Hydan: Hiding Information in Program Binaries

Rakan El-Khalil and Angelos D. Keromytis

Department of Computer Science, Columbia University in the City of New York
{*rfe3,angelos*}*@cs.columbia.edu*

Abstract. We present a scheme to steganographically embed information in *x86* program binaries. We define sets of *functionally-equivalent instructions*, and use a key-derived selection process to encode information in machine code by using the appropriate instructions from each set. Such a scheme can be used to watermark (or fingerprint) code, sign executables, or simply create a covert communication channel. We experimentally measure the capacity of the covert channel by determining the distribution of equivalent instructions in several popular operating system distributions. Our analysis shows that we can embed only a limited amount of information in each executable (approximately $\frac{1}{110}$ bit encoding rate), although this amount is sufficient for some of the potential applications mentioned. We conclude by discussing potential improvements to the capacity of the channel and other future work.

1 Introduction

Traditional information-hiding techniques encode ancillary information inside data such as still images, video, or audio. They typically do so in a way that an observer does not notice them, by using redundant bits in the medium. The definition of "redundancy" depends on the medium under consideration (cover medium). Because of their invasive nature, information-hiding systems are often easy to detect, although considerable work has gone into hiding any patterns [1]. In modern steganography, a secret key is used to both encrypt the information-to-be-encoded and select a subset of the redundant bits to be used for the encoding process. The goal is to make it difficult for an attacker to detect the presence of secret information. This is practical only if the cover medium has a large enough capacity that, even ignoring a significant number of redundant bits, we can still encode enough useful information.

Aside from its use in secret communications, an information-hiding process [2] can be used for watermarking and fingerprinting, whereby information describing properties of the data (*e.g.,* its source, the user that purchased it, access control information, *etc.*) is encoded in the data itself. The "secret" information is encoded in such a manner that removing it is intended to damage the data and render it unusable (*e.g.,* introduce noise to an audio track), with various degrees of success.

In this paper, we describe the application of information-hiding techniques to arbitrary program binaries. Using our system, named Hydan, we can embed information using *functionally-equivalent instructions* (*i.e., i386* machine code instructions). To determine the available capacity, we analyze the binaries of several operating system distributions (OpenBSD 3.4, FreeBSD 4.4, NetBSD 1.6.1, Red Hat Linux 9, and Windows

XP Professional). Our tests show that the available capacity, given the sets of equivalent instructions we currently use, is approximately $\frac{1}{110}$ bits (*i.e.*, we can encode 1 bit of information for every 110 bits of program code). Note that we make a distinction between the overall program size and the code size. The overall program size includes various data, relocation, and BSS sections, in addition to the code sections. Experimentally, we have found that the code sections take up 75% of the total size of executables, on average. For example, a 210KB statically linked executable contains about 158KB of code, in which we can embed 1.44KB ($11,766$ bits) of data.

In comparison, other tools such as Outguess [1] are able to achieve a $\frac{1}{17}$ bit encoding rate in images, and are thus better suited for covert communications, where data-rate is an important consideration. The $\frac{1}{110}$ encoding rate achieved by the currently implemented version of Hydan is obtained when we only use instruction substitutions. In Section 5 we discuss improvements that may lead to a $\frac{1}{36}$ encoding rate.

This capacity can be used as a covert steganographic channel, for watermarking or fingerprinting executables, or for encoding a digital signature of the executable: we can encode an all-zeroes value in the executable, a process that modifies the code without damaging its functionality, sign the result, and then encode the signature by overwriting the same "zeroed-out" bits. Signature verification simply extracts the signature from the binary, overwrites it with an all-zeroes pattern, and finally verifies it. This signature can be a public- or secret-key based one (MAC). Apart from the fact that this information is hidden (and can thus be used without the user's knowledge or consent – admittedly, a worrisome prospect), one advantage of this approach is that the overall size of the executable remains unmodified. Thus, it may be particularly attractive for filesystems that provide functional-integrity protection for programs without increasing their size, nor rely on an outside database of hashes.

Paper Organization In the remainder of this paper, we briefly examine prior work in both classical and code steganography (Section 2), describe our approach (Section 3), and experimentally measure the capacity of this new medium by examining a large number of program binaries (Section 4). We discuss the weaknesses of our approach and potential ways to overcome them in Section 5.

2 Related Work

Unlike the medium of sound and image, data hiding in executable code has not been the subject of much study. One hindering particularity of the machine code medium is the inherently reduced redundancy encountered, a redundancy that information hiding depends on to conceal data. Most of the previous work on executable code was therefore done at the source code or compilation level. Our work differs in that we embed data at the machine-code level, without need or use of the source code. Here, we present an overview of research in both classical and code information-hiding.

Classical Information-Hiding Petitcolas *et al.* [3] classify information hiding techniques into several sub-disciplines: creation of covert channels, steganography, anonymity, and copyright marking. Each of those fields has specific, and often overlapping requirements. Steganography, literally the art of "covered writing," has the requirement of

being imperceptible to a third party. Unlike copyright marking for example, steganography does not have to resist tampering; the focus is to subliminally convey as much information as possible. In contrast, copyright marking need not always be hidden, and some systems use visible digital watermarks. Despite the opposite requirements, they are both a form of information hiding. A general information-theoretic approach to information hiding and steganography can be found in [4, 2], forming the basis for the design of such systems.

Executable Code Steganography Despite the relative lack of work on code steganography, there are a few general techniques that have been developed, and at least four patents issued [5–8]. Those techniques have primarily been geared towards software protection and watermarking. Following the classification introduced in [9], the techniques can be divided into two general categories: static and dynamic watermarking.

Static watermarks are embedded at compilation time and do not change during the execution of the application. Verifying the watermark becomes a matter of knowing where and what to look for. Examples include:

- Static data watermarks:
  - `const char c[] = "Copyright (c)..";` or
  - `const struct wmark = {0x12, 0x34, ... };`[5]
- Code watermarks, which depend on characteristics such as the order of `case` statements, the order of functionally independent statements, the order of pushing and popping registers on the stack [10], or the program's control flow graph [6].

In dynamic watermarking, the user executes the program with a specific set of inputs, after which it enters a state that represents the watermark. Some examples of these are:

- "Easter Eggs," where some functionality (such as a piece of animation) is only reachable when a secret sequence of keystrokes is entered.
- Dynamic data-structure watermark, where the content of a data structure changes as a program executes. The end-state of the structure represents the watermark [9].
- Dynamic execution-trace watermark is similar to the above, but uses execution traces [9]. The order and choice of instructions invoked constitute the watermark.

All of these techniques are only applicable if the source code is given, or obtained as a result of de-compilation, to the person performing the data-hiding. However, ease of de-compilation varies greatly across different programming languages. In theory, the differentiation of data and code in von Neumann machines reduces to the Halting Problem, and therefore perfect disassembly is impossible. Thus, in practice we are forced to settle for a less than perfect solution, often relying on heuristics that can sometimes fail. Those heuristics have varying degrees of success depending on the language the application was coded in. Java and .Net are much simpler to disassemble than *x86 asm* for example. In the Java case, significant amounts of meta-data is inserted into its classes, and JVM code must pass a stringent verification process before it is run. Under these correctness constraints, it becomes easy to accurately decompile Java byte-code.

Because *x86* code does not have such constraints, it is notoriously difficult to disassemble. As such, most of the research on code watermarking has been executed with Java in mind, and implemented for Java byte-code. One work was specifically developed for *x86* code [11] and outlines a spread-spectrum technique to embed watermarks.

This scheme was subsequently implemented for Java byte-code in [12]. Our work differs in that it was researched, and implemented, for *x86* code.

It is also worth noting that programmers have historically embedded "signatures" into the assembly code of their hand-coded Z80 and 6502 programs. The first tool to do this automatically was the A86 assembler, which embedded a signature into the code it produced by choosing between equivalent instructions to output, for registration purposes. As in the previous techniques, access to the original source is required.

3 Architecture

Hydan takes a message and an executable (covertext) as input, and outputs a functionally identical executable that contains the steganographically embedded message. We use the inherent redundancy in the machine's instruction set (*e.g.*, the i386 processor family instruction set) to encode the message, as several instructions can be expressed in more than one way. For example, adding the value *50* to register *eax* can be represented as either "`add %eax, $50`" or "`sub %eax, $-50`".

Using these two alternate forms, we can encode one bit of information anytime there is an addition or a subtraction in the executable code. Another example is that of XORing a register against itself to clear its contents: subtracting the register from itself has the same effect. For example, consider the code on the left column of Table 1. Using the `add/sub` substitution, we can encode 2 bits in that code. By convention we decide that all addition instructions represent bit 0, and subtraction instructions represent bit 1. Thus, encoding the binary values `00`, `01`, and `11` would yield the functionally identical code shown in Table 1.

Original code			*Encoding 00*		
`83 e8 30`	`sub`	`%eax, $0x30`	`83 c0 d0`	`add`	`%eax, $-0x30`
`83 f8 36`	`cmp`	`%eax, $0x36`	`83 f8 36`	`cmp`	`%eax, $0x36`
`77 e5`	`ja`	`$-27`	`77 e5`	`ja`	`$-27`
`83 c0 08`	`add`	`%eax, $0x8`	`83 c0 08`	`add`	`%eax, $0x8`
`89 04 24`	`mov`	`%eax, [%esp]`	`89 04 24`	`mov`	`%eax, [%esp]`
Encoding 01			*Encoding 11*		
`83 c0 d0`	`add`	`%eax, $-0x30`	`83 e8 30`	`sub`	`%eax, $0x30`
`83 f8 36`	`cmp`	`%eax, $0x36`	`83 f8 36`	`cmp`	`%eax, $0x36`
`77 e5`	`ja`	`$-27`	`77 e5`	`ja`	`$-27`
`83 e8 f8`	`sub`	`%eax, $-0x8`	`83 e8 f8`	`sub`	`%eax, $-0x8`
`89 04 24`	`mov`	`%eax, [%esp]`	`89 04 24`	`mov`	`%eax, [%esp]`

Table 1. Encoding the values `00`, `01`, and `11` using equivalent instructions (highlighted).

Another feature used to encode data is that several instructions have two forms, *e.g.*, "`insn r/m, reg`" and "`insn reg, r/m`", where `reg` is a register, and `r/m` can be either a register or a memory location. However, if we consider the case where `r/m` points to a register, then we can encode the instruction using either form. All we need to do is change the opcode, and swap the `reg` and `r/m` values in the instructions so that they point to the correct operands.

We can sometimes encode more than one bit per instruction by using as many as possible equivalent instructions, since we can embed $\log_2(n)$ bits when using a set of n functionally equivalent instructions. For a set of four instructions, any instruction in that set can be used to embed two bits of data. The sets we found usually contain two or four instructions. In two cases, we were able to find seven-instruction sets.

However, seven instructions are not enough to encode three bits of data, and too many to encode only two bits. In order to avoid having to use only four instructions and waste the other three, we devised an encoding scheme whereby we use one instruction from the set as a wildcard. This instruction is used when we encounter a value we cannot directly represent with the current set of instructions. For example, using a 7-instruction set, we can encode the binary values 0 through 5. However, we cannot encode the values 6 to 8. So we use the wildcard instruction to signify that this instruction does not encode any data, and try to embed the missing values with the next instruction. What will most likely happen is that those values (6–8) will be broken down into smaller chunks of one or two bits and encoded that way. For our example of seven instructions, we can embed $\log_2(6) = 2.58$ bits, instead of just 2. In general, we can encode $\log_2(n-1)$ bits in a set of n instructions, when n is not a power of two. This works well under the assumption that the message has equal distribution of values. To ensure this (as well as further secure the message, as is common practice in steganographic applications), we encrypt the data before embedding it and hence achieve nearly uniformly distributed binary data. We assume that ciphertext produced by a good block encryption algorithm, such as AES or Blowfish, has this property.

For simplicity, we chose to only consider replacement instructions of the same size. Indeed, if we were to replace an instruction with a functional equivalent of a larger size, such as two or more instructions, then we would need to shift all subsequent jump target and function call addresses, as well as data locations and their references by the difference in size. This is not infeasible, but significantly complicates the encoding process, especially as we need to be particularly careful with *x86* disassembled code since it may not always be accurate. A nice result of our choice is that the program code size remains the same.

Furthermore, although all of the instructions in each set have the same end result, they may exhibit different side-effects, *e.g.*, set some of the processor flags differently. For example, addition and subtraction set the carry and overflow flags differently in some cases. Hydan thus takes into account any flags that a replacement instruction might have an adverse effect on, and scans through the instructions following it to see if the flag differences might have an effect on execution flow. What we do is look at each instruction following the replacement instruction, and see whether it tests for one of the modified flags. We continue this way, following execution flow, until we either hit the end of the current function, or an instruction that modifies the value of one of the flags we are tracking. If all of the flags are modified by other instructions before they are tested, it is safe to replace the instruction. Otherwise, the instruction is not used for embedding, and left unmodified. Fortunately, in practice such instructions are rare (less than 0.2% of the total), and the bandwidth lost negligible.

The Embedding Process The embedding process itself is straightforward. Upon reading the message to be encoded and the corresponding covertext, Hydan asks the

user for a key to encrypt the message with. Hydan then prepends the size of the message to the message proper, and encrypts the resulting data with Blowfish in CBC mode.

The length of the message needs to be embedded for decoding purposes, but is encrypted to avoid being used as a means to detect the presence of hidden data in the binary. The length is a 64-bit value, and as such most of its MSBs will typically be all-zeroes, which could facilitate cryptanalysis (the attacker can use that information to mount a dictionary attack against the Blowfish session key, which is derived from a user-supplied passphrase). We therefore XOR this length with a hash of the user-supplied passphrase before encrypting it, as a whitening step.

Once the encryption step is completed, Hydan determines the locations of instructions in all code sections of the executable which can be used for embedding. For example, ELF executables can have multiple executable code sections, while *a.out* and *PE/COFF* executables only have one such section. Starting from the first code section, we embed bits following a random walk by skipping a random amount of instructions before embedding anything. This random walk is seeded by the user-supplied key as well, and its purpose is to increase the workload of any detection attempts. We use the technique described in [1] to spread the embedded bits uniformly in the covertext: the number of bits we skip is in the range [0, $2 \times \frac{\omega_c}{\omega_m}$], where ω_c is the number of bits remaining in the covertext, and ω_m the remaining length of the message. This interval is updated each time 8 bits of message are embedded.

The Decoding Process To extract the message, Hydan uses the user password to seed the random-walk algorithm, and extracts the size of the embedded data first. This size is decrypted, and Hydan then proceeds to extract the relevant amount of data from the covertext. Care is taken to keep the random-walk intervals and other variables identical to those obtained in the embedding process by using the same techniques.

4 Analysis

Although Hydan currently does not attempt to respect the statistical distribution of instructions when embedding a message, it is of interest to see what distribution those instructions have in the 'wild,' as any large deviation in instruction distribution can provide an easy means of detection of secret information.

We analyzed the executables in some readily available operating system distributions (OpenBSD 3.4, FreeBSD 4.4, NetBSD 1.6.1, Red Hat Linux 9, and Windows XP Professional) and recorded the number of instances of each instruction we have equivalents for. We then calculated the distribution of each instruction within their sets, as well as the distribution of each set globally. We give the results for OpenBSD in Appendix A; the other operating systems exhibit a similar instruction distribution.

The first column in the table is an identifier for the set of equivalent instructions. The instructions themselves are present in the second column. Looking at the set *xor32-1* for example, we can see that "`xor r32, r/m32`" is equivalent to three other instructions, one of which is "`sub r32, r/m32`." In the case of additions and subtractions, the regular "`add register, imm`" refers to the instance where *imm* is a positive number, whereas the negative form refers to a negative immediate value. See Appendix A for details about the most relevant sets of instructions.

Examining the data, we can readily see that not all instructions are created equal. In fact, it is often the case that only one instruction in each set is overwhelmingly present in the wild. It is therefore quite easy to detect Hydan, especially as the message size increases. Some compilers even insert a tag into the `.comment` section, making it easier for an attacker to know what distribution to expect. To make detection of Hydan harder, we could limit ourselves to using those sets of instructions that have a more amenable distribution in the wild, and respecting their own internal distributions with an encoding scheme. The drawback is that the encoding rate would greatly suffer, as the amount of information we can encode cannot exceed the frequency of the rarest instruction in each set. For example, the OpenBSD distribution drops by $\frac{1}{5099}$th if we want to stealthily embed data. The other distributions are more amenable to stealth however, as shown in Table 2.

OS	*Original Encoding Rate*	*Stealthy Encoding Rate*	*KB per bit*
`OpenBSD`	$\frac{1}{106}$	$\frac{1}{5099}$	66.0
`FreeBSD`	$\frac{1}{104}$	$\frac{1}{4823}$	61.2
`NetBSD`	$\frac{1}{95}$	$\frac{1}{846}$	9.81
`Windows XP`	$\frac{1}{137}$	$\frac{1}{74}$	1.24

Table 2. Original and Stealthy Encoding Rates

As is evident, it is difficult to stealthily encode substantial amounts of data with instruction substitution alone. However, embedding short messages such as keys and signatures is still feasible, especially for NetBSD, Linux, and Windows XP. We describe some techniques for improving the stealthiness of the encoding process in Section 5.

5 Further Discussion

The strength of any information-hiding system is a function of its data rate (the amount of information that can be embedded in a covertext of a given size), stealth and resilience [9]. In this section, we describe Hydan's currently implemented characteristics, as well as ways to improve them.

Data Rate Our current embedding rate is an average of $\frac{1}{110}$, *i.e.*, we can embed, on average, 1 bit of information per 110 bits of code. Since we currently only use the sets of equivalent instructions to embed messages, the data rate is highly dependent on what instructions are present in the executables. Our analysis shows that the distribution of instructions is very similar in most binaries on a given operating system, and even across UNIX operating systems: the OpenBSD, FreeBSD, NetBSD, and Red Hat distributions are only different by a few percentage points. This is easily explainable: they all make use of the same compiler (GCC, various versions). The Windows XP distribution is the most different as different compilers are used. Figure 1 shows the repartition of bandwidth amongst the different instruction sets.

There are several ways to improve the data rate. One approach is to simply find more sets of functionally equivalent instructions, especially if we disregard our restriction on

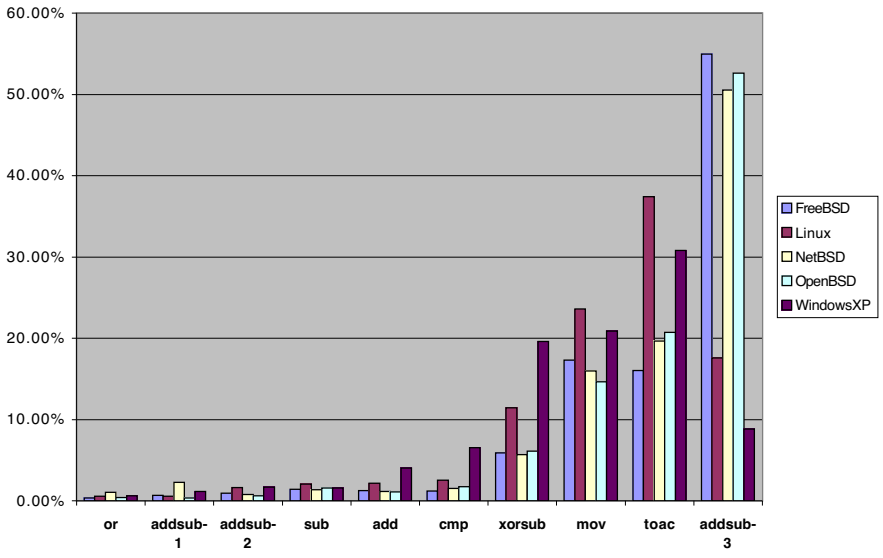

Fig. 1. Instruction Distribution

maintaining the size of the executable. However, replacing a single instruction with two or more equivalent instructions would be suspiciously inefficient from the compiler's point of view. Detecting such a sequence would alert an attacker to the presence of hidden information. On the other hand, compiled code is not always optimized as much as it could be, and a run through GCC compiled code revealed several inefficiencies that can be used to encode data. One of them is the use of multiple additions and subtractions when one would suffice.

We could replace one of the instructions with the net operation, and use the other one as a bogus instruction solely for the purpose of encoding data. Or we could use this obviously redundant sequence itself to encode data: we could keep both operations, but modify whether they were additions or subtractions, which instruction the larger immediate value is placed into, and the relative sizes of the immediate operands. All told, a maximum of 34 bits could be embedded into these two instructions alone (two in the choice of operation, one for the position of the larger immediate value, and 31 into the difference in size). One can also simply swap instructions that are independent of each other, by building a codebook of such groups of instructions, as is done in [11].

Analyzing the binary's control flow graph yields two other ways we can improve the data rate. The simplest approach is to identify code areas that are never executed (dead-code analysis). The other approach is to identify functionally independent code blocks and reorder them.

Dead-code analysis is simple to perform as we would simply tag every instruction that is ever reachable – regardless of input – by recursively following every call and jump in the code. The instructions that are left can be modified without fear of changing the functional equivalence of the executable. This method would only yield the

minimum amount of dead code possible, as there is probably more dead code depending on input and other factors. One way to increase the amount of dead code available for use by Hydan is to use statically linked binaries, as the whole of the library is linked in. In many cases, most of the code included in the library is not used. Once we have located the dead code, we can replace it with assembly mimicry to encode our message.

Identifying functionally independent code blocks would allow us to reorder them in the executable. For example, the ordering of functions is defined by the ordering of their declaration in the source code, and by the order in which the object files were assembled. We can thus reorder the location of those functions in the assembly code without changing the functionality of the code. One method is to consider each reorderable block as a number. The sorted list of numbers represent the zero-state. Assuming we have N reorderable blocks, the number of bits we can encode is $N_{bits} = floor(log_2(n!))$. We take N_{bits} bits of input, and decompose that number along the factorials of $(N-1, ..., 1)$. For example, if $N = 5$, then $N_{bits} = 6$. If the next 6 bits of input are 110110 (decimal 54), its decomposition is: $2*4!+1*3!+0*2!+0*1!$. Each factor now refers to the index of an item in the sorted sublist directly following it. If we place each item in the list one by one according to those indices, we obtain a re-ordered list with an encoded N_{bits} of data. Continuing the example, if $N = 5$, the sorted list could be: "abcde", and the factors are 2, 1, 0, 0. The first factor means that the third item in the list should be at the first position in the encoded list. So we now have the list "cabde". Next, we consider only the sublist "abde", having taken care of the first list item. According to the factors, we are now to have the second item head the list, thus obtaining: "bade". The next two items remain untouched as they are where they need to be. Our encoded list is therefore: "cbade". Decoding this list follows a similar process, where we construct the factors by looking at the relative positioning of each list item according to the list that follows it. Following this process, we can reliably encode a bit-stream into an ordered set. Two similar approaches have been described in [13, 14]. Our measurements show that there are on average 315 functions in each executable. Thus, we can increase our encoding rate from $\frac{1}{110}$ to $\frac{1}{80}$ by using function reordering in conjunction with instruction substitution.

There are several other elements in an executable that we can reorder, such as arguments to functions, the order with which elements are pushed and popped off the stack [10], the register allocation choice, the *.got*, *.plt*, *vtables*, *.ctors*, and *.dtors* tables. We can also reorder the data in the *.data*, *.rodata*, and *.bss* sections. Most of these sections are specific to ELF, but there are equivalent data structures in *a.out* and *PE/COFF*. By counting the number of such entries in typical binaries, we estimate that ordering the functions, and the *.got, .plt, .ctors, .dtors* tables alone would yield us an encoding rate of $\frac{1}{36}$. This is a significant improvement over the instruction-substitution technique.

By reordering the data sections, we can further improve the encoding rate, as there are typically many more data objects than there are function calls or table entries. However, it is difficult to accurately determine the bounds of data blocks when we only have access to the program binary. Compiler output being predictable, we can however attempt to determine those bounds heuristically. This is an area for future research.

Stealth Since the instruction distribution is fairly uniform across executables, it is very difficult to embed any large amount of data while avoiding detection when using

single instruction substitution. In fact, we would only be able to embed the smallest amounts of data as some of the replacement instructions almost never appear in the wild. Thus, unless used to embed signatures, keys or other small amounts of data, we believe that the single instruction substitution method is not suited for stealth. The additional encoding methods described in the previous section have, however, a much larger stealthy bandwidth. For example, replacing dead-code sections with assembly mimicry is a quite powerful and easy to implement method. In that case, stealthiness is limited only by the intelligence of the mimicry algorithm. Typical algorithms use context-free grammars (CFGs) to determine mimicry patterns, and as such their theoretical security is based on the fact that there are no known polynomial-time algorithms that predict membership in a class defined by CFGs [13].

Any form of ordering described above is also very stealthy, as this order is not platform dependent (except perhaps for the `push/pop` order, where IBM used this order to claim a signature for their PC-AT ROM when litigating against software pirates [10]). The ordering is otherwise determined mainly by the source code, which is different for every executable, and therefore provides a good source of "controlled" randomness that we can exploit to encode information.

There is one caveat: generally, there are not nearly as many executables as there are data files (*e.g.*, music or image files). It would thus be easier to detect hidden data by cataloging as many possible instances of executables, and checking for differences. Furthermore, it would be fairly easy to check statically linked binaries for different function orderings, since there is a limited number of versions of any particular library, *e.g., libc*, to compare against. However, if this technique is used as software watermarking for registration purposes, where a signature is embedded into the executable in order to identify its owner in case of piracy, every executable's hash signature will be different.

Resilience As was observed in [15], achieving protection of large amounts of embedded data against intentional attempts at removal may ultimately be infeasible. The techniques we have described rely on statically embedding data into a binary executable. The location of this data is easy to find and modify, since we only embed into specific instructions. In fact, there is no way to protect against overwriting all potentially hidden data, by randomly permuting any instruction or other aspect of the binary that could be used by Hydan. If we expect a message to be encoded in a binary (*e.g.*, in a watermark/fingerprint, or a digital signature), we can easily detect corruption by appending a message authentication code to the encoded data. We could also use ECC to try to recover parts of the message in case of overwrites. But a removal attempt does not cripple the software. A potentially more fruitful technique would be to use the dynamic watermarking techniques, as described in [9]. The implementation of such techniques would be more difficult in *x86,* as they require access to source code to be effective. This is an area of future research.

6 Concluding Remarks

We presented Hydan, a system for embedding data in *x86* program binaries by using functionally equivalent instructions as redundant bits. We analyzed the program binaries of the OpenBSD, FreeBSD, NetBSD, Red Hat Linux, and Windows XP Professional

operating systems to estimate the encoding rate of such a system. We discussed our implementation of Hydan and its resistance to discovery and removal attacks. Applications include steganographic communication, program registration, and filesystem security.

We determined that we can embed 1 bit for every 110 bits of program code. In comparison, standard steganographic techniques using JPEG as the cover medium can achieve an encoding rate of $\frac{1}{17}$, due to the large amount of redundant bits in typical images. However, we have identified several potential improvements to Hydan that may lead to an encoding rate of $\frac{1}{36}$ in program binaries.

Our plans for future work include finding new techniques to increase the capacity of the program binary cover, and investigating the use of of dynamic watermarking techniques in machine code. The Hydan implementation can be freely downloaded from:

`http://www.crazyboy.com/hydan/`

Acknowlegements We are greatly indebted to Michael Mondragon for *The Bastard* disassembler, *libdisasm*, and much guidance. Thanks to El Rezident for the PE/COFF parsing code. We also thank Josha Bronson and Vivek Hari for their help in obtaining the FreeBSD and WindowsXP instruction statistics. Niels Provos provided valuable feedback while this work was in progress.

References

1. Provos, N.: Defending Against Statistical Steganalysis. In: Proceedings of the 10th USENIX Security Symposium. (2001)
2. Cachin, C.: An Information-Theoretic Model for Steganography. LNCS **1525** (1998) 306–318
3. Petitcolas, F.A.P., Anderson, R.J., Kuhn, M.G.: Information hiding — A survey. Proceedings of the IEEE **87** (1999) 1062–1078
4. Moulin, P., O'Sullivan, J.: Information-theoretic analysis of information hiding (1999)
5. Samson, P.R.: Apparatus and method for serializing and validating copies of computer software. US Patent 5,287,408 (1994)
6. Davidson, R.L., Myhrvold, N.: Mehod and system for generating and auditing a signature for a computer program. US Patent 5,559,884 (1996)
7. Moskowitz, S., Cooperman, M.: Method for stega-cipher protection of computer code. US Patent 5,745,569 (1996)
8. Holmes, K.: Computer software protection. US Patent 5,287,407 (1994)
9. Collberg, C., Thomborson, C.: On the Limits of Software Watermarking. Technical Report 164, Department of Computer Science, The University of Auckland (1998)
10. Council for IBM Corporation: Software birthmarks. Talk to BCS Technology of Software Protection Special Interest Group (1985)
11. Stern, J.P., Hachez, G., Koeune, F., Quisquater, J.J.: Robust object watermarking: Application to code. In: Information Hiding. (1999) 368–378
12. Hachez, G.: A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards (2003)
13. Wayner, P.: Disappearing Cryptography. 2nd edn. Morgan Kaufmann, San Francisco, California (2002)
14. Kwan, M.: gifshuffle. `http://www.darkside.com.au/gifshuffle/` (2003)
15. Bender, W., Gruhl, D., Lu, A.: Techniques for data hiding. IBM Systems Journal **35** (1996)

Appendix A: Instruction Statistics

The folowing is a description of the more important sets of equivalent instructions. For clarity, the 8 and 32-bit versions of instruction sets are shown on the same line, and the lesser-used instructions are omitted from the table. Instructions in each set are only equivalent, and used by Hydan, if they follow certain constraints:

- The following instruction sets' operands must both point to registers:
  - `add[8, 32]`, `and[8, 32]`, `add[8, 32]`, `cmp[8, 32]`, `mov[8, 32]`, `or[8, 32]`, `sbb[8, 32]`, `sub[8, 32]`, `xor[8, 32]`
- `addsub[8, 32][-1, -2]`: The addsub sets all refer to the equivalence of addition with negative subtraction. There are several such sets since instructions differ depending on the size of their operands. The "negative form" represents instances where the immediate value is negative.
- `toac[8, 32]`: The listed instructions have the same effect (namely, set the flag register according to the result) when their arguments are identical, *e.g.*, "`test %eax, %eax`"

r/m	register/memory
r[8,32]	register
imm[8,32]	immediate value

Table 3. Statistics legend.

Set of Instructions	Instruction Name	Bits	Dist w/in Set	Global Dist
add32	`add r/m32, r32`	31909	100.00%	1.06%
	`add r32 , r/m32`	0	0.00%	
addsub32-1	`add eax, imm32`	4576	84.66%	0.18%
	negative form	547	10.12%	
	`sub eax, imm32`	282	5.22%	
	negative form	0	0.00%	
addsub32-2	`add r/m32, imm32`	7356	44.89%	0.55%
	negative form	1470	8.97%	
	`sub r/m32, imm32`	7560	46.14%	
	negative form	0	0.00%	
addsub32-3	`add r/m32, imm8`	960798	60.86%	52.60%
	negative form	509372	32.27%	
	`sub r/m32, imm8`	108266	6.86%	
	negative form	174	0.01%	

Set of Instructions	Instruction Name	Bits	Dist w/in Set	Global Dist
addsub8	`add al , imm8`	1698	35.26%	0.16%
	negative form	1041	21.62%	
	`sub al , imm8`	2076	43.12%	
	negative form	0	0.00%	
and32	`and r/m32, r32`	2683	100.00%	0.09%
	`and r32 , r/m32`	0	0.00%	
and8	`and r/m8 , r8`	70	100.00%	0.00%
	`and r8 , r/m8`	0	0.00%	
cmp32	`cmp r/m32, r32`	49784	100.00%	1.66%
	`cmp r32 , r/m32`	0	0.00%	
cmp8	`cmp r/m8 , r8`	1302	100.00%	0.04%
	`cmp r8 , r/m8`	0	0.00%	
mov32	`mov r/m32, r32`	435702	100.00%	14.52%
	`mov r32 , r/m32`	0	0.00%	
mov8	`mov r/m8 , r8`	2640	100.00%	0.09%
	`mov r8 , r/m8`	0	0.00%	
or32	`or r/m32, r32`	10653	100.00%	0.35%
	`or r32 , r/m32`	0	0.00%	
sbb32	`sbb r/m32, r32`	1021	100.00%	0.03%
	`sbb r32 , r/m32`	0	0.00%	
sub32	`sub r/m32, r32`	46750	100.00%	1.56%
	`sub r32 , r/m32`	0	0.00%	
toac32	`test r/m32, r32`	593610	100.00%	19.78%
	`or r/m32, r32`	0	0.00%	
	`or r32 , r/m32`	0	0.00%	
	`and r/m32, r32`	0	0.00%	
	`and r32 , r/m32`	0	0.00%	
toac8	`and r/m8 , r8`	2012	7.12%	0.94%
	`and r8 , r/m8`	0	0.00%	
	`test r/m8 , r8`	26230	92.88%	
	`or r/m8 , r8`	0	0.00%	
	`or r8 , r/m8`	0	0.00%	
xor32	`xor r/m32, r32`	3729	100.00%	0.12%
	`xor r32 , r/m32`	0	0.00%	
xorsub32	`xor r/m32, r32`	182524	100.00%	6.08%
	`xor r32 , r/m32`	0	0.00%	
	`sub r/m32, r32`	0	0.00%	
	`sub r32 , r/m32`	0	0.00%	

Table 4. OpenBSD Instruction Statistics (592 binaries)