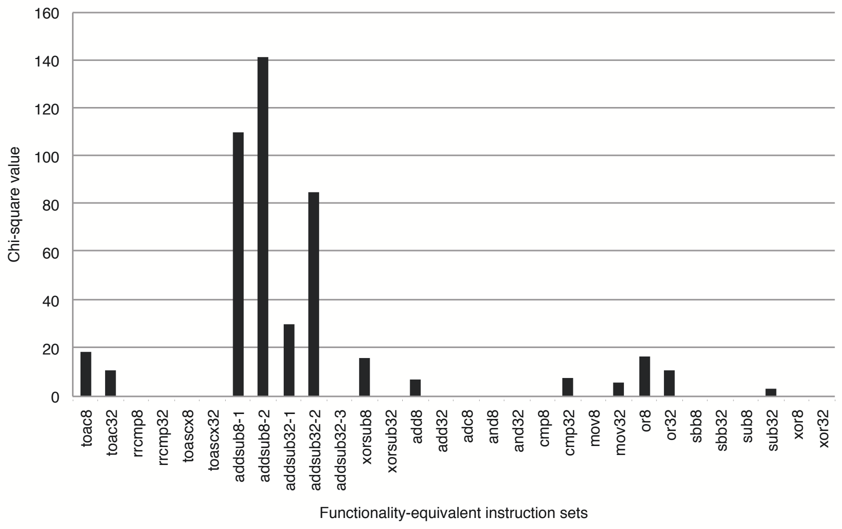

Fig. 5 Chi-square values for each of the equivalent instruction sets in *apt-get* with hidden information

set of files with embedded information up to an 80% of the capacity of each file. Distinguisher values obtained for each of the files are shown in Fig. 6.

Values obtained by our distinguisher for the clean files are separated from the ones offered by files with embedded information. Some results offered by embedded information files are low, but higher than the values returned by any of the clean files. In fact, our distinguisher has classified all the executables correctly (Table 4).

Table 4 Distinguisher classification results for different sets of executable files

	Expected clean executables	Expected embedded exec.
Predicted clean executables	1063	0
Predicted embedded exec.	0	2126

In order to produce executable files that are not detected by our tool some changes should be done to *Hydan*. Our analysis have shown that replacement of *functionality-equivalent instructions* is not secure if the frequency distribution of instructions inside a *functionality-equivalent instruction* set is constant. A first approach to secure Hydan would be to use only the functionality-equivalent instruction sets not used by our distinguisher. This would reduce the capacity of hidden infor-

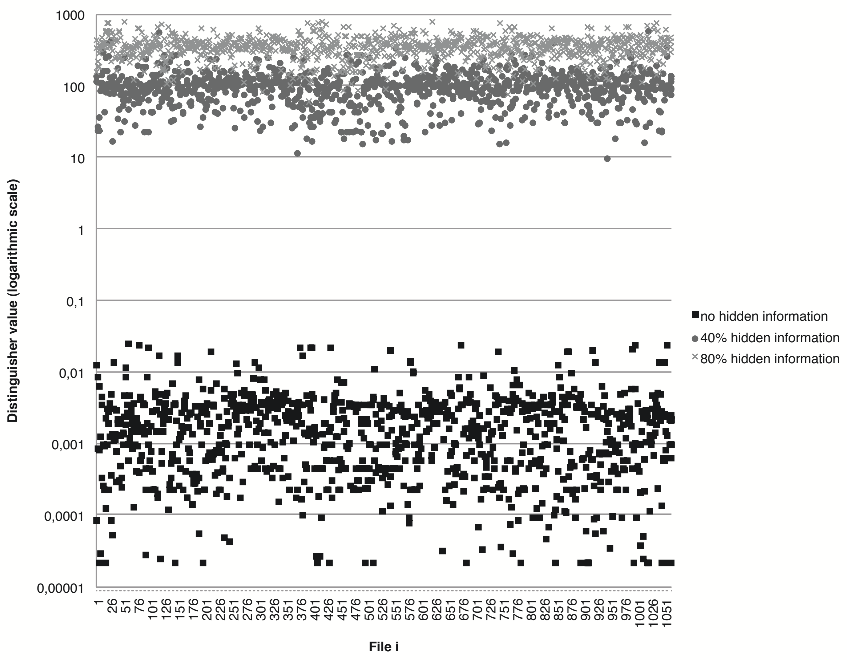

Fig. 6 Distinguisher results for sets of executable files

mation up to a 35% of the original capacity. Stego-files generated this way would not be detected by the distinguiser, producing false negatives.

6 Conclusions and Future Work

Steganalysis techniques are needed in order to ensure and improve the security of stego-systems in the same way cryptanalysis is needed to foster the security of cryptography techniques. With this work, we have developed a distinguisher that is able to recognize executable files with hidden information through *Hydan*. To create our distinguisher we have built a statistical model of *clean* executable files. In our tests, the proposed distinguisher classified correctly all executable files in different proportions of concealment (0%, 40% and 80%). We have also described how to overcome this steganalysis. Research on steganography of executable files is not extensive at the moment, but improvements to secure *Hydan* and other related steganographic tools [1] could only be achieved through extensive research in the field. We have advanced in this direction, and plan to further advance by refining the steganalytic methods proposed in [1] against *Stilo*.

References

1. Anckaert B., De Sutter B., Chanet D., De Bosschere K.: Steganography for Executables and Code Transformation Signatures. Lecture Notes in Computer Science **3506**, 425–439 (2005)
2. El-Khalil, R.: Hydan: Hiding Information in Program Binaries (2003). Lecture Notes in Computer Science **3269**, 187–199 (2004) http://crazyboy.com/hydan/. Cited 20 Oct 2008
3. Hernandez-Castro J.C., Lopez I.B., Tapiador J.M.E., Ribagorda A.: Steganography in Games. Computers and Security **25**(1), 64–71 (2006)
4. Johnson N.F., Jajodia S.: Exploring steganography: Seeing the unseen. Computer **31**(2), 26–34 (1998).
5. Kipper, G.: Investigator's Guide to Steganography. CRC Press (2004)
6. Murdoch S.J., Lewis S.: Embedding Covert Channels into TCP/IP. Lecture Notes in Computer Science **3727**, 247–261 (2005)
7. Naor M., Yung M.: Universal One-Way Hash Functions and Their Cryptographic Applications. Proceedings of the twenty-first annual ACM symposium on Theory of computing, pp. 33–43. ACM, New York, NY, USA (1989).
8. Peterson W., Brown D.: Cyclic Codes for Error Detection. Proceedings of the IRE **49**(1), 228–235 (1961)
9. Petitcolas F.A.P., Anderson R.J., Kuhn M.G.: Information Hiding:A Survey. Proceedings of the IEEE **87**(7) pp. 1062–1078 (1999)
10. Petitcolas F.A.P.: MP3Stego (2006). http://www.petitcolas.net/fabien/steganography. Cited 20 Oct 2008
11. Shirali-Shahreza M., Shirali-Shahreza M.H.: Text Steganography In SMS. Int. Conference on Convergence Information Technology pp. 2260–2265 (2007)
12. Simmons G.J.: The History of Subliminal Channels. IEEE Journal on Selected Areas in Communications, **16**(4), pp. 452–462 (1998)
13. Zhu W., Thomborson C.: Recognition in Software Watermarking. Proceedings of the 4th ACM international workshop on Contents protection and security, pp. 29–36. ACM (2006)