

How to swap instructions midstream: An embedding algorithm for program steganography

Ryan Gabrys
ryan.gabrys@navy.mil
Naval Information Warfare Center
San Diego, CA

Luis Martinez
luis.martinez@navy.mil
Naval Information Warfare Center
San Diego, CA

Sunny Fugate
fugate@spawar.navy.mil
Naval Information Warfare Center
San Diego, CA

ABSTRACT

In this work, we propose an encoding/decoding algorithm for program executable steganography. Some salient features of our approach is that unlike previous work it does not require the introduction of new instructions, which may be detectable. Furthermore, our scheme does not require storing the locations of where changes in the program executable are made.

CCS CONCEPTS

• **Security and privacy** → **Database and storage security**;
Data anonymization and sanitization.

KEYWORDS

steganography, information hiding

1 INTRODUCTION

Steganography is the process of embedding hidden information into a cover object. One of the well-studied problems in the field of steganography is to design an embedding scheme where the cover object “appears” the same before and after the embedding has taken place. There are many previous works that design embedding schemes by modifying the redundant data in digital cover objects such as images and videos. [3]

This work is concerned with the less studied problem of designing embedding schemes for program executables. We note that this problem is fundamentally different than the problem of designing steganographic schemes for digital media. Modifying even a single line of executable code can cause the program to perform drastically different, and in some cases, even break. Previous works such as Hydan and Stilo [1, 5] have proposed switching between semantically equivalent instructions in order to embed data into program executables. The fundamental drawback to such an approach is that, since these techniques often make use of unusual instructions, detecting the presence of hidden information in cover objects after these techniques have been applied is relatively straightforward and several studies have shown that Hydan in particular is easily detectable [2]. Rather than substitute equivalent instruction sequences, and in order to make detection more difficult, the approach taken here is to permute the order of instructions.

Our preliminary work [6] addressed the question of identifying a large set of pairs of assembly-level instructions such that for any pair, when the order of the two instructions in the pair is switched, the functionality and performance of the program is unchanged. These instruction pairs were termed *interchangeable pairs*, and the main result in [6] was to illustrate the existence of large sets of interchangeable pairs in across Linux programs.

As an illustration, suppose $\{i_j, i_k\}$ is an interchangeable pair of instructions where instruction i_j is the j -th instruction in the program and i_k is the k -th instruction in the program. Then, for simplicity and as a starting point, we restrict our attention to the case where $k = j + 1$ and k is even. Therefore, under this setup, a program with 6 lines contains at most 3 pairs of interchangeable instructions.

One straightforward way to embed information into an executable program provided a set of interchangeable pairs is the following. Suppose $\{i_1, i_2\}$ are two instructions and that $i_1 < i_2$ so that i_1 is lexicographically smaller than i_2 . Then, we can embed a single bit of information into this pair of interchangeable instructions by changing the order of i_1 and i_2 . For instance, if i_1 appears before i_2 in the program we can read this information as a 0 and otherwise if i_1 appears after i_2 we can read this information as a 1.

The drawback to this approach is that in order to decode, one has to know the locations of the interchangeable pairs of instructions. In this work, we propose a scheme which does not require knowledge of the set of interchangeable pairs of instructions. The gist of the approach is to generate a small collection of random matrices where one matrix in this collection is used at a time for the embedding. To enable unique decoding, one simply needs to store the index of the matrix used from this collection during the encoding process. One of the main results of this work which is stated in Corollary 3.2 is that the size of this collection is small so that we can store the index using a small number of additional information bits. We note that the main difference between the approach described here and the one in [4] is that we rely on a collection of matrices for the embedding whereas the goal in [4] was to identify one such matrix.

2 ENCODING AND DECODING ALGORITHM

We assume the executable program consists of $2n$ instructions. We represent the instruction sequence using a binary vector \mathbf{x} of length n . The idea is that every adjacent pair of instructions will be mapped to a bit in \mathbf{x} that indicates the order of the instructions in the pair. Let $\mathbf{x} = (x_1, \dots, x_n) \in \{0, 1\}^n$ be the binary vector which corresponds to the first $2n$ instructions in the executable. Suppose that I_j represents the j -th line of text in the assembly executable. Formally, we define \mathbf{x} so that

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HotSoS '20, April 7–8, 2020, Lawrence, KS, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7561-0/20/04.
<https://doi.org/10.1145/3384217.3384224>

$$x_j = 0 \text{ if } I_{2j} \geq I_{2j-1}, \quad x_j = 1 \text{ else.} \quad (1)$$

Let $J \subseteq [n]$ be the set of interchangeable pairs of instructions. In particular, if $j \in J$, then it follows that we can swap the instructions I_{2j} and I_{2j-1} without affecting the functionality of the program. The set J is only known to the encoder at the time of encoding and our method works with any set J . Let \mathbb{F}_2 denote the Galois Field of size 2. For a matrix $M \in \mathbb{F}_2^{m \times n}$, let M_J be a sub-matrix which is composed of the columns of M indexed by the set J . For example, if $M = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$, then $M_{\{2,4\}} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$.

The procedure for encoding is the following. Suppose we want to encode the information $\mathbf{u} \in \mathbb{F}_2^{|J|}$. Let S be a set of random seeds and we suppose that $S = \emptyset$ initially.

- (1) Generate a $|J| \times n$ random matrix M over \mathbb{F}_2 using one of the seeds from S . If all the seeds in S have been attempted then randomly generate the matrix M using a new seed s' .
- (2) If M_J does not have full rank, then go back to step 1). Otherwise, if M_J has full rank continue to the next step.
- (3) If $s' \notin S$ and M was generated using s' , then add s' to S .
- (4) Let $\mathbf{z} = M_{[n] \setminus J} \cdot \mathbf{x}_{[n] \setminus J} \in \mathbb{F}_2^{|J|}$. Let $\hat{\mathbf{z}} = \mathbf{z} + \mathbf{u}$. Then, let

$$\mathbf{y} = M_J^{-1} \cdot \hat{\mathbf{z}} \in \mathbb{F}_2^{|J|}.$$

- (5) Let $\hat{\mathbf{x}}_{[n] \setminus J} = \mathbf{x}_{[n] \setminus J}$ and let $\hat{\mathbf{x}}_J = \mathbf{y}$.
- (6) For every $j \in [n]$ where $x_j \neq \hat{x}_j$ swap the instructions I_{2j} and I_{2j-1} .
- (7) Store the index of the seed used to generate M .

Next, we discuss the decoding algorithm.

- (1) Recover the seed which was used to create the matrix M in step 1) of the encoding algorithm. Next, recover the matrix M using the seed.
- (2) Let \mathbf{x} be the vector which represents the first $2n$ instructions of the assembly executable according to (1).
- (3) Recover the vector $\hat{\mathbf{u}} = M \cdot \mathbf{x} \in \mathbb{F}_2^{|J|}$.

Now, we prove the correctness of our decoding algorithm.

THEOREM 2.1. *The vector $\hat{\mathbf{u}}$ satisfies $\hat{\mathbf{u}} = \mathbf{u}$.*

PROOF. First, note that step 5) is correct since the matrix M_J by construction has full rank. Furthermore,

$$M \cdot \mathbf{x} = M_J \cdot \mathbf{y} + M_{[n] \setminus J} \cdot \mathbf{x}_{[n] \setminus J} = \hat{\mathbf{z}} + \mathbf{z} = (\mathbf{z} + \mathbf{u}) + \mathbf{z} = \mathbf{u}.$$

□

3 ANALYSIS AND RESULTS

Next, we turn to proving the efficiency of the algorithm. In particular, we bound the number of times the encoding algorithm executes step 1).

THEOREM 3.1. *The probability step 1) of the encoding algorithm is executed more than T times is $(\frac{3}{4})^T$.*

PROOF. It can be shown that the probability that a $|J| \times |J|$ matrix has full rank is

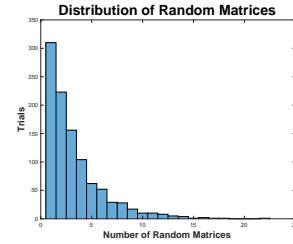
$$\prod_{j=1}^{|J|} (1 - 2^{-j}) > \frac{1}{2} \cdot \left(1 - \sum_{j=2}^{\infty} 2^{-j}\right) = \frac{1}{4},$$

which implies that the probability a random matrix does not have full rank is at most $\frac{3}{4}$. Since the event that the matrix M_J has full rank at each iteration i is independent of the event that the matrix M_J has full rank at iteration $i + 1$, the result follows. □

The next result follows from the previous theorem. The set S is the set of random seeds after the algorithm has been executed.

COROLLARY 3.2. *Suppose we encode N times using our algorithm. Then, $E[|S|]$ is at most $O(\log N)$.*

We evaluated the performance of the decoding algorithm by running 2^{10} trials where at each trial, we kept generating random binary matrices of dimensions 100×100 until we arrived at one which was full rank. The results of these trials are displayed in the histogram below. In the figure below, notice that most of the time only 1 or 2 random matrices had to be generated. In fact, the first bar shown below implies that in over 300 of the 1024 trials, only a single random matrix had to be generated (so that this initial random matrix had full rank).



We note that according to our analysis, the expected size of the set S is roughly 10. Our simulations required that $|S| = 22$ under the setup used to generate the histogram above.

4 CONCLUSION

In this work, we proposed a new encoding/decoding algorithm based upon random matrices for embedding information into program executables. Future work involves extending the scheme to handle more than a single type of program mutation.

REFERENCES

- [1] B. Anckaert, B. De Sutter, D. Chagnet, and Kan De Bosschere, "Steganography for executables and code transformation signatures," in *Proceeding of the 7th International Conference on Information Security and Cryptology*, Beijing, China, pp. 425-439, 2011.
- [2] J. Blasco, J.C. Hernandez-Castro, J.M.E. Tapiador, A. Ribagorda, and M.A. Orellana-Quiros, "Steganalysis of Hydan," in *Emerging Challenges for Security, Privacy, and Trust*, SpringerLink, pp. 132-144, 2009.
- [3] J. Cazalas, T.R. Andel, and J.T. McDonald, "Analysis and categorical application of LSB steganalysis techniques," in the *Journal of Information Warfare*, 13(3), 2014.
- [4] J. Fridrich and D. Soukal, "Matrix embedding for large payloads," in *IEEE Transactions on Information Forensics and Security*, vol. 1, no. 3, pp. 390-395, 2006.
- [5] R. El-Khalil and A.D. Keromytis, "Hydan: hiding information in program binaries," in *International Conference on Information and Communications Security (ICICS)*, LNCS, Malaga, Spain, 2004, pp. 187-199, 2004.
- [6] R. Gabrys and L. Martinez, "A change would do you good: GA-Based approach for hiding data in program executables," in *The Genetic and Evolutionary Computation Conference (GECCO)*, ACM, Prague, pp. 285-286, 2019.
- [7] W. Mahoney, J. Franco, G. Hoff and J. T. McDonald, "Leave it to Weaver," in *Proceeding of the 8th Software, Security, Protection, and Reverse Engineering Workshop*, San Juan, Puerto Rico, pp. 1-9, 2018.