Instruction Embedding for Improved Obfuscation

Charles LeDoux
Center for Advanced Computer Studies, University of Louisiana at Lafayette
cal@louisiana.edu

Michael Sharkey
Center for Advanced Computer Studies, University of Louisiana at Lafayette
mxs6064@louisiana.edu

Brandon Primeaux
Center for Advanced Computer Studies, University of Louisiana at Lafayette
bmp7996@louisiana.edu

Craig Miles
Center for Advanced Computer Studies, University of Louisiana at Lafayette
csm9684@cacs.louisiana.edu

ABSTRACT

Disassemblers generally assume that assembly language instructions do not overlap, therefore, an obvious obfuscation against such disassemblers is to overlap instructions. This is difficult to implement, however, as the number of instructions existing in a program which can be overlapped are typically very few. We propose a modification of instruction overlapping which instead *embeds* the hexadecimal representation of an instruction in the memory offset and immediate operand of an inserted instruction. We implement a obfuscator which is capable of embedding a limited number of instructions and find that it is able to hide 23% of an X86 assembly program's total instructions on average. This is significantly higher than results reported by past works using standard instruction overlapping obfuscations which were only able to hide 1% of instructions.

1. INTRODUCTION

A fundamental assumption made by many disassemblers is that the bytes of instructions do not overlap. In other words, given instructions I and J where J immediately follows I, it is assumed that the first byte of J is the byte immediately following the last byte of I. An obvious obfuscation is to attack this assumption using *instruction overlapping* [2]. Linn et al. [5] define instructions I and J to be overlapping if the last k bytes of I are the first k bytes of J. When a disassembler attempts to continue disassembly at the byte following I, which it is has assumed to be the first byte of J, the disassembler will become "misaligned" and begin to misinterpret the byte sequences it encounters.

An evaluation of instruction overlapping by Linn et al. [5] found that there are two difficulties in implementing instruction overlapping which prevent it from being an effective obfuscation:

- **It is difficult to locate candidate instructions for overlapping.** In order to overlap instruction I with J, the last k bytes of I must *exactly* match the first k bytes of J. Linn et al. found that this condition is not often met.
- **Instruction overlapping does not achieve a high "confusion factor."** The confusion factor is defined as the percentage of assembly instructions that are incorrectly disassembled. Confusion factor is thus a measure of how much the obfuscation "confused" the disassembler. Linn et al. only achieved a 1% confusion factor. This is due to the fact that x86 disassembly tends to quickly "realign" itself, usually within only a few instructions [5].

To address these two problems, we propose a variation of instruction overlapping which prevents the correct disassembly of instructions by *embedding* them within other instructions. Given two instructions, I and J, J is said to be embedded in I if the last k bytes of I comprise the *entirety* of the bytes in J. This definition reduces the instruction overlapping problem to a "byte insertion" problem. Instructions can be hidden by inserting bytes in front of them to transform the instruction J into instruction I and adding a jump over the added bytes to execute J instead of I.

Our method of embedding instructions utilizes instructions which can contain a memory offset followed by an immediate value. By "embedding" the hexadecimal representations of instructions within the offset and immediate value, we are able to embed any instruction less than or equal to the available size. In 32 bit code, this translates to instructions of size 8 bytes or less.

2. RELATED WORK

Cohen [2] was the first to discuss ways to protect a program from reverse engineering by modifying the program. He speculated that an "evolving" program increases the complexity of analysis. He suggested several such evolutions or modifications: replacement of instruction sequences with semantically equivalent sequences, code reordering, junk byte insertion, and instruction overlapping.

Linn et al. [5] evaluated the efficacy of Cohen's suggested obfuscations through a set of experiments. They obfuscated a number of benchmark programs and then measured the confusion factor incurred on a linear sweep, a recursive traversal, and a commercial disassembler. Confusion factor is defined as the percentage of instructions that a disassembler incorrectly disassembles. In other words, the confusion factor is a measure of how often a disassembler becomes "confused." The confusion factor incurred from instruction overlapping was abysmally low; typically less than 1%. The stated reason for such a low confusion factor was the small number of opportunities where two instructions could be overlapped. Our method of instruction embedding remedies this situation, thus allowing many more instructions to be embedded.

The previously referenced obfuscation called "junk byte insertion" is applied by inserting extraneous "junk" bytes into places where (1) the disassembler is likely to expect code and (2) control flow never reaches. For example, junk bytes can be inserted immediately after an unconditional jump. By inserting bytes which correspond to instructions, a disassembler can often be made to misinterpret subsequent byte sequences. Like instruction overlapping, the candidate locations into which junk bytes can be inserted are limited by the structure of the program on which the obfuscation is to be applied. Conversely, our instruction embedding obfuscation is only constrained by limitations on the size of embeddable instructions.

More recently, Desai et al. [3] presented a metamorphic virus generator that modifies its control flow graph by using the types of techniques proposed by Cohen. In a comparison of the generations of a recursive application of their metamorphic engine, the first generation was generally 70% similar to the base virus and the ninth generation only 10% similar.

While Desai et al. attempt to modify the control flow, Balachandran et al. [1] discuss a method of hiding it. They accomplish this by placing a copy of each jump instruction in the data section and then changing the original to a dummy instruction. At runtime, they dynamically reconstitute these instructions immediately before they are executed, and re-obfuscate them afterwards. Instruction embedding is also capable of hiding the control flow of a program. By embedding the control flow instructions of a program, the control flow graph of will be modified and the original control flow hidden.

3. INSTRUCTION EMBEDDING

Instruction embedding is a variation on instruction overlapping designed to increase the number of candidate instructions. Instruction overlapping is implemented by locating two instructions which share a common byte sequence, combining these instructions at the shared bytes, and restoring the original control flow. Instruction embedding, on the other hand, operates by locating an embeddable instruction, inserting bytes in front of this instruction to create a new (embedding) instruction, and prepending a jump to the first byte of the original (embedded) instruction.

Instruction embedding is illustrated in Figure 1. Figure 1a gives the original, unobfuscated X86 assembly snippet. This code segment initializes `eax` to the value 1, then loops until `eax` is equal to the value 2, incrementing `eax` each time. In order to embed the `inc` instruction, we insert the byte `B8` in front of the `inc` (byte `40`). This transforms our `inc` instruction into `mov eax, 0x40`. In order to preserve the semantics, we insert a jump immediately before the newly created `mov` instruction which directs control flow to the embedded `inc` instruction. The obfuscated code segment is given in Figure 1b.

It should be noted that both the type of jump chosen and its placement directly preceding the embedding instruction were made to simplify the example and are not specifically necessary to instruction embedding. The only requirement is that the embedded instruction is executed and no additional instructions which alter the semantics of the program are executed. Other choices may increase the effectiveness of the obfuscation. For example, using an indirect rather than a direct jump may provide an additional layer of difficulty for some disassemblers [6]. Furthermore, placing additional instructions between the jump and embedding instruction may aid in preventing signature based detection of instruction embedding.

Embedding in Operands.

In order for an instruction to be a candidate for embedding in another instruction, only one condition must be met: The last k bytes of the embedding instruction must consist of the *entirety* of the embedded instruction. As seen in the above example of instruction embedding, an easy way to meet this condition is to make use of instructions containing an immediate operand. Since X86 assembly is simply a textual representation for binary values, we can interpret the binary as a numerical value and use this value as the immediate operand. Then, by directing control flow directly to the immediate operand, the binary representing the numerical value is interpreted as code rather than data at runtime.

The types of instructions which can be embedded in an immediate operand are limited by the size of the operand. If it is a 32-bit immediate value (the largest present in X86 assembly code), then only instructions which use four bytes or less can be embedded. This, then, excludes other instructions which contain a 32-bit immediate value. In order to further increase the number of instructions which can be embedded, we also make use of instructions with the ability to specify a memory address with an offset immediately before an immediate operand. For instance, the first operand of the `mov` instruction can be a memory address specified by an offset from an address stored in a register. An example of this is the instruction `mov [eax+0x1234], 0x9876`. The offset is contained within the binary directly before the immediate value, thus enabling us to spread out the bytes of the embedded instruction between the offset and the immediate operand. This effectively doubles the number of bytes available for embedding instructions.

Caveats.

A studious reader will have noted by now that instruction embedding will only be effective against a linear sweep disassembler [4]. Linear sweep disassemblers begin at the first byte of the first instruction and linearly "sweep" through the binary interpreting every byte sequence as an instruction, completely oblivious to the control flow of the program. A control flow aware disassembler, such as a recursive traversal disassembler, will not be affected by instruction embedding as described above. When a recursive traversal disassembler reaches the jump, it will skip over the inserted bytes

Address	Assembly	Hex
0x0	mov eax, 1	B8 01
	LOOP:	
0x2	inc eax	40
0x3	cmp eax, 2	83 F8 02
0x6	je EXIT	74 02
0x8	jmp LOOP	EB F8
	EXIT:	

(a) Original

Address	Assembly	Hex
0x0	mov eax, 1	B8 01
	LOOP:	
0x2	jmp $+3	7C 01
0x4	mov eax, 0x40	B8 40
0x6	cmp eax, 2	83 F8 02
0x9	je EXIT	74 02
0xb	jmp LOOP	EB FB
	EXIT:	

(b) Embedded

Figure 1: Example of Instruction Embedding

and disassemble the embedded instruction. To combat this, we simply replace the unconditional jump with a conditional one and force the proper branch to be taken; such a conditional jump is said to have an opaque predicate. Since the recursive traversal disassembler is not aware that only one path of the branch is ever taken, it will attempt to follow them both and not know how to handle the conflicting paths. In our experimentation, the recursive traversal disassembler we used simply marked everything that followed the conditional jump as data.

4. OBFUSCATOR

To test the feasibility of our method of instruction embedding, we designed and built a proof-of-concept obfuscator. The obfuscator is implemented using the source transformation language TXL.[1] To perform the obfuscations, the assembly code of the input program is first parsed into a parse tree using a grammar we defined. Substitution rules which implement the embedding are then applied to the tree and the modified tree is unparsed into obfuscated X86 assembly code. The substitution rules convert the instruction to be embedded to its hexadecimal form and insert this value into a new instruction as described in Section 3. The substitution rules are also responsible for prepending the jumps such that the embedded instruction is executed.

One of the reasons we chose to perform source level obfuscation was that this eliminated the need to update jump addresses. The obfuscator operates on code pre-linker, so the linker is still able to correctly determine the jump addresses, except in two cases: jumps addresses containing an offset and indirect jumps. It is possible to specify a jump target by giving an offset from some address (`jmp LABEL + 5`). If the source transformation affects any instructions between this type of jump and its target, this jump will be broken. Indirect jumps, on the other hand, don't provide a jump address, but rather a register or memory location which is expected to contain the target address. While the problem of resolving these jumps is an interesting research problem, it is not directly related to instruction embedding. Thus, for our proof-of-concept obfuscation, we assume these two types of jumps are not present.

4.1 Embedding Control Flow Instruction

There are four types of control flow instructions: unconditional jumps, conditional jumps, procedure calls, and returns from procedures. In order to embed the control flow

[1]www.txl.ca

```
...
jmp LABEL
...
LABEL:
...
```

(a) Before

```
...
push eax
mov eax, LABEL
jmp eax
...
LABEL:
pop eax
...
```

(b) After

Figure 2: Transformation used to embed an unconditional jump

instructions, it is necessary to first convert them to their hexadecimal equivalent. This is trivial for the `ret` instruction, as it takes no operands, but it is not possible for the other types of control flow instructions. These instructions often specify their target address as a label (remember that we are ignoring indirect jumps). This label is then resolved into the correct address at link time. Since the obfuscation occurs before the linker is run, it is impossible to know what the target address will be. Thus, in order to embed control flow instructions in our obfuscator, we must first transform them into a form we can embed.

In order to embed unconditional jumps using labels as their target address, we replace the jump with a `mov` of the target address into a register followed by an indirect jump using that register. Including a `push` before and a `pop` after the `mov` ensures that the register is not clobbered. Once the described transformation has taken place, the indirect jump may then be embedded. This transformation is illustrated in Figure 2. The same procedure can be used for call instructions.

The above trick will not work for conditional jumps as indirect conditional jumps do not exist in X86 assembly. One possible way to embed a conditional jump to a labeled target is to invert the condition and change the target such that the not-taken branch becomes the taken branch. After which, an unconditional jump to the target address is inserted. This is illustrated in Figure 3. The unconditional jump can then be embedded using the previously described method.

4.2 Embedding Data Flow Instructions

A listing of some of the more common data flow instructions is provided in Table 1. To embed a data flow instruc-

```
LABEL:
...
jle LABEL
...
```

(a) Before

```
LABEL:
...
jg ORIGINAL
jmp LABEL
ORIGINAL:
...
```

(b) After

Figure 3: Transformation to enable embedding target address of a conditional jump

tion, the hexadecimal representation of the instruction is embedded in the memory offset and immediate operand of a `mov` instruction. The only data flow instructions which cannot be embedded are those which exceed the size limit (64 bits in 32-bit assembly).

Table 1: Common data flow instructions

Mnemonic	**Op1**	**Op2**
	Movement Instructions	
`mov`	`r/m`	`r/m/imm`
`xchg`	`r/m`	`r/m`
`push`	`r/m/imm`	NA
`pop`	`r/m`	NA
`lea`	`r`	`r/m`
	Arithmetic Instructions	
`add`	`r/m`	`r/m/imm`
`sub`	`r/m`	`r/m/imm`
`inc`	`r/m`	NA
`dec`	`r/m`	NA
`mul`	`r/m`	NA
	Logic Instructions	
`and`	`r/m`	`r/m/imm`
`not`	`r/m`	NA
`or`	`r/m`	`r/m/imm`
`xor`	`r/m`	`r/m/imm`
	Key	
`r:`	Operand can be a register	
`m:`	Operand can be a memory address	
`imm:`	Operand can be an immediate value	
NA:	Operand not used	

4.3 Implementation Status

A partial grammar for parsing assembly and a number of transformation rules have been written. The grammar can be used to recognize and parse a large number of assembly instructions, however this list is still grossly incomplete. Additionally, neither data nor section declarations are recognized. The transformation rules currently written are capable of embedding the following instructions: `jmp, push, pop, inc, dec, mov`. The grammar was semi-automatically generated from a list of 32-bit x86 assembly instructions found at `http://ref.x86asm.net/coder32.html`.

5. EVALUATION

In order to test the prototype obfuscator, we obtained a number of assembly programs from the web, obfuscated them with our source obfuscator, and evaluated the obfuscations against a defined set of metrics.

5.1 Metrics

We defined four metrics for evaluation: Instruction Confusion Factor (ICF), Edge Confusion Factor (ECF), Size Cost Factor (SCF), and Edge Penalty Factor (EPF).

Instruction Confusion Factor is defined as ICF = (OI - UI) / OI, where OI is the count of instructions in the original program and UI is the count of instructions from the original program which are still in the obfuscated program. The ICF, then, is the percentage of original instructions which are no longer present in the obfuscated program, i.e., the percentage of instructions which have been embedded. A higher ICF indicates greater obfuscation.

Edge Confusion Factor is defined as ECF = (OE - UE) / OE, where OE is the count of control flow edges in the original program and UE is the number of original control flow edges which remain in the obfuscated program. Thus, ECF is the percentage of original edges in the control flow graph which have been hidden through embedding. A higher ECF indicates greater obfuscation.

Size Penalty Factor is defined as SPF = TI / OI, where TI is the count of instructions in the transformed (obfuscated) program and OI is as defined above. This is a measure of the code size increase. We do not reason as to whether an increase or decrease in the SPF is desirable as the specific application of the obfuscation will dictate the answer. This metric, then, is simply provided for the benefit of the reader.

Edge Penalty Factor is defined as EPF = TE / OE, where TE is the count of control flow edges in the transformed (obfuscated) program and OE is the number of edges in the original program. This metric is a measure of the ratio of new edges to original edges. The more edges present in the control flow graph of the transformed program that were not in the control flow graph of the original program, the harder it should be to reconstruct the original edges that are missing since each such new edge adds its own disinformation.

5.2 Results

The collected measurements for the defined metrics are listed in Table 2. On average, our obfuscator attained an ICF of 0.2316, an ECF of 0.0607, a SPF of 1.6698, and an EPF of 2.7565. The ICF, ECF, SPF, and EPF for our test set are graphed in Figures 4, 5, 7, and 6 respectively. Note that no EPF bar is present for 64bitbcdadd.asm because the result of the calculation is undefined due to the absence of edges in its original control flow graph.

5.3 Discussion

Even with our limited implementation, we obtained a 23% instruction confusion factor. This is a significant improvement over the 1% confusion factor reported by Linn et al. [5]. The improvement is due to the larger number of candidates for embedding over the number of candidates for overlapping. This increase in confusion factor on average incurred a size increase penalty of 167%. Additionally, application of our techniques for embedding control flow instructions obtained a 6% edge confusion factor. This means that 6% of the edges in the control flow graph of the unobfuscated program are not present in the control flow graph of the obfuscated program. The average increase in the number of edges as indicated by the edge penalty factor was 275%.

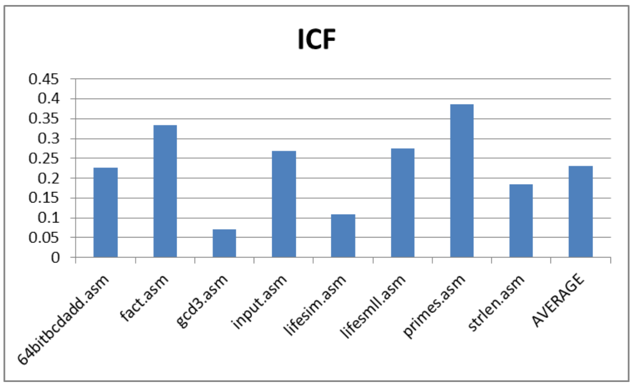

Figure 4: Instruction Confusion Factor

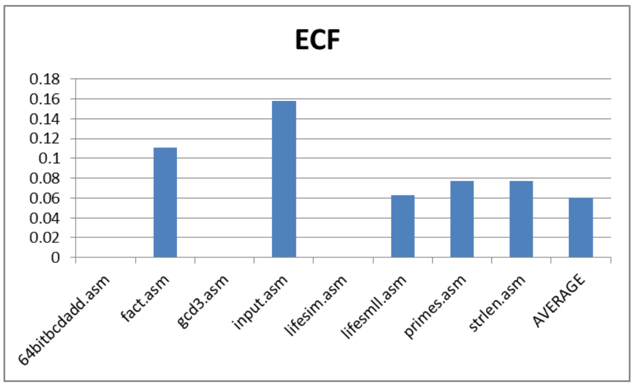

Figure 5: Edge Confusion Factor

Thus, there are nearly three times as many edges in the obfuscated program's control flow graph as there were in the original, implying that the complexity of the control flow graph has increased by a similar amount.

6. CONCLUSION

We have proposed a variation of instruction overlapping called instruction embedding that is designed to increase the number of instructions which can be hidden from disassembly. A prototype obfuscator which uses instruction embedding has been implemented and empirically evaluated on a set of X86 assembly programs. On our limited set, we were able to embed 23% of the instructions. We find that embedding unconditional jumps removes 5% of the edges from the control flow graph.

7. ACKNOWLEDGMENTS

This research was sponsored in part by the Air Force Research Laboratory and DARPA (FA8750-10-C-0171) and the Air Force Office of Scientific Research (FA9550-09-1-0715).

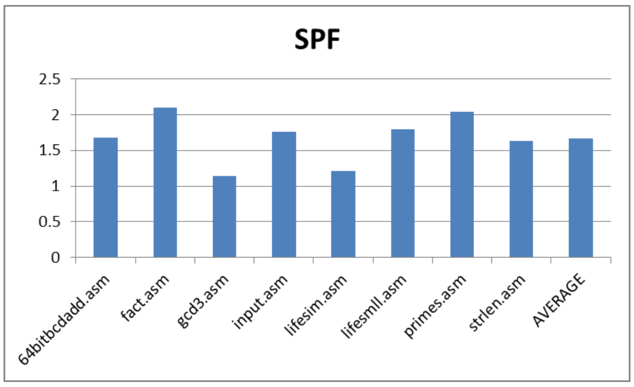

Figure 6: Size Penalty Factor

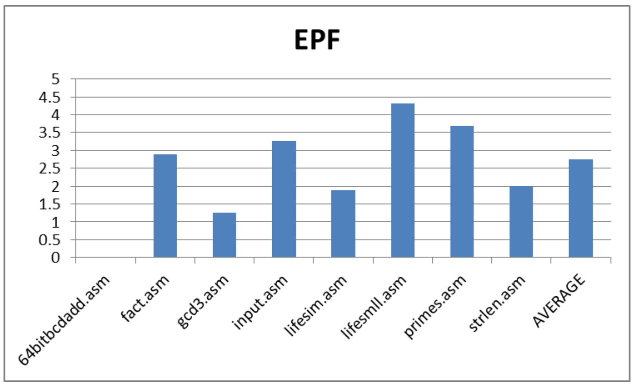

Figure 7: Edge Penalty Factor

8. REFERENCES

[1] Vivek Balachandran and Sabu Emmanuel. Software code obfuscation by hiding control flow information in stack. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*, pages 1 –6, 29 2011-dec. 2 2011.

[2] Fredrick B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[3] P. Desai and M. Stamp. A highly metamorphic virus generator. *Int. Journal of Multimedia Intelligence and Security*, 1(4):402–427, 2010.

[4] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.

[5] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[6] T. Ogiso. Software obfuscation on a theoretical basis and its implementation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 2003.

Table 2: Metrics

File Name	**OI**	**UI**	**TI**	**OE**	**UE**	**TE**	**ICF**	**ECF**	**SPF**	**EPF**
64bitbcdadd.asm	31	24	52	0	0	14	0.2258	0.0000	1.6774	Undefined
fact.asm	21	14	44	9	8	26	0.3333	0.1111	2.0952	2.8889
gcd3.asm	14	13	16	8	8	10	0.0714	0.0000	1.1429	1.2500
input.asm	71	52	125	19	16	62	0.2676	0.1579	1.7606	3.2631
lifesim.asm	37	33	45	9	9	17	0.1081	0.0000	1.2162	1.8889
lifesmll.asm	91	66	163	16	15	69	0.2747	0.0625	1.7912	4.3125
primes.asm	44	27	90	13	12	48	0.3864	0.0769	2.0455	3.6923
strlen.asm	27	22	44	13	12	26	0.1852	0.0769	1.6296	2.0000
Averages	42.00	31.38	72.38	10.88	10	34	0.2316	0.0607	1.6698	2.7565