A Change Would Do You Good: GA-Based Approach for Hiding Data in Program Executables

Extended Abstract[†]

Ryan Gabrys
Naval Information Warfare Center
ryan.gabrys@navy.mil

Luis Martinez
Naval Information Warfare Center
lmmartin@spawar.navy.mil

ABSTRACT

We consider the application of a genetic algorithm (GA) to the problem of hiding information in program executables. In a nutshell, our approach is to re-order instructions in a program in such way that aims to maximize the amount of data that can be embedded while, at the same time, ensuring the functionality of the executable is not altered. In this work, we focus on the problem of identifying a large set of instructions which may be re-ordered, and some initial results on the IA-64 architecture are then presented that illustrate the potential benefit of such an approach.[*]

CCS CONCEPTS

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

KEYWORDS

Information hiding, genetic algorithm, tracing

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 [1,4] 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 makes 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.

This paper focuses specifically on the problem 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. We call these instruction pairs *interchangeable pairs*, and one of the primary goals in this work is to identify a large set of interchangeable pairs that can be used to embed information. 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 first focus 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 according to some ordering. 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. A more thorough discussion of the encoding and decoding functions of our proposed embedding scheme is deferred for a longer version of the work, and, in this work, we restrict our focus to identifying large sets of instructions which can be permutated for the purposes of embedding hidden information.

[*] 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).

GECCO'19, July 13-17, 2019, Prague, Czech Republic
© 2019 Copyright held by the owner/author(s). 978-1-4503-6748-6/19/07. . . $15.00
DOI: 10.1145/3319619.3321905

In Section 2, we discuss our approach for identifying sets of instructions that can be permuted and we present preliminary results in Section 3.

2 APPROACH

In order to find a set of interchangeable pairs of instructions, we first generate a variant of the original program which has switched the order of every interchangeable pair of instructions using a genetic algorithm. From this variant, we produce a set of interchangeable pairs by inspecting where the original program and the variant differ. Finally, in order to validate that the order of each interchangeable pair can be switched independently, for every pair in the set, we create a variant of the original program by switching the order of only a single interchangeable pair and verify the program variant maintains its functionality.

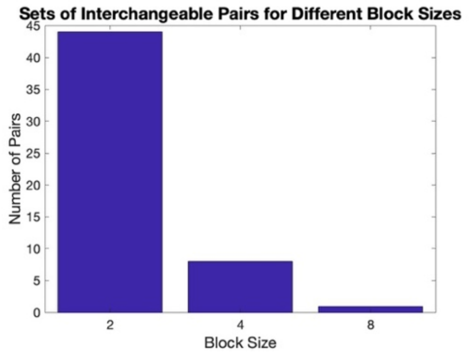
The use of genetic algorithms for modifying program executables has previously been explored in works such as [5,7,8]. Similar to [5,8], we also make use of the ``Software Evolution'' library to produce our program variant. The algorithm modifies executables through either of the following methods: a) two-point crossover, or b) mutation. The cross-over method takes as input two randomly selected variants of the same program. It replaces a portion of the instruction sequence from one variant with a portion of the instruction sequence from the other variant. The mutation method operates by first randomly choosing an odd and an even number. The odd number will be called the offset and the even number the length. The GA then swaps every pair of instructions contained within this region. For example, suppose the offset chosen is 5 and that the length is 6. Then, the GA would swap 3 instructions. In particular, it would swap i_5 and i_6, i_7 and i_8, and i_9 and i_{10}.

Our genetic algorithm maintains a pool of 100 candidates (or variants of the original program). At each iteration, the algorithm replaces the lowest scoring variant with a new variant that is either the result of the two-point crossover or our mutation method. The fitness function employed by our GA evaluates whether the program variant compiles properly, and then it tests the functionality of the program using a pre-defined set of tests. If the program does not compile correctly, it is assigned a default score of -200. Otherwise, if the program compiles but it returns an error code after any of the fitness runs exit with an error code, the program is assigned a fitness score of -100. Finally, if the program compiles correctly and does not return an error code, then its fitness score is equal to the number of tests that it successfully executed scaled by the percentage of instructions swapped when compared to the original binary (a higher percentage of swaps is preferred). The GA is terminated after 100,000 executions.

3 RESULTS AND DISCUSSION

As a starting point, we evaluated the genetic algorithm discussed in the previous section on the gcd program from the Linux coreutils library. The GA generated set of 44 interchangeable pairs. Since gcd is only comprised of 81 lines, roughly 54% of the lines of code were permutable. Since the gcd program is only 1434 bytes, this corresponds to an encoding rate of $\frac{44}{1434*8} \approx .004$, which is less than earlier proposed methods [1,4].

In order to increase the encoding rate, we extended our idea to larger blocks of code. In particular, we generated mutations where one block of code is swapped with an adjacent block of code. As a concrete example, suppose i_1, i_2, i_3, i_4 represent instructions 1-4 in our executable. Then, a block swap of size two could swap i_1 and i_2 with the instructions i_3 and i_4 so that the resulting sequence of instructions would then be i_3, i_4, i_1, i_2. We refer to the pair $\{\{i_1, i_2\}, \{i_3, i_4\}\}$ as a set of interchangeable pairs of block size 2. Note that under this setup, the order of i_1 and i_2 as well as the order of i_3 and i_4 could also be swapped leading to 4 potential ways of permutating the instructions given interchangeable pairs of block size 1 and 2. The results of modifying our GA to mutate gcd based upon larger blocks of code is shown below.



As can be observed from the table, the benefit of using larger blocks diminishes pretty quickly as the block size increases. Despite this limited benefit, we note that by leveraging interchangeable pairs of different block sizes, we were able to raise the encoding rate to $\frac{53}{1434*8} \approx \frac{1}{200}$, which is less than a factor of two off from the results reported in [1,4]. However, unlike [1,4], our method does not inject unusual instructions and is therefore more difficult to detect. In addition, we briefly note that it may be possible to identify larger sets of interchangeable pairs in larger executables (since they contain more instructions) so that the coding rate may be greater in many cases.

REFERENCES

[1] B. Anckaert, B. De Sutter, D. Chanet, and Kan De Bosschere, ``Steganography for executables and code transformation signatures,'' In *Proceeding of the 7th ICISS*, 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 *ECSPT*,, 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] R. El-Khalil and A.D. Keromytis, ``*Hydan: hiding information in program binaries,''* In *ICICS*, LNCS, Malaga, Spain, 2004, pp. 187-199, 2004.
[5] J. Landsborough, S. Fugate, S. Harding, ``Removing the kitchen sink.'' In *GECCO*, ACM, Madrid, Spain, pp. 833-838, 2015.
[6] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, ``Post-compiler software optimization for reducing energy,'' In *SIGARCH Computational Architecture News,* 42(1), pp. 639-652, 2014.
[7] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, ``Automated repair of binary and assembly programs for cooperating embedded devices,'' In *18th ASPLOS*, ACM, New York, NY, pp. 317-328, 2013.