Covert Communication through Executables

Bertrand Anckaert, Bjorn De Sutter and Koen De Bosschere
Electronics and Information Systems Department
Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
E-mail: {banckaer,brdsutte,kdb}@elis.ugent.be

Abstract

This paper explores the potential of executables for covert communication. Three techniques for the embedding of secret information are presented. Possible attacks on the stealthiness of these techniques are identified and countered. The presented concepts are implemented and evaluated for the IA-32 architecture, for which we obtain an encoding rate of 1/27 before countermeasures and 1/89 after countermeasures.

1 Introduction

Steganography embeds a secret message in a seemingly innocuous cover-object. Digital cover-objects most often are media, such as image and music files, that involve noise and are perceived by imperfect human senses. As a result, they contain many redundant bits, which can be modified to embed secret messages.

In this paper, we explore the fitness of executables as cover-objects. These differ significantly from media because changing as little as a single bit of a program can cause it to fail entirely. Hence different techniques are required to embed a message in an executables.

With the exception of Hydan [1], little information on this subject is publicly available. Therefore, we explore the available redundancy in executables and present three techniques to exploit this for the embedding of a secret message. Furthermore, the stealthiness of the different code transformations applied by these techniques is evaluated and a number of countermeasures are presented to prevent possible attacks.

2 Steganographic Potential

While changing a single bit in a program can cause it to fail, this does not imply a lack of potential for the purpose of steganography.

The element of noise can be replaced by an element of choice, as many equivalent executables exist for any real-life executable. We do not want to deoptimize the executable and therefore, we depart from the code a compiler has produced. Typically, a compiler's back-end goes through 4 phases: instruction selection, register allocation, instruction scheduling and code layout. Each of these phases involves an element of choice and we will show how this can be exploited for the embedding of bits.

2.1 Encoding bits in a choice

For each of the choices between equivalents, a number of bits can be encoded in the program. If there are n equivalents because of some type of choice, the number of bits that can be encoded can be computed as follows.

As $n \geq 2^{\lfloor \log_2(n) \rfloor}$, it is clear that at least $\lfloor \log_2(n) \rfloor$ bits can be encoded. This may however result in a significant decrease in encoding capabilities: if $log_2(n) \notin \mathbb{N}$ for large n, many equivalents do not correspond to an encodable bit-string.

Therefore we propose the following scheme: If $\log_2(n) \notin \mathbb{N}$, then $\lfloor \log_2(n) \rfloor = \lceil \log_2(n) - 1 \rceil$. We can thus always embed $\lceil \log_2(n) - 1 \rceil$ bits. If we associate each of the remaining $n - 2^{\lceil \log_2(n) - 1 \rceil}$ equivalents with one of the $2^{\lceil \log_2(n) - 1 \rceil}$ already used ones, we can embed an additional bit by allowing the embedder to choose between one of the two associated equivalents, as illustrated for $n = 7$ in Figure 1. Therefore, we can embed an extra bit in

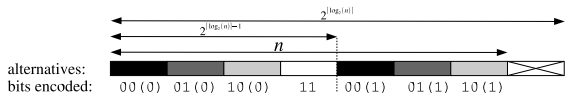

Figure 1: Encoding bits in the choice of 7 equivalents

$n - 2^{\lceil \log_2(n)-1 \rceil}$ of the $2^{\lceil \log_2(n)-1 \rceil}$ possibilities for the next $\lceil \log_2(n) - 1 \rceil$ bits.

If the embedded message is encrypted, all bitstrings to be embedded have equal probability, and hence the average number of bits that can be encoded in the choice out of n valid equivalents is given by

$$b(n) = \lceil \log_2(n) - 1 \rceil + \frac{n - 2^{\lceil \log_2(n)-1 \rceil}}{2^{\lceil \log_2(n)-1 \rceil}}. \qquad (1)$$

One can easily verify that Equation (1) also holds in the case that $\log_2(n) \in \mathbb{N}$

2.2 Instruction Selection

During instruction selection, intermediate code operations are translated into assembly instructions. Often multiple instructions can be chosen to implement an intermediate code operation. In the context of steganography, this choice can be exploited for the embedding of the secret message. If we have n alternatives for a certain instruction, then we can on average embed $b(n)$ bits in the choice of one of these alternatives.

Stealthiness and countermeasures: compilers are deterministic in nature. When a compiler has to choose between different alternative instructions, he will always select the same alternative under comparable circumstances. As a result, certain instructions simply never occur in clean programs. The presence of any such instruction would immediately give away the presence of secret information in an executable.

An embedder should thus only use alternative, possibly suboptimal, instructions when they could be the remnant of a piece of information that is no longer available to the attacker. This information could, e.g., be the source code or relocation information.

But even if multiple alternatives for an instruction do occur in clean programs, they tend to have a skew distribution. A possible attack would therefore be to compare the observed relative frequencies in an executable under evaluation to those of a large set of training executables.

This kind of attack can be countered by forcing the relative frequencies to resemble the expected frequencies during embedding.

2.3 Register Allocation

On the IA-32, the number of registers is very limited and most registers have fixed designations. Moreover, the calling conventions specify precisely how registers should be used. As a result, little information can be stealthily embedded in the register allocation.

2.4 Instruction Scheduling

Typically, instruction scheduling is performed per basic block. As two or more instructions that perform independent operations can be permuted within a basic block, we can encode bits in the instruction order within basic blocks.

To do so, we first determine all valid schedules by constructing a dependency graph of a block's instructions. Using a branch and bound algorithm to select instructions from the ready-set, we can easily generate all the possible schedules. Supposing there are n possible schedules, the number of bits that can be encoded on average is given by Equation (1).

Stealthiness and countermeasures: as compilers try to optimize the generated code, care needs to be taken not to generate schedules that are obviously suboptimal. Furthermore, they will generate identical schedules for identical dependency graphs. Therefore, we must limit the number of schedules generated for identical dependency graphs.

2.5 Code Layout

If there exists no fall-through path between two consecutive basic blocks, these blocks can be moved apart. Hence the order of the basic blocks in a program, i.e. the code layout, is to some degree free.

We encode bits in the ordering by dividing the m chains in in n sets of identical chains. We iteratively select a chain for placement out of the n remaining sets of chains. The average number of bits that can be encoded in this selection is once again given by Equation (1). Depending on whether the selected chain was the last of a set of identical chains or not, the number of sets will be $n-1$, respectively n in the next iteration. The process is repeated until all chains have been placed.

Stealthiness and countermeasures: clean programs exhibit spatial locality as the basic block chains of procedures will usually be placed consecutively. When the code in a program is reordered randomly, the spatial locality of the code is hence likely to decrease. A poor locality may therefore indicate the presence of a secret message.

To avoid this, we take a three step approach. First, chains are reordered within their procedures. Secondly, all procedures that are (transitively) connected through inter-procedural jumps are reordered. Finally these groups of procedures are reordered.

2.6 Extracting the Message

The extraction of the secret message is very similar to the embedding of the message. But to enable the extraction, some additional information is needed, e.g., to identify the basic blocks. This information is available at the embedding phase, as it is done at link-time, but this information is lost in the resulting executable.

Fortunately most of the necessary information can be derived from a static analysis of the executable program itself. As a consequence, we only need to communicate the discrepancy between the derived information and the actual information to the decoder. To do so, we store this information in the first instructions of the resulting binary.

3 Evaluation

To evaluate the presented concepts we have implemented Stilo, our steganographic tool for the IA-32 architecture, using the link-time rewriting framework Diablo [2], and applied it on 9 SPECint2000 benchmark programs to embed and extract "King Lear" by W. Shakespeare.

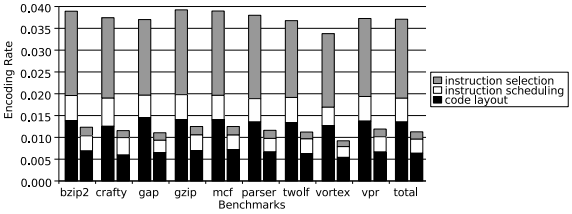

Figure 2: Encoding rate before (left) and after (right) countermeasures

The obtained encoding rates are presented in Figure 2. The distribution over the different techniques is also indicated. We achieve an encoding rate between 1/29.6 and 1/25.49 and a total encoding rate of 1/26.96 before countermeasures, four times the encoding rate of the previous prototype tool Hydan (1/110).

As the instruction distribution is fairly uniform across executables [1], little information can be stealthily embedded this way. When we take countermeasures to address this problem, only 9% of the encoding rate due to instruction selection remains.

Fortunately, there is a much greater variation in schedules across executables and 47% of the encoding rate due to instruction scheduling can be safely retained.

Furthermore, as code layout is largely determined by the source code, a piece of information that is no longer available to the warden, 59% of the encoding rate due to code layout can be safely retained.

Combined, we thus achieve a stealthy encoding rate ranging from 1/108.59 to 1/80.1 and averaging 1/88.76.

References

[1] R. El-Khalil and A. Keromytis. Hydan: Hiding information in program binaries. In *ICICS '04*. To appear.

[2] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of ARM binaries. In *LCTES'04*, pages 211–220.