Steganography for Executables

Bertrand Anckaert, Bjorn De Sutter and Koen De Bosschere

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

Abstract. Steganography embeds a secret message in a seemingly innocuous cover-object. This paper presents a framework for steganography with executables as cover-objects built on top of an existing binary rewriter. Executables differ significantly from previously explored cover-objects, we thus first identify the cover-specific redundancies that can be used to embed a secret message. Techniques are then proposed to exploit these redundancies to embed a secret message, obvious steganalytic attacks are identified and countered. As a proof-of-concept the introduced techniques are applied to and evaluated for the IA-32 architecture.

1 Introduction

Information hiding techniques such as watermarking and steganography have known an important surge of interest from both the academic and the commercial community, mainly in the context of media such as image, sound and video. Watermarking aims at embedding a message in a cover-object that identifies the author (or original buyer in the case of fingerprinting) to dissuade piracy. Steganography aims at embedding a message in a seemingly innocuous cover-object and to conceal the existence of that message. As opposed to watermarking this message is not necessarily related to the cover-object. Furthermore a watermark is not necessarily hidden.

Governments say that widely available encryption software could make wiretapping more difficult and therefore they try to restrict the strength of encryption algorithms or require that spare copies of the keys are available for them to seize. While cryptography is about protecting the content of messages, steganography is about concealing their very existence. Citizens seeking privacy could use steganography as an alternative to conceal their communication. Unfortunately these techniques can also be used by criminals to evade laws against unapproved cryptography. Covert channels could also be used to leak confidential information. For these reasons we believe it is important to have a thorough understanding of the potential and weaknesses of steganography for different kinds of cover-objects.

Most of the efforts in information hiding so far have been targeted toward media. The last few years several ways to embed a watermark in a program have been explored and significant advances have been made to increase their

robustness. Steganography for programs is however still a largely unexplored area. The main contributions of this paper are:

- the true potential for diversity of executables is explored;
- three information hiding techniques are applied in the context of steganography;
- these techniques are combined to increase the encoding rate;
- steganalytic attacks are identified and countered.

The remainder of this paper is structured as follows: section 2 discusses related work. Section 3 presents the used model. The usability of executables for steganographic purposes is explored in section 4. Section 5 proceeds with an overview of the used techniques. Steganalytic attacks are illustrated and countered in section 6. A short description of the framework is given in section 7. The discussed techniques are then evaluated for the IA-32 architecture in section 8 and conclusions are drawn in section 9.

2 Related work

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

Digital steganography has mainly been applied to media, such as image, sound and video. A large number of systems has been proposed; many of them have been broken. We refer to [6, 10] for an overview.

Steganography in the context of executables has, to the best of our knowledge, only been addressed by hydan [8], a steganographic tool for IA-32 compatible executables. The key idea is that some instructions can be replaced by equivalent ones and every time such an instruction is encountered, one or more bits are encoded in the choice of one of the equivalent instructions. The tool has an average encoding rate of 1/237 on our benchmark set.

Significantly more research has been conducted in the related field of software watermarking. Given the close interaction between these two fields we will present a short overview of the used approaches. Since our interest lies in the relevance of the proposed techniques for steganographic purposes, we will not follow the usual taxonomy as introduced by Collberg [4]. Rather we will classify them according to the type of redundancy that is exploited to encode the watermark or fingerprint.

Several techniques add an identification to the program by transforming program properties of the existing program. The first one, proposed by Davidson and Myhrvold [7], encodes the watermark in the sequence of basic blocks. Pieprzyk [17] suggests assigning a unique identity to every copy in the choice of equivalent instruction sequences. Another approach encodes the watermark in the frequency of groups of instructions [19]. The frequencies are modified by

subsequently applying code modifications that are accepted if the frequencies of the resulting program are closer to the desired ones and discarded otherwise.

Other approaches add a piece of data [9] or code [12, 13, 20] to the original program that will not be used during execution. Since this data or code will not be used by the program it can easily be constructed to contain the desired watermark.

The third and last class adds code to the original program that will actually be executed by the program. This is done in such a way that it doesn't affect the original functionality. We further distinguish between code that could be removed harmlessly and code that is necessary for the correct execution of the program. Examples of the former are the addition of code that builds a graph in memory that is not relevant to the program [3, 4, 15] and code that encodes the watermark in the addition of redundant computations [5]. An example of a watermark that cannot be removed directly when it has been detected is due to Collberg et al. [2]. With this technique direct jumps are replaced by a function call that redirects the program based on the return address.

3 Model

We will follow the classic model for invisible communication as proposed by Simmons [18]. This model is also known as the "prisoners' problem". Alice and Bob are two prisoners in different cells. All communication between them is arbitrated by a warden named Wendy. The warden will not let them communicate through encryption and will not tolerate any suspicious communication. Both prisoners need to communicate invisibly to develop an escape plan and thus have to set up a subliminal channel.

As in the related field of cryptography, we must assume that the mechanism in use is known to the warden (Kerkhoffs' principle [11]) and so the security must depend solely on a secret key that Alice and Bob somehow managed to share, possibly before their imprisonment.

The general principle of steganography is as follows: when Alice wants to share a secret message with Bob, she 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, resulting in a stego-object s. This must be done in such a way that a third party knowing only the apparently harmless message s, cannot detect the existence of the secret. Alice then transmits s over an insecure channel to Bob and hopes that Wendy will not notice the embedded message. Bob can reconstruct m since he knows the embedding method and has access to the key k used in the embedding process. This extraction process should be possible without the original cover c.

A third person watching the communication should not be able to decide whether the sent objects contain secret messages or not. 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. She defines a test function $f : O \to \{0, 1\}$:

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

This function can make two types of errors: detect a hidden message when there is none (type-I error) and not detect the existence of a hidden message when there is one (type-II error). Most steganographic systems try to maximize the probability that an attacker makes a type-II error. In this paper we will further assume that the warden is passive, i.e. he will not attempt to modify the program, but only to classify the covers.

4 Fitness of executables as cover-objects

Not all digital data are equally suited for use as cover-objects c, since the modifications employed in the embedding process should not be visible to anyone not involved in that process, especially the warden. To enable the embedding of a secret message m, a certain degree of redundancy has to be available in which we can encode bits.

A common used approach in steganography for media consists of identifying redundant bits in the cover-object c, which can be modified without destroying its integrity. Their redundancy is due to the imperfection of our human perception and the presence of noise. These bits are used to encode the secret message m. This approach is not directly transferable to executables, since a change as small as a single bit could cause the entire program to fail. However, when the specific characteristics of software are taken into account many forms of redundancy become apparent.

If we take the definition of equivalent programs of Cohen [1], we consider two programs as equivalent if, given identical input sequences, they produce identical output sequences. In practice, more stringent requirements for time, space and power consumption will be taken into account. Even then a large number of equivalent executables exists. This has been exploited for several purposes:

- program optimization transforms a program into an equivalent faster, smaller or less power-consuming program;
- obfuscation uses the redundancy to produce an equivalent, harder to reverse engineer program;
- watermarking exploits this characteristic to embed an identification into the program;
- software diversity generates a large number of equivalent programs to reduce the vulnerability to widespread faults and attacks;
- ...

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 m. However, to the best of our knowledge, no efforts have been made to get a feel of the true potential for diversity.

To explore this potential we have developed a tool that is capable of exhaustively generating all possible sequences of instructions for the IA-32 architecture. The tool takes a code sequence, a set of registers that contain the output and a set of registers whose value is no longer used after the code sequence. For each generated sequence it checks whether it performs the same function as the original code sequence by testing the output values for all possible input values. If the test succeeds we have found an equivalent sequence.

Because of the exhaustive nature we have limited the number of allowed instructions significantly and restricted the values of immediates to $\{-1, 0, 1, 31\}$. The length of sequences that can be handled within reasonable time-constraints is however still very small. Despite these restrictions we can still find a lot of equivalent sequences that perform realistic computations. To illustrate this we now consider two simple examples.

The registers are assumed to contain signed words. We have generated sequences for the Intel P6 processor family. The first example has the following properties:

function: ECX= max(EAX,EDX)
input: EAX, EDX
output: ECX
scratch: $statusflags$(=CF,PF,AF,ZF,SF,OF)

The tool is able to find 433 different encodings, each representing three instructions that compute the given function. The second example has the following properties:

function: EAX= (EAX/2)
input: EAX
output: EAX
scratch: EDX,$statusflags$(=CF,PF,AF,ZF,SF,OF)

This example returns 3708 equivalent encodings each representing four instructions. We note that the tool did not find shorter possible sequences because of the limited list of immediates that e.g. does not contain 2.

We would like to stress that these examples are no exception. We observe that the number of alternatives increases fast as the number of instructions in the sequence increases. In fact, the number of alternatives 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 alternatives then combined we have at least $a^{\frac{n}{i}}$ alternatives. Many additional alternatives arise when considering the larger piece as a whole.

The number of equivalent programs for any real-life program is thus huge, even more so if we allow the addition of suboptimal or useless code and data. If we would be able to associate each of these alternatives with a separate message we would be able to encode large messages in an executable. Unfortunately, the time consumed by our tool to generate alternatives for small sequences is already several hours and it is thus not practically useful for real-life executables. In the

following section we will set up a more practical method to hide a secret message m.

5 Toward a practically usable implementation

The goal of watermarking is similar to that of steganography: both try to embed a message m into a cover-object c. The techniques used in software watermarking are however not directly transferable to steganography. The main reason is the limited length of the mark embedded in a program for watermarking or fingerprinting purposes: "... a watermark or fingerprint can be encoded in no more than a thousand bits. Much smaller fingerprints will be sufficient in many cases. A 64-bit fingerprint, for example..." [4]. Some approaches, e.g. the one described by Stern [19], require knowledge of the embedded mark in the detection phase. Obviously this is not an option in the context of steganography. Furthermore watermarks are not necessarily hidden, their main goal is to be irremovable. This required resilience against attacks is the main cause for the limited length of the embedded watermark or fingerprint. In our model we assume a passive warden and we thus relax the requirement for resilience against modifications, enabling the embedding of longer messages. On the other hand we have more stringent requirements as to stealth.

As we have seen in section 2 there are many ways to embed additional information in programs. The amount of information that can be hidden while maintaining an equivalent program is virtually unlimited. It is trivial to add code or data, representing the message, that will not be executed or that has no effect on the program. Keeping the warden in mind, we believe it to be more stealthy to embed the message in the existing program. Code that is not executed or of which the computations have no influence on the behavior of the program can be detected by static or dynamic analysis and may arouse suspicion of the warden. To overcome this we have identified three forms of redundancy in programs that can be used to embed a secret message m within reasonable time-constraints.

5.1 Ordering of chains

A chain is basically a set of basic blocks that need to be placed consecutively. A basic block is a set of subsequent instructions for which the following property holds: the flow of execution can only enter the block by the first instruction, execute all the instructions in the block and leave the block through the last instruction. Provided that the necessary information to relocate the code is available, these chains can be positioned at will.

In the context of steganography we would like to encode bits in the chosen ordering. If we have n different chains, we have $n!$ possible orderings and we can thus theoretically embed $log_2(n!)$ bits. In practice the faculty of the number of chains proves to be too big to calculate. Furthermore not all chains are unique.

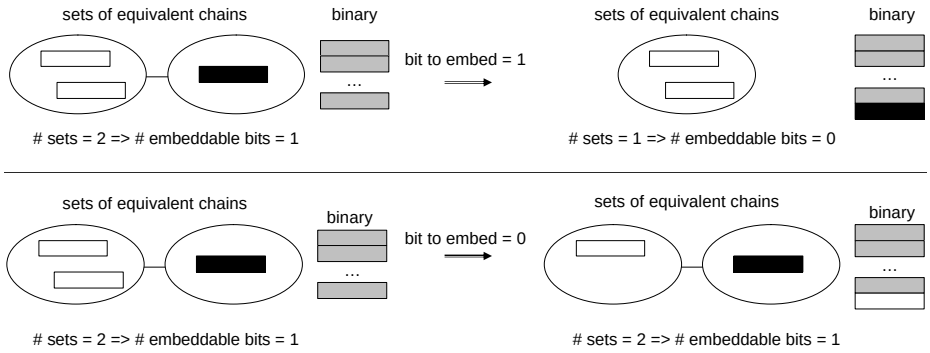

Fig. 1. Encoding bits in the ordering of chains

For these reasons we create a sorted list of sets of identical chains. We then iteratively select a chain for placement. Each time this represents a choice between the remaining sets of identical chains. If their count is c, we can encode $log_2(c)$ bits. Depending on whether the chosen chain was the last of a set of identical chains or not we will have a choice between $c-1$ respectively c chains in the next iteration. This is illustrated in figure 1. It also shows that the number of bits that can be encoded in a program depends on the message that is embedded. In the upper example no additional bits can be encoded because the previous choice eliminated a set, in the latter a choice between two sets remains and we can still embed an additional bit.

For the programs in our benchmark suite the number of sets of identical chains is only between 47 to 59% of the total number of chains. This may seem surprising, because if this is the case, why not remove all but one of a set of equivalent chains. In reality things are a bit more complicated and most of the chains in a set are not truly identical, but we cannot make a distinction between them at the time the ordering is decided. This is due to relocatable operands. These typically identify the address of a function that is to be called or the destination of a jump. Since chains with such a call or jump can be placed before the location of the called function or jump destination is determined, we have no idea what the corresponding operand will be at that time and we have to exclude it from the comparison.

5.2 Ordering of instructions within a basic block

Within basic blocks there is also some degree of freedom in the ordering of instructions. When two or more instructions perform independent operations they can be permuted. Again, we would like to encode bits in this ordering. However we cannot use the straightforward approach as with the ordering of chains since the instructions cannot be ordered independently because of dependencies. Therefore we construct a dependency graph following a strict scheme to assure that the decoder will start with the same dependency graph. Using a branch and

bound algorithm, we generate all the possible permutations. Supposing there are n possibilities, we now know that we can encode $log_2(n)$ bits in the ordering of instructions in the specific basic block and select the permutation that encodes the next $log_2(n)$ bits of the secret message m. We note that the number of possible orderings can be very large for some basic blocks and that we impose an upper limit on the number of considered permutations for timing considerations.

5.3 Equivalent instructions

Substituting instructions by equivalent ones is basically the approach taken by hydan. Information is embedded in the choice of an instruction out of a set of equivalent instructions. If there are e.g. two possibilities, then the first choice would embed a 0 and the second a 1. Hydan has limited knowledge of the program. Since it has no relocation information, it is forced to substitute instructions with instructions of the same length. Furthermore it does not compute liveness information and as a result it can only replace instructions by instructions that perform the same operation in every context (a constraint that is sometimes broken by the tool). Because our tool operates at link-time, these restrictions are eliminated, resulting in more possibilities.

To make sure that no alternatives were forgotten, we used the tool discussed in section 4 in a first phase to find equivalent instructions. We impose the restrictions that only locations read in the original instruction can be read and only locations written by the original instruction can be written. However if liveness determines that certain status flags are dead, i.e. the value they contain will not be used by the program, we allow them to be overwritten. The set of immediates is expanded with the immediates used in the original instruction and the negate thereof. However even this proved to be too time-consuming for practical purposes and thus we built a histogram of the most frequent instructions and their alternatives and made static tables describing these. We found that this has little impact on the number of bits that can be encoded, since the number of instructions for which the tool would find alternatives not included in our tables is small.

Another issue well worth noting is that the number of bits that can be encoded in an instruction is dependent on the chosen ordering of instructions in the basic block. This is another reason why the number of bits that can be encoded in a program depends on the secret message m. We illustrate this with an example: consider the following sequence:

```
INC ECX
CMP EAX,EDX
JB  destination
```

The first instruction increments the value in register `ECX`, and does not change the carry flag. The second instruction compares the values in the registers `EAX` and `EDX`, and sets all condition flags according to the result. The last instruction jumps depending on the value of the carry flag. If we further assume that the

contents of the flags are not used after the jump instruction, then two possible orderings exist for this sequence: the increment and compare instruction can be exchanged since there are no dependencies on live values between them. In the case that the increment is placed before the compare we can substitute it by the following instruction: `ADD 1,%ECX`. This instruction however writes the value of the cary flag and is thus not an alternative in the case that the increment instruction would have been placed after the compare instruction.

5.4 Extraction

The extraction of the secret message m is very similar to the embedding of the message. There are however a couple of issues that need to be addressed appropriately. We need to be able to identify the basic blocks to reconstruct the chains and to build the control flow graph for the computation of liveness information. And we need to be able to identify relocated operands since they cannot be taken into consideration in the ordering of chains. This information is available at the embedding phase, as it is done at link-time, but it is lost in the resulting binary. Fortunately most of the necessary information can be derived from the program itself. We then only need to communicate the discrepancy between the derived information and the actual information to the decoder. For our benchmarks we found that only 5 to 29 basic blocks leaders were not correctly detected. This information is stored 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 he cannot compute liveness information at this point.

Another essential piece of information is which operands are relocated, since we have to eliminate them from consideration with the ordering of chains. To identify these, we implemented a simple heuristic that is used at both the encoding and decoding phase: immediates (or the negates thereof) that are within the address space of the binary are considered to be relocated. Relocated operands that are not identified by this heuristic are communicated to the decoder. This results in a safe assumption: some operands will be considered to be relocated even though they are not, but the only consequence is that its containing chain might be considered equal to another chain where it otherwise wouldn't be.

The last problem on which we will elaborate here is that the ordering of chains is based upon a compare function of those chains, but that the chains are subsequently modified because of the two other techniques to encode information. The same holds for the ordering of instructions in the basic blocks: selecting alternative instructions might change the dependency graph from which this phase departs. To circumvent this problem we transform the basic blocks and the instructions to their first permutation respectively alternative at both the embedding and extraction phase before the chains are sorted and the dependency graph is constructed.

6 Steganalysis and countermeasures

Steganography studies methods of communication in such a manner that the existence of the message m should be undetected. As pointed out earlier we cannot rely on the secrecy of the used algorithm, the safety has to rely solely in some sort of secret key k. When all the details of our embedding method are known, warden Wendy can simply apply the inverse function to extract the message m which will then be visible in plain-text. Therefore we suggest that the message m is encrypted using known cryptographic methods with key k before embedding. This combination of steganography and cryptography clearly increases the security as it is more difficult for an attacker to detect embedded cipher-text as it is not directly readable and it has a rather random appearance. Alternatively the key k could consist of both the ordering function, which determines the outcome of the ordering of chains and instructions within a bbl, and the set and associated encoding of alternative instructions.

Even if the warden cannot directly read the embedded message m he can still suspect its presence. To embed a message modifications are made to the cover program c. In order not to arouse suspicion these distortions should be imperceptible, thus making it difficult to distinguish a cover-object c from a stego-object s.

To create a function f that divides objects into cover-objects and stego-objects some notion of what constitutes a normal program is needed. A program generated by a traditional tool chain will typically be optimized to a certain level and certain suboptimal constructs will not be present. Furthermore certain typical patterns can be identified for common operations. Unusual patterns and suboptimal constructs stand out and expose the possibility of hidden information. Usually, there is a trade-off between encoding rate and stealth, and the discussed techniques are no exception. To hide the presence of a secret message m additional restrictions are imposed and this inevitably leads to a reduction of the number of bits that can be encoded. We will now identify some of these patterns and suboptimal constructs and how they can be avoided.

6.1 Ordering of chains

The freedom in the ordering of chains is o.a. used to improve the code locality, resulting in a better use of the cache and paging system. Specialized optimizations such as the one discussed by Pettis and Hansen [16] try to approximate the optimal placement. But even if no special optimizations are performed the chains of a function will still be grouped, resulting in some level of code locality. When we position the chains in a random order these features will be absent. The average jump offset will be larger and the program will have lesser code locality. All off these properties may arouse the suspicion of the warden.

To overcome this we do not consider all the chains at once, rather, we divide them in smaller groups, e.g. per function and permute only within these smaller groups.

6.2 Ordering of instructions within a basic block

The ordering of instructions within a basic block influences program performance as well and it is thus not surprising that techniques have been developed to approximate the optimal ordering. This is typically done by the scheduler [14]. Some typical patterns will also occur. For example, upon entry to a function the first two instructions will typically save the previous frame pointer and assign it with the value of the stack pointer. Since an attacker will be able to reconstruct most of the control flow graph suboptimal and unusual patterns will stand out.

To circumvent this, we add artificial dependencies in the dependency graph that impose certain orderings and thus prevent some unusual patterns and suboptimal constructs.

6.3 Equivalent instructions

While many instructions may perform the same operation a compiler will typically use only a few of them to perform it. If we e.g. consider the common operation of setting the contents of register `REG` to 0, this will typically be achieved by the instruction `XOR REG,REG`, which is the most efficient in time and space, and sometimes by a `MOV 0,REG`, which has the advantage that it doesn't overwrite the flags as opposed to the former instruction. There are however many other ways to achieve this. We could e.g multiply the content of the register (or any other valid location for that matter) with zero and store it in the register, load the address zero in the register, subtract the register from itself, or and it with zero. Many of these alternatives will not be used in practice because a more efficient alternative is available. A warden could look for such suboptimal instructions to detect the presence of hidden information.

To avoid suboptimal instructions it suffices to prune the tables describing the alternatives for instructions.

We note that these countermeasures provide some level of protection, but it is likely that when the scheme is put into practice other ways of detecting the presence of a secret message will emerge, possibly resulting in an iterative process where new steganalytic techniques require new steganographic techniques and vice versa. This is comparable to the evolution of steganography in the context of media and the interaction of cryptography and cryptanalysis.

7 The Framework

While the examples used in this paper and the evaluation are targeted toward the IA-32 architecture, the proposed techniques are independent of the target architecture. The steganographic tool was built on top of Diablo, a retargetable binary rewriter. Diablo currently supports several architectures, including the IA-32, IA-64, MIPS, ARM and Alpha. The architecture-independent implementation has been separated as much as possible from the architecture-dependent implementation. As a result the creation of a steganographic tool for one of the

other supported architectures requires the implementation of only two functions: a function that compares two instructions and a function that finds alternatives, given the original instruction, output values and over-writable values.

It might be interesting to explore the potential of architectures with different characteristics. A RISC architecture for example is unlikely to have the same redundancy in its instruction set as the IA-32 architecture. On the other hand, due to the fixed instruction length, some bits of an instruction are not used and can thus be used to embed information.

The tool is available at http://obviousreference.

8 Evaluation

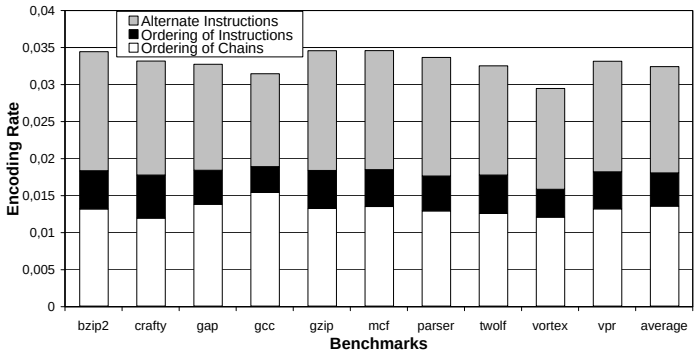

Fig. 2. Encoding rate before countermeasures

To evaluate our steganographic tool we used the SPECint2000 benchmark suite. We compiled using the GCC compiler (v3.2.2) and linked them with the glibc library (v2.3.2). The experiments were conducted on a Intel Pentium 4 processor running Linux 2.4.21. The number of allowed orderings for instructions within a basic block was limited to 1024. The embedding and extraction of messages takes less than a minute for all benchmarks; we believe this to be acceptable. The chosen text for evaluation purposes is is the unencrypted "King Lear" by William Shakespeare. The encoding rate before the introduction of countermeasures for steganalysis is illustrated in figure 2. The distribution over the different techniques is also indicated. We achieve an encoding rate between 1/34.02 and 1/28.99 and an average of 1/30.91, an improvement of almost an order of magnitude over the previous tool hydan, which has no countermeasures for steganalysis either.

Figure 3 shows the results after countermeasures for steganalysis were taken. We can see that the importance of the ordering of instructions within basic blocks increases as it is less affected by the imposed restrictions. Here we can

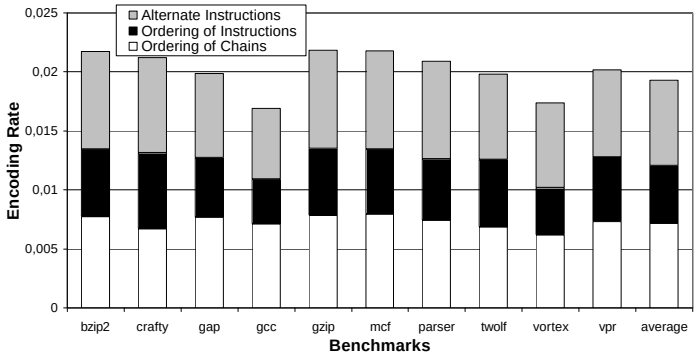

Fig. 3. Encoding rate after countermeasures

see an encoding rate ranging between 1/59.20 and 1/45.76 and an average of 1/51.84.

9 Conclusion

This paper has illustrated the potential for diversity of executables and how the redundancy in executables can be exploited to embed a secret message within reasonable time-constraints and with an average data rate of 1/30.91. Some steganalytic attacks have been identified and countermeasures were adopted, which reduced the average data rate to 1/51.84.

Acknowledgments

The authors would like to thank the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT), the Fund for Scientific Research - Belgium - Flanders (FWO) and the University of Ghent for their financial support.

References

1. F. Cohen. Operating system evolution through program evolution, 1992.
2. C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *Programming Language Design and Implementation*, 2004.
3. C. Collberg, S. Kobourov, E. Carter, and C. Thomborson. Error-correcting graphs for software watermarking. In *Workshop on Graph Theoretic Concepts in Computer Science*, 2003.
4. C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles Of Programming Languages*, pages 311–324, 1999.

5. P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Principles Of Programming Languages*, 2003.
6. I. Cox, M. Miller, and J. Bloom. *Digital watermarking*. Morgan Kaufmann Publishers Inc., Pine Street, San Fransisco, 2002.
7. R. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program, 1996. Microsoft Corporation, US Patent 5559884.
8. R. El-Khalil. Hydan, information hiding in program binaries, 2003. http://www.crazyboy.com/hydan/.
9. K. Holmes. Computer software protection, 1991. International Business Machines Corporation, US Patent 5287407.
10. S. Katzenbeisser and F. Petitcolas. *Information hiding techniques for steganography and digital watermarking*. Artech House, Inc., Canton Street 685, Norwood, MA 02062, 2000.
11. A. Kerkhoffs. La cryptographie militaire. *Journal de Sciences Militaires*, 9:5–38, 1883.
12. A. Monden, H. Iida, K. ichi Matsumoto, K. Inoue, and K. Torii. Watermarking java programs. In *International Symposium on Future Software Technology*, pages 119–124, 1999.
13. A. Monden, H. Iida, K. ichi Matsumoto, K. Inoue, and K. Torii. A practical method for watermarking java programs. In *Computer Software and Applications Conference*, pages 191–197, 2000.
14. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997.
15. J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Annual Computer Security Applications Conference*, pages 308–316, 2000.
16. K. Pettis and R. Hansen. Profile guided code positioning. In *Programming Language Design & Implementation*, pages 16–27, 1990.
17. J. Pieprzyk. Fingerprints for copyright software protection. In *Information Security Workshop*, pages 178–190, 1999.
18. G. J. Simmons. The prisoners' problem and the subliminal channel. In *Advances in Cryptology*, pages 51–67, 1984.
19. J. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In *Information Hiding Workshop*, pages 368–378, 1999.
20. R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *Information Hiding Workshop*, pages 157–168, 2001.