Optimal Values for Disrupting x86-64 Reverse Assemblers

Sara Shinn, William Mahoney,

School of Interdisciplinary Informatics University of Nebraska at Omaha Omaha, Nebraska 68182-0500

Summary
We present experimental results from intentionally obfuscating binary instruction data on the x86 64-bit environment. The work is a part of a larger research project to implement an obfuscating compiler, where the intent is to make reverse engineering of compiled binaries more difficult by hiding instructions from the reversing tools. We empirically determine good selections of "junk" bytes in order to maximize the number of instructions hidden or misrepresented by reverse engineering tools.

Key words:
Reverse engineering, intel x86, assembly language, obfuscation

1. Introduction

Protection of intellectual property has been a major concern for countless professions in the engineering world, particularly where binary data is exported for manufacturing of embedded devices, or where binary executables contain keys or activation and licensing data. It is all too easy for software to be reverse engineered and have its methods and secrets analyzed. This research examines the effectiveness of binary obfuscation by inserting various bytes of "junk" at addresses between instructions, in an attempt to confuse disassemblers used in the process of reverse engineering.

Our paper is organized such that section two contains a review of obfuscation in general, to make the reader aware of the various techniques in use, and also lists other studies done in this area; we are aware of 32-bit experiments along these lines, but we examine 64-bit binaries. Following this we present the research question and the methods we employed. The results are given in the next section, section three, and finally some thoughts about future work are listed in section four. This was an empirical study but we mention there that a brute-force approach may also be obtained with a little effort.

2. Obfuscation

Obfuscation is defined as the process of making something obscure or unclear [15]. When applied to software, it is the process of making source code difficult to understand and therefore difficult to reverse engineer. In this section we describe how obfuscation works, and our question concerning the determination of which "junk" yields the best results.

2.1. Overview

Reverse engineering of executable programs, in order to steal intellectual property or modify program behavior, can be thwarted in part by various techniques for performing software obfuscation. These techniques generally fall into three categories:

1. Obfuscation at the source code level, in order to hide variable names, strip comments, remove indentation that is indicative of program structure, and other source level modifications. These techniques might be used where the original source code must be given to the end user, but where we do not desire the end user to understand the intellectual property represented by the code. These also may apply where, for example, a Java file is compiled to a "class" file, which then would contain all of the original variable names, types, methods, etc.

2. Obfuscation at an intermediate level, where techniques can be used to alter the way the program accomplishes certain tasks, but in such a way as to not modify the observable behavior of the program. Examples include replacing "return" statements containing values with "thrown" exceptions containing these values, which are then caught by the caller. These techniques might be applicable to an environment such as compiled C++ where tools exist which will attempt to recreate the program flow in C++ pseudo code from a binary program image. Additional information can be found in Chen [1], Laszlo [8], and others.

3. Finally, obfuscation at a binary level. In low-level obfuscation the assumption is that the reverse engineering process is concerned primarily with recreating an accurate assembly language representation of the binary code. Here, techniques include inserting conditional jumps where a tested predicate is known to be false, in order to lead reverse assemblers astray. Other techniques include inserting "junk" into key areas in the binary so as to throw off disassemblers in architectures with varying length instructions (particularly x86) as described by

Manuscript received November 5, 2011
Manuscript revised November 20, 2011

Balakrishnan [2] as well as Johansson [8] and other previous work as noted above.
It is this final type of obfuscation that is the focus of this paper. The inserted "junk" may represent the first bytes of a valid operation code mnemonic (opcode) that causes subsequent bytes of valid opcodes to be considered as a part of the previous instruction. We address this in greater detail below.

2.2. Reverse assembly

The reverse assembly process is largely accomplished using a mixture of two methods: a linear sweep of the instruction space of the program, and/or a tree-based approach whereby there is a certain amount of program flow analysis is involved.
In the linear sweep method, the program is assumed to contain one long area of instructions and the disassembler starts at the entry point of the program and goes from there. Each instruction is assumed to be decoded correctly, the appropriate number of bytes is added to the current disassembly point, and the process repeats. This method is very simple to thwart because after any unconditional jump or call, "junk" bytes can be inserted at that point. However the linear method is rarely used alone in practice when good reverse assemblers are at use.
The tree-based method should, intuitively, give better results where the code might have a mixture of instructions and data in the same space. A good flowchart describing this approach can be found in the work by Johansson [8]. A detailed description of the method, along with additional heuristics to improve the process, can be found in Kruegel [10]. The basic idea is to keep a stack of destination addresses, initialized with the starting address of the program. At every iteration, the next address is popped and disassembly starts from that address. If the disassembled instruction is a jump or call, then the destination address is pushed onto the stack. The process continues until the stack is empty, and as a result the process mirrors the control flow graph of the program being disassembled.
The two popular reverse assembly tools in widespread use are OllyDbg [12] and IdaPro [6]. IdaPro, in particular, uses a tree-based method, which is described in [4]. In the overall architecture of our obfuscating compiler, these are the tools that will be used for determining the level of complexity added by our overall approaches.
Lastly we note that although IdaPro and OllyDbg are the main tools in practice, in some cases the reverse assembler tools can be automatically created. One example is the DERIVE system described by Hsieh [4], where the user feeds the system examples of assembler mnemonics and output, and the system creates C language declarations which are then used for reverse assemblers or other tools. One of the CPUs tested was the x86 platform and they note that the output is suitable for a reverse assembler which is largely automatically created, although they only provide the instruction-at-a-time level.

2.3. Previous work

Many papers concerning reverse engineering have proposed the insertion of "junk" to throw off reverse assemblers. However the previous studies primarily deal with 32-bit executables. Additionally, many obfuscation techniques deal not with disassembly, as is our intention, but decompilation. The following examples are 32-bit experiments that deal with obfuscation during disassembly.
Dube [14] examined operation code (opcode) shifting in 32-bit applications. Essentially, after being presented with a jump, anywhere from one to eight bytes are inserted in the executable. The debugger must then decide if the bytes were data or instructions – opcodes with possible prefixes. The experiment measured how many instructions were manipulated with two different debuggers: IdaPro and OllyDbg version 1.10. The results varied, with more instructions disguised with more bytes inserted. For a single byte, the average number of instructions obscured was about 2.8.
Linn and Debray [15] implemented a system for 32-bit Linux executables that obfuscated executables, at the user's request, using a variety of techniques. Their research into inserting "junk bytes" in selected locations resulted in 26% to 30% of instructions incorrectly disassembled during a linear sweep.
Although this is apparently a popular technique to use, few have investigated the best choices for "junk" insertion, and those that have explored the problem with respect to x86 32-bit operations.

2.4. Research question

We are in the process of implementing an obfuscating compiler, which will generate code that is difficult to reverse engineer. A part of the project is to eliminate the "fall through" areas in the program flow of control; these are locations where an instruction either jumps to a different location in the object file, or continues with the subsequent instruction ("falls through"). The obfuscating compiler inserts jumps, either to the destination of the original jump, or the destination of the original "fall through" instruction. This allows the sections in the machine code to be rearranged and makes it possible to insert extraneous information between the basic blocks of machine code.
We include selected "junk" which is inserted after jumps and constructed in such a way that they maximize the mistaken identification of instructions buy a reverse assembler. For example, a byte representing the first

portion of an instruction that is N bytes in length will cause the next N-1 bytes to be assumed as part of the instruction. In fact these may contain shorter instructions that are now missed by the reverse assembly process.
The following figure provides an example of this technique in action. The original reverse assembly is at the top of the figure and starts with a "NOP" – no-operation – instruction. If this byte is modified and is set to a byte value of 05 (all numbers are in hexadecimal), the reverse assembly is as shown in the bottom half of the figure.

```
00000000 90         nop
00000001 31ed       xor ebp, ebp
00000003 4989d1     mov r9, rdx
00000006 5e         pop rsi
00000007 4889e2     mov rdx, rsp

After:

00000000 0531ed4989  add eax, 0x8949ed31
00000005 d15e48      rcr dword [rsi+0x48],1
00000008 89e2        mov edx, esp
```

Figure 1. One Byte Replacement Hides Four Instructions.

By modifying the hex 90 to a 05 we have hidden four instructions, the "XOR" through the second "MOV"; the reverse assembler incorrectly assumes that this is a three-instruction sequence as shown.
Our aim in the research is to discover optimal values to insert, in such a way as to maximize the number of missed instructions.

3. Results

Initially we tried to predict the expected number of hidden instructions just by examining the 64-bit instruction set and making an approximation of the average result. However it became clear that the real results using actual 64-bit executable programs may be different because of the actual mix of instructions typically found. We describe here our method for determining this.

3.1. Experimental method

A simple approach to the problem could be to try each combination of several instructions, modifying the initial byte each time to determine which bytes make good choices. This is obviously computational intensive, but also suffers from another drawback, in that as mentioned, certain instructions may be more prevalent than others. An example of this is the x86 32-bit instruction set, which includes, for instance, the "AAA" instruction, "ASCII Adjust after Addition" [4]. This instruction is used for binary-coded-decimal (BCD) arithmetic. However, BCD is not used as often as it might once have been. This leads to the conclusion that we should test against actual x86 64-bit programs, using their actual mix of instructions, as opposed to a synthesized set of all instructions. Further, a large class of instructions that were available in 32-bit x86 mode are not available in 64-bit mode, and cause program exceptions. The "AAA" instruction is, in fact, one such operation. Our conclusion is that we should test not against every potential sequence of bytes, but against actual instruction sequences from 64-bit programs. Thus our first tool was a shell script to extract the instruction area from a genuine x86 64-bit executable program.
For the reverse assembly process, we examined the "objdump" utility available on Linux platforms, but settled instead on "udcli", a tool that is part of the "udis86" project [12]. This tool allows one to feed an arbitrary string of bytes to the program, which displays the disassembly of the bytes as they are entered. Eventually the "gold standard" will be determining the difficulty of reverse engineering our obfuscated binaries using IdaPro; for this experiment any reverse assembler is sufficient and "udcli″ is convenient.

We used this reverse assembly as our starting point, and took advantage of the addresses that are displayed by the tool. In our eventual obfuscating compiler, blocks of code will have space between them where the junk bytes can be inserted. We thus implemented software that inserts a "NOP" instruction at one of these addresses in the binary data. The resulting file is then disassembled and the results are saved. The same location is then replaced with all byte values from 00 hex through FF hex in turn, and this was repeated for every possible instruction address. Each of these is disassembled and compared with the original, and the number of missed instructions is noted. In this manner we collected the differences for each byte value at each instruction boundary in the original program.
Finally we selected a set of utilities from the "/bin" directory on a x86 64-bit Linux machine; our obfuscation data is tied to the binary files on Linux, presumably compiled with GCC. As other compilers might generate slightly different instruction sequences as output, we note that these may be biased towards the Linux/GCC model. However, we feel that the results are likely to be similar for other operating environments and compilers as long as they are on x86 64-bit platforms.

3.2. Experimental results

We considered that a best-case scenario for the maximum number of obfuscated instructions is when the original instructions are one or two bytes in length, and the "junk"

causes them to be considered as the suffix of a longer complex instruction. Although this upper bound might be useful to know, it would not appear in practice because of the unlikely scenario that the code contains such a long sequence of short instructions. Regardless, in order to gauge our results we determined that the maximum number of instructions we could potentially obfuscate was likely around ten, and estimated that the real number was likely to be three or four. We based this estimate on the x86 64-bit instruction format, where longest instruction sequence is 15 bytes [1], and is described below.

Instructions can start with a “legacy prefix” which is bytes whose presence is due to backwards compatibility. In 64-bit mode they are allowed but ignored and include segment overrides, operand size overrides, and other prefixes which were necessary in earlier versions of the architecture. There are five groups of legacy prefixes and normally an instruction can contain a maximum of one byte from each of the five groups. If we are analyzing a current executable compiled on a 64-bit platform with a 64-bit compiler, these prefixes should not normally be present in the instruction if the executable program was created for a 64-bit platform to begin with. In intel CPUs (but not AMD) an additional branch prediction prefix may also be present before the actual operation code.

Following the legacy prefixes is a one byte optional “REX” prefix. The “REX” prefix allows access to extended registers in the hardware and allows access to the full 64-bit operand size. Although the use of the “REX” prefix is optional there are few instructions that default to 64-bit, and thus most instructions would likely contain the prefix if they deal with 64-bit data.

Following the prefixes is the actual x86 operation code, which is either one or two bytes, and then a “ModRM” byte. The “ModRM” byte is used in certain instruction encodings to provide additional opcode bits with which to define the function of the instruction. Next is the “SIB”, or “Scale Index Base” byte. Some instructions have a “SIB” byte following their “ModRM” byte to define memory addressing for the complex-addressing modes.

Finally there are two additional fields, “Displacement” and “Immediate” which can be one, two, four, or eight bytes in length.

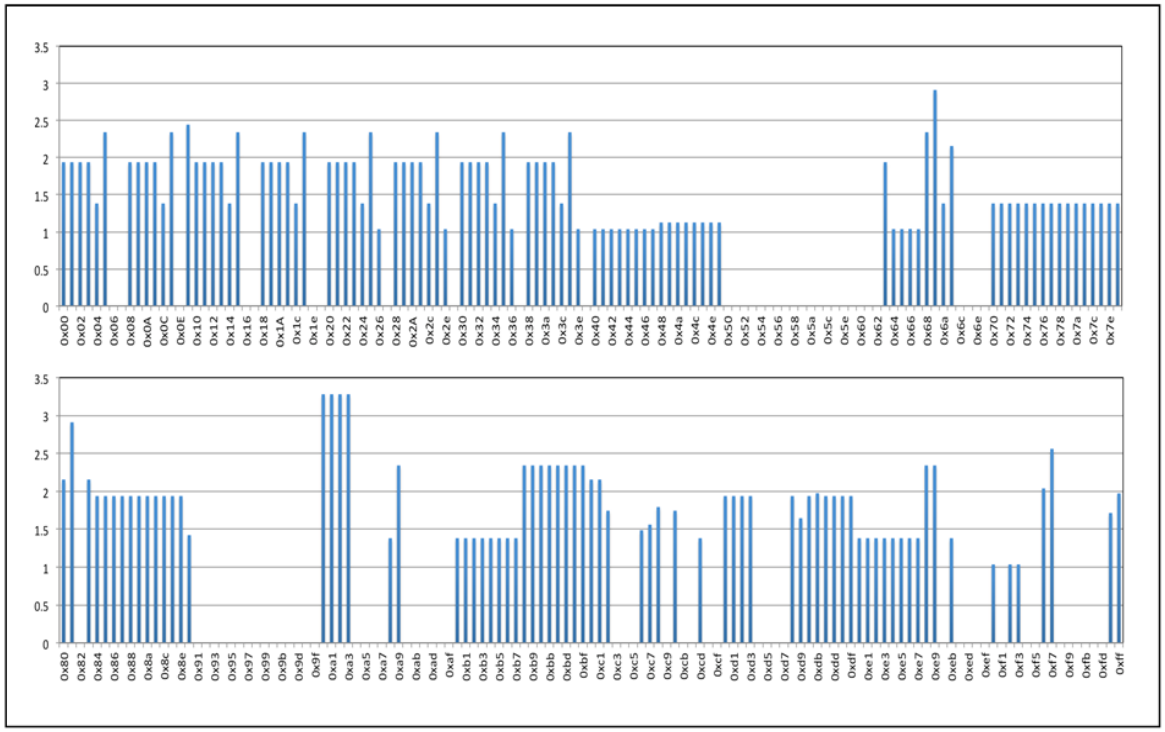

Figure 2. Obfuscated instruction count for “junk” byte values.

Not all instructions contain all fields; thus the total number of bytes described here is greater than the 15 bytes allowed according to the technical reference. By examining some of the object files disassembled, we observed that the average instruction length was 2.57 bytes on our 64-bit Linux system. While a bit lower than expected we note that the areas between sections in the files are padded with “NOP” bytes, which skews the

average lower than if we simply consider instructions that "do something". But using this average we should be able to hide approximately 15/2.57=5.8 instructions. Since the 2.57 average is low due to the propensity of "NOP"s, the 5.8 figure is likely high for the same reason when considering instructions that do actual work. We thus predicted a three to four instruction estimate.

After conducting experiments in the environment previously described, we created a scaled graph of the average number of instructions obfuscated. Figure 2 shows the results of the experiment, and our maximum average was about 1.5 – considerably lower than we anticipated, but with good results in certain areas depending on the selected bytes of "junk". One aspect we neglected to consider is that there are still a large number of one-byte opcodes, and if these are inserted in place of the "NOP" instruction they will yield no change at all in the resulting disassembly; they obstruct nothing. The best results were given with bytes with hex values A0 through A3, which had an average of 3.281 instructions. A quick examination reveals that hex A0 for example represents a "MOV" instruction from a 64-bit address, which causes eight additional bytes to be included in the instruction. Hiding eight bytes with an average instruction length of 2.57 gives a figure of 3.11 instructions, approximately what was actually measured.

For our use, the results indicate that certain bytes should be used for obfuscation and certain bytes should not. Those that should not include obvious selections that are one byte opcodes, thus appearing as zeros on the graph since they hide no instructions at all, but also those that perform poorly such as hex 3E, which only masks 1.037 instructions on average. Sorting the opcodes according to the best obfuscation and selecting those that mask at least two operations yields Table 1 below:

Table 1. Values obfuscating two or more instructions

Value	Instructions	Value	Instructions
0xa0	3.281	0xa9	2.342
0xa1	3.281	0xb8	2.342
0xa2	3.281	0xb9	2.342
0xa3	3.281	0xba	2.342
0x69	2.911	0xbb	2.342
0x81	2.911	0xbc	2.342
0xf7	2.561	0xbd	2.342
0x0F	2.444	0xbe	2.342
0x05	2.342	0xbf	2.342
0x0D	2.342	0xe8	2.342
0x15	2.342	0xe9	2.342
0x1d	2.342	0x6b	2.156
0x25	2.342	0x80	2.156
0x2d	2.342	0x83	2.156
0x35	2.342	0xc0	2.156
0x3d	2.342	0xc1	2.156
0x68	2.342	0xf6	2.039

These values represent those that will be inserted between blocks in the obfuscation process. Selecting from among this set at random is a feature that will be added to our obfuscating compiler in the near future.

4. Conclusions and future work

We determined that the optimal byte values used for obfuscation are actually hex A0 through A3. These were not the instructions we originally considered as yielding best results – initially, we assumed the "REX" prefixes (bytes 40 through 4F) would be the bytes of interest. As it was, these bytes actually yielded below-average results.

We obviously considered only one-byte insertions into the instruction stream, so that the processing of the files could be accomplished in a reasonable timeframe. Multi-byte "junk" may be desirable, and one area of future work would be to craft custom opcode prefixes to maximize the obfuscation.

Currently we are using actual binary executable from our Linux machine as the experimental data. However by modifying the source code for the "udcli" tool it would be possible to perform the experiment again, easily, using the brute force approach. While we feel that the empirical numbers we have obtained are probably better from the obfuscation point of view, because they relate to actual binary code, modifying the "udcli" tool would facilitate the brute force approach and allow for longer sequences of "junk" bytes to be tested.

5. Acknowledgements

This research is based upon work supported by the National Science Foundation under Grant No. CNS-1062995.

References

[1] AMD, "AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions", November 2009, Figure 1.1.

[2] Balakrishnan, Arini, Chloe Schulze, "Code Obfuscation Literature Survey", http://pages.cs.wisc.edu/~arinib/writeup.pdf or http://www.obfuscators.org/2008/04/code-obfuscation-literature-survey.html

[3] Chen, Haibo, et. al. "Control flow obfuscation with information flow tracking" MICRO 42 Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009, ACM New York.

[4] Ferguson, Justin, Dan Kaminsky, "Reverse Engineering Code with IdaPro", 2008, Elsevier.

[5] Hsieh, Wilson C., Dawson R. Engler, Godmar Back, "Reverse-Engineering Instruction Encodings", Proceedings of the 2001 USENIX Conference, 133-146.

[6] IdaPro at http://www.hex-rays.com/idapro/
[7] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual" Volume 2A: Instruction Set Reference A-M, intel, 2008.
[8] Johansson, Hakan T., "Tuning Intel x86 Executables", Master's Thesis, Goteborg University, December 2002, figures 2.2 and 2.3.
[9] Krishnamoorthy, Nithya, Saumya Debray, Keith Fligg, "Static Detection of Disassembly Errors", 16th Working Conference on Reverse Engineering, 2009. http://www.cs.arizona.edu/~debray/Publications/disasm-resist.
[10] Kruegel, Christopher, William Robertson, Fredrik Valeur and Giovanni Vigna, "Static Disassembly of Obfuscated Binaries", 13th USENIX Security Symposium, 2004.
[11] Laszlo, T. and A. Kiss, "Obfuscating C++ Programs via Control Flow Flattening", http://www.inf.u-szeged.hu/~akiss/pub/pdf/laszlo_obfuscating_journal.pdf
[12] OllyDbg at http://www.ollydbg.de/
[13] Thampi, Vivek, "udis86 Disassembler Library for x86 and x86-64", at http://udis86.sourceforge.net/
[14] Dube, Thomas, Kenneth Edge, Richard Raines, Rusty Baldwin, Barry Mullins and Christopher Reuter, "Metamorphism: A Software Protection Mechanism", Proceedings of the International Conference on Information Warfare and Security. 15-16 March 2006, University of Maryland Eastern Shore, Baltimore. Reading, UK: Academic Conferences Limited, 2006. Print.
[15] C. Linn and S. Debray. "Obfuscation of Executable Code to Improve Reisistance to Static Disassembly", 10th ACM Conference on Computer and Communications Security(CCS), pages 290-299, October 2003. http://www.cs.arizona.edu/~debray/Publications/disasm-resist.pdf
[16] Dictionary.com, "obfuscation," in Dictionary.com Unabridged. Source location: Random House, Inc. http://dictionary.reference.com. 2011.

Sara Shinn is from Nebraska City, Nebraska and is an undergraduate student double major in Computer Engineering and Computer Science at the University of Nebraska at Omaha. She participated in a Research Experiences for Undergraduates (REU) program sponsored by the National Science Foundation during the summer of 2011. This paper is the result of her research project and will be a portion of the overall obfuscating compiler project.

William R. Mahoney received his B.A. and B.S. degrees from Southern Illinois University, and his M.A. and Ph.D. degrees from the University of Nebraska. He is an Assistant Professor in the College of Information Science and Technology, University of Nebraska at Omaha, and is the Director of the Nebraska University Center for Information Assurance (NUCIA). His primary research interests include language compilers, hardware and instruction set design, and code generation and optimization, as these topics relate to information assurance goals. As such is interests are in areas such as code obfuscation, reverse engineering and anti-reverse engineering techniques, and vulnerability analysis. Prior to the Kiewit Institute Dr. Mahoney worked for 20+ years in the computer design industry, specifically in the areas of embedded computing and real-time operating systems.