Steganography for Executables and Code Transformation Signatures

Bertrand Anckaert, Bjorn De Sutter, Dominique Chanet, and Koen De Bosschere

Ghent University, Electronics and Information Systems Department, Sint-Pietersnieuwstraat 41 9000 Ghent, Belgium
{banckaer, brdsutte, dchanet, kdb}@elis.UGent.be
http://www.elis.UGent.be/paris

Abstract. Steganography embeds a secret message in an innocuous cover-object. This paper identifies three cover-specific redundancies of executable programs and presents steganographic techniques to exploit these redundancies. A general framework to evaluate the stealth of the proposed techniques is introduced and applied on an implementation for the IA-32 architecture. This evaluation proves that, whereas existing tools such as Hydan [1] are insecure, significant encoding rates can in fact be achieved at a high security level.

Keywords: code transformation signature, steganography, executables.

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.

This paper explores the largely unexplored field of steganography for executable programs. This differs significantly from steganography for media because changing as little as a single bit of a program can cause it to fail entirely. Hence different techniques are required for embedding messages in executables.

With the exception of Hydan [1], little information on this subject is publicly available. While the related subjects of software watermarking and fingerprinting, which also involve information hiding, have received considerably more attention [2, 3], the results of that research are not applicable in the context of steganography. This follows from the fact that watermarking and fingerprinting typically deal with very short embedded messages (shorter than 1 Kb), and that those messages first of all need to be irremovable, rather than hidden. Moreover, some watermarking approaches also require knowledge of the embedded message in the detection phase, which is obviously not possible in steganography.

Rather than implementing ad-hoc techniques, as in Hydan [1], we present a thorough study of the available redundancy in compiled programs. Furthermore,

C. Park and S. Chee (Eds.): ICISC 2004, LNCS 3506, pp. 431–445, 2005.

we present a general framework for evaluating the stealthiness of the different program transformations that exploit the redundancies. Based on this framework, a number of countermeasures to prevent possible attacks are presented.

This paper is structured as follows: Section 2 presents the used model. The fitness of executables for steganography is explored in Section 3. A framework for the evaluation of statistical signatures of code transformations is discussed in Section 4. The concepts are then evaluated for the IA-32 architecture in Section 5. Related work is the topic of Section 7 and conclusions are drawn in Section 8.

2 The Prisoners' Problem

We will follow Simmons' [4] classic model, a.k.a. *the prisoners' problem* for invisible communication. Alice and Bob are two prisoners in different cells. Wendy, the warden, arbitrates all communication between them, and will not let them communicate through encryption or suspicious communication. Both prisoners therefore need to communicate invisibly about their escape plan.

Furthermore, we will assume that the mechanism in use is known to the warden (Kerkhoffs' principle [5]). Hence its security must depend solely on a secret key that Alice and Bob managed to share, possibly before their imprisonment.

The general principle of steganography is as follows. To share a secret message with Bob, Alice randomly chooses a harmless message, called a cover-object c, which can be transmitted to Bob without raising suspicion. The secret message m is then embedded in the cover-object using the secret key k, resulting in a stego-object s. This is to be done in such a way that Wendy, knowing only the apparently harmless message s, cannot detect the presence of the secret. Alice then transmits s to Bob via Wendy. Bob can reconstruct m since he knows the embedding method and has access to the key k. It should not be necessary for Bob to know the original cover c. The security of invisible communication lies mainly in the inability to distinguish cover-objects from stego-objects. The task of Wendy can be formalized as a statistical hypothesis-testing problem, for which she defines a test function on objects (of the set O) $f : O \rightarrow \{0, 1\}$:

$$f(o) = \begin{cases} 1 & if\ o\ contains\ a\ secret\ message \\ 0 & otherwise \end{cases}$$

This function can make two types of errors: detect a hidden message when there is none (false positive) and not detect the existence of a hidden message when there is one (false negative). In this paper we will further assume that the warden is passive, i.e. she will not modify the object, but only classify it. This is generally accepted in steganography [6].

3 Fitness of Executables as Cover-Objects

While changing a single bit in a program can cause it to fail, this does not imply a lack of redundancy for the purpose of steganography. Instead the specific characteristics of software indeed result in many forms of redundancy.

In theory, we can consider two programs extensionally equivalent if they produce identical output, given identical input. In practice, more stringent requirements for time, space and power consumption need to be taken into account. But even then a large number of equivalent executables exists. This has been exploited for several purposes including program optimization, program obfuscation, software watermarking and fingerprinting, and software diversity. It is thus generally accepted that the number of equivalent executables for any real-life application is large and that there is indeed a lot of redundancy in a program which, in this context, we would like to exploit to encode a secret message.

Besides being equivalent to the original program, any program with an embedded message also needs to pass the warden's test function described in the previous section. Since we believe useless (suboptimal) code added to a program will be easily detected, we will only allow embedding transformations that do not deoptimize a program. In other words, the message should be embedded in the code a (optimizing) compiler back-end has produced from its intermediate code representation of the program. Typically a compiler's back-end goes through 4 phases (in varying orders), each of which inserts a number of redundancies.

During *instruction selection*, the intermediate code operations are translated into assembly instructions. Often multiple instruction sequences can be chosen to implement an intermediate code operation. During the *register allocation*, architectural registers are chosen to store values temporarily. Usually there are multiple valid allocations. In the *instruction scheduling* phase, the selected instructions are put in their final order. Again, multiple orderings are often valid. Finally, multiple compiled files are combined into a program. During this *code layout*, multiple orderings can be chosen. These types of choices/redundancies and their exploitation are the topic of this section. As the target architecture, we have chosen the IA-32 architecture [7], because it is most commonly used.

3.1 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 equivalent programs 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: it suffices to number each equivalent, and to take that equivalent who's (binary) number corresponds to the bit-string to be encoded. This simple approach may result in a significant decrease in encoding capabilities however: if $log_2(n) \notin \mathbb{N}$ for large n, many equivalents may not correspond to an encodable bit-string.

A more efficient scheme is as follows: 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 $n - 2^{\lceil \log_2(n)-1 \rceil}$ of the $2^{\lceil \log_2(n)-1 \rceil}$ possibilities for the next $\lceil \log_2(n) - 1 \rceil$ bits.

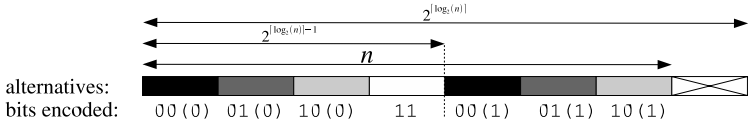

Fig. 1. Encoding bits in the choice of 7 equivalents

If the embedded message is encrypted with the secret key k, all bit-strings 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

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

One can easily verify that equation (1) also holds if $\log_2(n) \in \mathbb{N}$.

3.2 Instruction Selection

To explore the steganographic potential of executables, we have developed a tool that is capable of exhaustively generating all possible instruction sequences for the IA-32 architecture. This tool operates in a similar manner as the so-called superoptimizer [8].

Its input consists of a code sequence, a set of output registers and a set of (scratch) registers whose value is no longer used after the sequence has been executed in a program. For all generated sequences, the tool checks whether they perform the same function as the original code sequence, by testing the output values for all possible input values. If the test succeeds an equivalent sequence is found.

Because of the halting problem, it is in general undecidable if a generated sequence will terminate. Hence the equivalence test can run forever. By restricting the set of instructions to the integer instructions, that do not include any control flow, we can assure that each tested sequence terminates. But even then the number of potential equivalent sequences is still too large. To make the problem tractable, and to terminate the exhaustive generations within reasonable time, we further limit the immediate operands (constants encoded in an instruction) that can be used to $\{-1, 0, 1, 31\}$. Finally, we restrict the length of the generated sequences.

Even with these restrictions we can still find many equivalent sequences that perform realistic computations. For the operation `ECX`$= max($`EAX`,`EDX`$)$, e.g., our tool was able to find 433 different encodings of three instructions. Similarly, for the computation `EAX`$= ($`EAX`$/2)$, 3708 equivalent sequences of 4 instructions were generated. Note that the tool did not find shorter sequences because of the limited list of immediates that does not contain 2.

It should be noted that these examples are no exception. Moreover, the number of equivalents is exponential in the number of instructions: if we have n instructions which we can divide in groups of i instructions, of which each group

has at least a equivalents then combined we have at least $a^{\frac{n}{i}}$ equivalents. Furthermore, many additional equivalents arise when considering the larger piece as a whole, in which instructions can be moved from one group to another.

While our tool thus shows great potential for encoding bits, it is too slow for a practical tool. Hence we had the tool generate a database of equivalence classes for the instructions that occur most often in our suite of training programs. During this process, we imposed the additional restriction that equivalent instructions can only read/write locations that are read/written in the original instruction. However, if liveness analysis [9] determines that certain status flags are dead, we allow them to be overwritten. Finally, the set of immediates is expanded with the immediates used in the original instruction and the negate thereof.

3.3 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. Hence the little choice that a compiler in theory has to choose a register allocation, is in practice unexploitable: any deviation from the calling conventions would be spotted by the ward. As a result, changing the allocated registers is not an option to embed secret messages in IA-32 programs.

3.4 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 orderings by constructing a dependency graph of a block's instructions, in which dependent instructions are connected by directed edges. By iteratively removing instructions from this graph that do not depend on other instructions in it, a valid schedule can be determined. At each iteration, multiple instructions may be ready to be removed from the graph. They are, in other words, in the *ready-set* [9] of instructions. Using a branch and bound algorithm to select instructions from the ready-set, we can easily generate all the possible permutations. Supposing there are n possible schedules, the number of bits that can be encoded on average is given by equation (1).

Since finding valid instruction orderings using a dependency graph is time-consuming, and since the marginal gain of additional orderings decreases steadily when the number of orderings increases, it is useful to put an upper bound on the number of valid permutations that are considered. In our implementation this upper bound is 1024 orderings. As basic blocks are usually not longer than 4-5 instructions, this upper limit rarely is reached. Hence it has little influence on the amount of bits that can be encoded. For the rare, long basic blocks that offer billions of valid orders, setting an upper limit is absolutely necessary for obtaining practical execution times.

3.5 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. More precisely, all basic block chains, i.e. lists of consecutive basic blocks with fall-through paths between them, can be positioned in any order. When there are c different chains, we have $c!$ possible orderings to choose from, and hence we can encode $b(c!)$ bits.

While the order of unique elements to encode a bit-string can be exploited with existing methods [10], all chains in a program are not necessarily unique. This follows from the fact that most compilers only compile one source code module at a time, and hence never have an overview over all the code that constitutes a final program. As a result, duplicated code ends up in programs [11].

This problem is aggravated for our purpose, since we need to number and qualify all chains independently of their position in the program. Hence we cannot base our differentiation between two chains on any contents of them that depends on their location. *In concreto*, this means that all relocatable addresses [12] encoded in the instructions in the chains need to be neglected when comparing chains. For the programs in our benchmark suite the thus computed number of sets of identical chains is only between 47 to 59% of the total number of chains.

With m chains divided in n sets of identical chains $s_1 \dots s_n$, the theoretical average number of bits that can be encoded in their ordering is given by

$$b(\frac{m!}{\prod_{i=1}^{n}(|s_i|!)}). \tag{2}$$

We can approximate this number by iteratively selecting 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.

3.6 Interactions Between the Techniques

The discussed techniques are not completely orthogonal. In order to combine them successfully, a couple of issues need to be addressed.

First, it is worth noting that the number of bits that can be encoded in instruction selection is dependent on the chosen ordering of instructions in the basic block, and vice versa. When orders change, liveness ranges change, and hence the condition flags and scratch registers that may be changed by equivalent instructions also change.

For the same reason, instruction selection influences the order in which an embedder or extractor will generate equivalent orderings, and hence how specific bit sequences are encoded in the ordering. Vice versa, if scheduling is applied first, it influences the order in which equivalent instructions are generated.

Moreover, if the embedder first encodes bits in the instruction selection of the instructions in their original order in the program, and subsequently reorders

```
55          push EBP          |  55          push EBP
89 e5       mov  ESP,EBP      |  89 e5       mov  ESP,EBP
83 ec 08    sub  0x8,ESP      |  83 c4 f8    add  0xfffffff8,ESP
```

Fig. 2. Two equivalent code sequences

the instructions, the extractor does not know the order in which the information embedded in the instruction selection needs to be extracted. Clearly, the extractor and the embedder need to depart from the same dependency graph in order for the extractor to obtain the correct embedded information.

Before the embedding and the extraction, all basic blocks in a program should therefore be transformed into a canonical form, in which both the instruction selection and their ordering are predetermined.

3.7 Practical Considerations for Extracting an Embedded Message

In order to extract embedded information from a program, an extractor needs to identify the basic blocks, and he needs to pinpoint relocated operands, since these should be neglected for the ordering of chains.

The necessary relocation information is available at the embedding phase, as the embedding is done at link-time, when the whole program is first available. This information is lost in the resulting executable however.

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 can store this information in the first instructions of the resulting binary, without taking liveness information into account. This is the only option since the decoder cannot identify basic blocks or chains and it cannot compute liveness information at this point.

4 Code Transformation Signatures

While Section 5 shows that the encoding rate achieved by the discussed techniques is fairly high, its security is obviously too low. The reason is that the techniques introduce very unusual code that will arise suspicion of the warden. Consider, e.g., the equivalent code sequences in Figure 2. Anyone somewhat familiar with assembly code will agree that the likelihood of a compiler generating the code on the right is extremely low. But this code is present in executables that have been put through Hydan or our tool (without countermeasures). In short, the application of our tool has left an obvious signature.

We define a *code transformation signature* (CTS) as a code property that results from that transformation. The security of the discussed embedding techniques depends by and large on the absence of such signatures. While this is obvious for steganography, it is also of importance for other embedding techniques such as watermarking and fingerprinting, as the distortion of a watermark or fingerprint is facilitated if an attacker can accurately locate it.

Despite the importance of the stealthiness of applied code transformations, almost all research efforts have targeted the development of new techniques. Little work has been done on the security evaluation of the techniques. So far, most claims for security have been ad hoc and often based on author's belief.

4.1 A Framework for Detecting Code Transformation Signatures

Because quantitative methods have proved so powerful in many other domains, we will first quantify unusual properties using quantitative software metrics. On these metrics, we build models of the expected behavior, after which we can compare the observed value of a metric to the expected behavior, and thus classify software into clean and suspect software.

Software Metric. A software metric summarizes and quantifies properties of a given piece of software, called a *unit*, in order to detect signatures. Hence the property to measure depends on the applied code transformations. Metrics can in general be classified along two axes: that of *aspects* and that of *granularities*.

The aspect identifies what type of software unit is inspected. This could, e.g., be the static code or the dynamically executed code. It could be the heap or the stack as well, as they result from the executed code. We should note that even a dynamic data watermark [2] may introduce a signature in the static code.

The granularity of a metric identifies the size of the unit that is the subject of measurement. Possible granularities are the instruction, the basic block, the procedure, the memory location, the graph structure, etc. Granularity is important for an attacker, because, the smaller the granularity, the more accurately the attacker can pinpoint the location of the suspicious software.

Statistical Code Model. In order to evaluate executables for the presence of suspicious units with respect to some metric, we need a model of what constitutes a "clean" unit. We will do so by means of statistical distributions that are constructed by evaluating a population of units for some metric. On such a distribution, a statistical test can then be postulated that decides on the behavior of a unit under investigation.

For each model, the population's *locality* identifies how closely related the units that make up the population are to the unit under investigation. If the granularity of the metric is, e.g., a basic block, then we could test each block by comparing it to the blocks in its own procedure or we could compare it to all the blocks in a training set of programs. In the former case, the locality of the model would be that of procedures, in the latter that of the software universe.

Based upon the postulated model of the clean behavior of a metric, we can then compute how unusual it is to observe a particular value for a metric. If we then define a threshold to differentiate suspect units from clean ones, we obtain a statistical test. In some cases, a single CTS will suffice to classify units, but in other cases several CTSs will need to be combined to increase the reliability.

Stealthy Code Transformations. Knowing that a warden uses such statistical models to detect CTSs of suspicious code, we need to defend against them.

This can either be done by elevating the false negative rate of a test, i.e. thwarting the recognition of the CTSs, or by elevating the false positive rate, i.e. transforming original code to contain the same CTSs.

Consider, e.g., a tamper-proofing mechanism that reads a piece of the program code, computes a checksum over it and compares it to some predefined value. Since programs rarely read their own code, an attacker trying to locate the detection mechanism may search for the CTS consisting of a (static) instruction that reads the code section. A countermeasure against this attack consists of hiding the fact that the instruction reads from the code section by obfuscating the involved address computation (increasing the false negative rate) or by transforming the original code to read constant values from the code section throughout (increasing the false positive rate).

4.2 Application of the Framework to the Steganographic Techniques

Presence of Unusual Instructions. When multiple instructions are available to perform a given operation, a compiler will deterministically select the (assumed) optimal equivalent during the instruction selection phase. A CTS can therefore be found in an unusual instruction in the static code.

To detect such CTSs, the aspect of the metric is static code, and the granularity is that of instructions. The locality of the model is that of a large, predefined set of clean programs. The test considers a program suspicious if it contains an instruction that never occurred in the clean programs.

As a countermeasure against this test, an embedder should only use alternative, possibly suboptimal, instructions when they could be the remnant of a choice based on information that is no longer available to the warden. This information could, e.g., be the source code or relocation information.

Unusual Relative Frequencies of Instructions. Replacing instructions by equivalent ones changes the frequencies with which the equivalents occur in a program. Therefore a suitable metric is the relative frequency of equivalents in the static code of the program. When an encrypted secret message is embedded in an executable using equivalent instructions (and without taking countermeasures), the equivalents for a certain instruction can be expected to have uniform frequencies. In clean programs, by contrast, they likely have skew distributions.

Suppose that there are n equivalents for some instruction, whose relative frequencies in a programs x are presented by $\mathbf{f}(x) = [f_1(x) \ldots f_n(x)]$. To measure how clean a particular program p is, the metric then consists of the euclidean distance from $\mathbf{f}(x)$ to $\mathbf{f}(P)$, in which P is large set of clean programs. The model of programs can be obtained by postulating that this distance follows a normal distribution, of which the parameters μ and σ^2 can be estimated by computing the average and variance of the distance in P.

To classify an executable e we compute the probability that a distance occurs that is larger than the observed distance between $\mathbf{f}(e)$ and $\mathbf{f}(P)$. A low likelihood indicates a high probability that a secret message is present. The reliability of this test can be increased by averaging the likelihoods of a number of sets of

equivalent instructions. The classification could then classify programs with a score below, e.g., 0.5 as suspect.

To avoid the detection of the presence of an embedded message through the discussed CTS, we need to approximate the relative frequency of equivalents through a biased embedding. To do so the equivalents for a particular instruction are partitioned in two groups with relative frequency r_1 and $r_2 = 1 - r_1$ as close to 0.5 as possible. Without loss of generality we can assume that $r_1 \geq 0.5$. Using a random number generator seeded with a value derived from the secret key k (to assure that both the embedder and extractor follow the same scheme), numbers are then generated, on which a test is defined that succeeds with chance $p = (r_1 - 0.5) * 2$. Whenever the test succeeds for a generated number, we are forced to take an equivalent from group 1, and no bit is encoded. Otherwise, we encode a bit by choosing between group 1 and group 2. As a result of Bernoulli's theorem, the relative frequencies r_1 and r_2 will converge to the chance $p(g_1)$, that group 1 is chosen, resp. $p(g_2)$, that group 2 to is chosen, with increasing n. As it is equally likely that the next bit will be a 0 as a 1, $p(g_1) = (r_1 - 0.5) * 2 + \frac{1}{2} * (1 - (r_1 - 0.5) * 2) = r_1$ and $p(g_2) = \frac{1}{2} \times (1 - (r_1 - 0.5) * 2) = r_2$. Note that if a group is not a singleton, we iteratively partition it in subgroups on which the the same process is applied.

Unusual Jump Behavior of Program. Clean programs exhibit spatial locality as the basic block chains of procedures will usually be placed consecutively. This locality is further increased when cache-optimizing code layout techniques are applied [13]. 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, and a suitable metric to test this consists of the average offset of executed jumps that cross chain boundaries.

In the execution of our population of training programs, the largest average jump offset during a program's execution was $3,725$. Respecting a considerable margin we therefore postulate the model that this average will be no higher than $10,000$ for clean programs. For our evaluation benchmark programs the largest average observed before embedding was $2,575$, while the smallest average observed after embedding (without countermeasures) was $59,815$. The classification thus has a perfect false positive and false negative rate for our benchmarks.

To avoid the presence of this CTS, we must retain the spatial locality present in the original code. To achieve this, it suffices to not reorder all of the different chains at once, but to take a three step approach. First, chains are reordered within their procedures. Secondly, all procedures are partitioned into equivalence classes, where two procedures are considered equivalent if they are (transitively) connected through inter-procedural jumps (that most often result from tail-call optimization by compilers), and procedures are reordered within their class. Finally, all classes are reordered.

Diversity in Scheduling. When instruction schedulers pick instructions from the ready-set (see Section 3.4), they try to optimize the generated schedules by taking instruction latencies, available processor resources, and other properties

into account. As a result, any badly scheduled basic block will raise suspicion. Moreover, schedulers typically are deterministic, and hence two identical dependency graphs of instructions will most often be scheduled identically within a program. An important exception is where libraries and the main application are compiled independently of each other. Furthermore, there may be considerable differences between schedules in different programs. Different compilers may have been used, or different optimization levels, or different processor targets. Obviously, the warden does not know which combination has been used.

One attack on this type of CTS consists of comparing the basic blocks in a program, and to consider identical dependency graphs that are scheduled differently suspicious. For this type of model, the metric's aspect would be static code, and the granularity the basic block, while the locality is the equivalence class of the block's procedure within the program.

Since the warden does not know which properties the original scheduler has taken into account, he cannot know which graphs were identical according to the original scheduler. So while the warden may consider two graphs identical, the compiler may have seen them as different. Hence the warden needs to build a statistical model of clean code with a large set of programs.

The approach resembles the one to detect suspect instruction frequencies. Now $\mathbf{f}(x)$ indicates how often a compiler has generated i different schedules for (assumingly) identical graphs. Again, we can compute the euclidean distance and obtain a model by postulating that this distance follows a normal distribution. In practice, we observed that 95% of the graphs occurring more than once occurred with a single schedule, while approximately 5% occurred with two different schedules. More schedules are rare.

As a countermeasure to these attacks, we suggest the following approach: instead of choosing any instruction from the ready-set, limit this choice to the set of, within reasonable boundaries, good instructions to schedule. Furthermore, identical dependency graphs should result in i schedules with chance f_i. To implement this, it suffices to maintain a database of already scheduled blocks and enforce i different schedules with chance f_i.

Please note that making compilers non-deterministic, to increase the false positive rate, is not an option: besides the simple fact that one cannot control all compilers, making them non-deterministic would make debugging the compilers themselves and the compiled programs even more difficult than it is today.

5 Experimental 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 [14], and applied it on 9 SPECint2000 benchmark programs to embed and extract "King Lear" by W. Shakespeare. The programs were compiled with GCC 3.2.2 and linked to glibc 2.3.2 for Linux. For each benchmark, the embedding and extraction took less than a minute on a 2.8GHz Pentium IV.

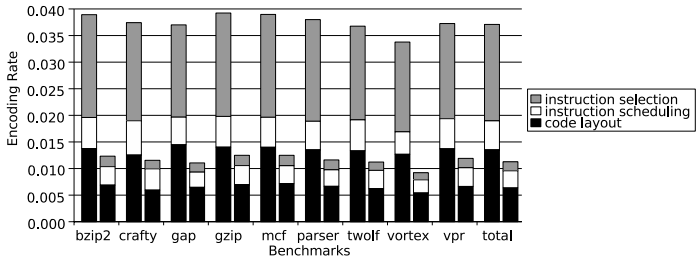

Fig. 3. Encoding rate before (left) and after (right) countermeasures for steganalysis

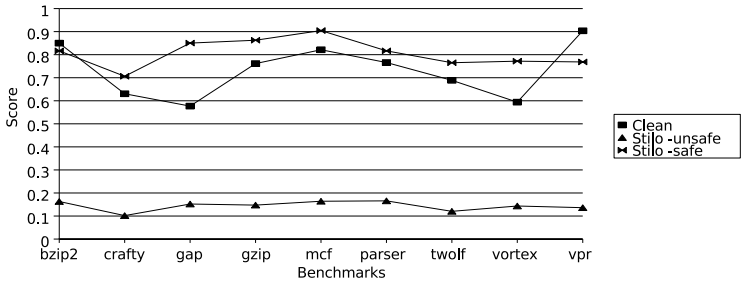

Fig. 4. Code transformation signature: unusual relative frequencies of instructions

The obtained encoding rates are presented in Figure 3. 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).

Without countermeasures, neither our tool nor Hydan, as noted by its authors, is stealthy. Most importantly, their encoding rate achieved through instruction selection is achieved by and large by choosing very unusual instructions. However, as the clean instruction distribution is 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, as can be seen in Figure 3. This clearly illustrates the usual trade-off between data rate and stealth.

The need for countermeasures is acknowledged by the results of our attack on the unusual instruction frequency CTS as discussed in Section 4.2. These results are presented in Figure 4, and confirm that unlimited instruction selection freedom indeed results in big, easily detectable discrepancies from clean code.

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. This is still higher than the unsafe rate of Hydan.

6 Future Work

The techniques discussed in this paper are portable to other architectures, as they exploit compiler choices common to all architectures. It would be interesting to see how the characteristics of an architecture influence the obtained data encoding rate. For example, a RISC architecture is unlikely to have the same redundancy in its instruction set as the IA-32 architecture. Therefore, the data rate due to instruction selection can be expected to be much lower on a RISC. On the other hand, RISC architectures typically have a larger set of registers, as a result of which register allocation might be a safe place to hide information.

While we have taken measures to prevent the detection of the presence of secret information in an executable in general, all executables generated by a single programmer are likely to be generated by the same compiler, with the same compiler flags, libraries, etc. If an embedder wants to use this tool repeatedly to defeat the same warden, his freedom of choice may need to be further reduced to assure that the attacker is not alarmed when different executables from the same programmer are unlikely to have been compiled with the same tool chain. This requires future research.

7 Related Work

Several types of cover-objects have been used to embed a secret message. The first reported occurrence is due to Herodotus. He tells of Histiæus, who shaved the head of his most trusted slave and tattooed it with a message that disappeared after his hair had regrown. Many other physical objects have since been used as cover-objects, e.g, earrings, written documents, and music scores.

Digital steganography has mainly been applied to media, such as images, sound and video. A large number of systems has been proposed [15, 16].

Steganography in the context of executables has, to the best of our knowledge, only been addressed by Hydan [1], a steganographic tool for IA-32 compatible executables.

Significantly more research has been conducted in the related field of software watermarking. The first one, proposed by Davidson and Myhrvold [17], encodes the watermark in the sequence of basic blocks. Pieprzyk [18] suggests assigning a unique identity to every copy in the choice of equivalent instructions. Another approach encodes the watermark in the frequency of groups of instructions [3]. All of these approaches change properties of the existing executable. Other techniques add a piece of data [19] or code [20] to the original program.

Whereas the mentioned work has mainly focused on the development of new techniques, more attention has recently gone into the evaluation of their security [21, 22, 23]. No general framework has been presented however.

8 Conclusion

This paper identified the redundancy present in executable programs and presented instruction selection, instruction scheduling and code layout as three techniques to exploit this redundancy for steganography. Combined, they resulted in encoding rates of approximately $\frac{1}{27}$, four times the rate of the previous approach by Hydan [1].

A framework for the evaluation of code transformation stealth was introduced and applied to the presented techniques, showing that our techniques can be made secure by the appropriate countermeasures, while still obtaining an encoding rate of $\frac{1}{89}$.

Acknowledgments

This work is supported by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), the Fund for Scientific Research - Belgium - Flanders (FWO) and Ghent University, member of the HiPEAC network.

References

1. El-Khalil, R., Keromytis, A.: Hydan: Hiding information in program binaries. In: International Conference on Information and Communications Security, LNCS. Volume 3269. (2004)
2. Collberg, C., Thomborson, C.: Software watermarking: Models and dynamic embeddings. In: ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, ACM Press (1999) 311–324
3. Stern, J., Hachez, G., Koeune, F., Quisquater, J.J.: Robust object watermarking: Application to code. In: Information Hiding, LNCS. Volume 1768. (1999) 368–378
4. Simmons, G.J.: The prisoners' problem and the subliminal channel. In: Advances in Cryptology. (1984) 51–67
5. Kerkhoffs, A.: La cryptographie militaire. Journal de Sciences Militaires **9** (1883) 5–38
6. Anderson, R.J., Petitcolas, F.A.: On the limits of steganography. I.E.E.E. Journal of Selected Areas in Communications (1998) 474–481
7. Intel: IA-32 Intel Architecture Software Developer's Manual. (2003)
8. Massalin, H.: Superoptimizer: a look at the smallest program. In: Architectual Support for Programming Languages and Operating Systems, IEEE Computer Society Press (1987) 122–126
9. Aho, A., Sethi, R., Ullman, J.: Compilers, Principles, Techniques and Tools. Addison-Wesley (1986)
10. Kwan, M.: Gifshuffle (1998) http://www.darkside.com.au/gifshuffle/.
11. De Sutter, B., De Bus, B., De Bosschere, K.: Sifting out the mud: Low level c++ code reuse. In: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. (2002) 275–291
12. Levine, J.: Linkers & Loaders. Morgan Kaufmann Publishers (2000)

13. Gloy, N., Smith, M.D.: Procedure placement using temporal-ordering information. ACM Transactions on Programming Languages and Systems **21** (1999) 977–1027
14. De Bus, B., De Sutter, B., Van Put, L., Chanet, D., De Bosschere, K.: Link-time optimization of ARM binaries. In: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. (2004) 211–220
15. Cox, I., Miller, M., Bloom, J.: Digital watermarking. Morgan Kaufmann (2002)
16. Katzenbeisser, S., Petitcolas, F.: Information hiding techniques for steganography and digital watermarking. Artech House (2000)
17. Davidson, R., Myhrvold, N.: Method and system for generating and auditing a signature for a computer program (1996) Microsoft Corporation, US5559884.
18. Pieprzyk, J.: Fingerprints for copyright software protection. In: Information Security, LNCS 1729. (1999) 178–190
19. Holmes, K.: Computer software protection (1991) International Business Machines Corporation, US5287407.
20. Venkatesan, R., Vazirani, V., Sinha, S.: A graph theoretic approach to software watermarking. In: Information Hiding, LNCS. Volume 2137. (2001) 157–168
21. Collberg, C., Thomborson, C., Townsend, G.: Dynamic graph-based software watermarking. Technical report, Dept. of Computer Science, Univ. of Arizona (2004)
22. Curran, D., Cinneide, M.O., Hurley, N., Silvestre, G.: Dependency in software watermarking. In: Information and Communication Technologies: from Theory to Applications. (2004) 569–570
23. Sahoo, T.R., Collberg, C.: Software watermarking in the frequency domain: Implementation, analysis, and attacks. Technical report, Dept. of Computer Science, Univ. of Arizona (2004)