A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries

Christopher Jämthagen
Department of Electrical and Information Technology, Lund University, Sweden
E-mail: christopher.jamthagen@eit.lth.se

Patrik Lantz
Ericsson Research, Security
Lund, Sweden
E-mail: patrik.lantz@ericsson.com

Martin Hell
Department of Electrical and Information Technology Lund University, Sweden
E-mail: martin.hell@eit.lth.se

***Abstract*—The problem of correctly recovering assembly instructions from a binary has received much attention and both malware and license validation code often relies on various anti-disassembly techniques in order to complicate analysis. One well-known anti-disassembly technique is to use overlapping code such that the disassembler starts decoding from an incorrect byte, but still recovers valid code. The actual code which is supposed to be executed is instead hidden inside a decoy instruction, and is overlapped with the disassembled code.**

We propose and investigate a new novel anti-disassembly method that allows for exceptional flexibility in the hidden instructions, while at the same time providing a disassembled main path that is executable. This allows the approach to be very efficient against static linear sweep disassembly, but also to be more difficult to detect using dynamic analysis methods. The idea is to utilize highly redundant instructions, e.g., multibyte no-operation instructions, and embed the hidden code in the configurable portions of those instructions. By carefully selecting wrapping instructions, providing overlaps, the hidden execution path can be crafted with great flexibility. We also provide a detection-algorithm, together with testing results, for testing software such that the hidden execution path can be identified.

I. Introduction

The application of analysis evasion techniques has both benign and malicious purposes. License validation code can be protected in order to make it more difficult to analyse the validation procedure. This can delay the development of illegal copies that bypass the license validation. At the other end, there are malware authors that want to prevent analysis of their malware. One way to accomplish this is to confuse the software or the analyst into making wrong conclusions about the code and its behaviour. New analysis evasion techniques often also requires new, and sometimes ad hoc, ways of testing software for the presence of evasion attempts, e.g., hidden code. For an analyst there is much time to gain from automated detection.

In this paper we focus on the problem of correct disassembly, and in particular an anti-disassembly technique that aims to trick the disassembler into recovering benign, valid and executable code, which will host the hidden code. The idea is based on using certain instructions in which several bytes can be chosen arbitrarily. We identify several such instructions, but focus on the no-operation (`NOP`) instruction. We show that a large set of instructions can be embedded inside a linear stream of `NOP` instructions. This technique allows a great flexibility in the hidden code and almost any sequence of instructions can be hidden inside the `NOP`s. Furthermore, we discuss how our basic technique can be extended to avoid certain heuristic detection techniques. Some well known disassemblers, both simple and advanced, are used to test the proposed technique and we show that it can be successfully hidden from all these tools. Finally, we discuss a testing approach that is tailored to detect the hidden execution path. We test it on a proof-of-concept implementation and it successfully detects the hidden instructions with good accuracy.

II. Background

This section will provide the necessary background for the rest of the paper and put the results into context by discussing related techniques.

A. Disassembly

Disassembly is the process of taking machine code as input, and outputting human-readable assembly code. There are two main ways of approaching the problem of disassembly, namely *static* and *dynamic disassembly*. Static disassembly consists of analysing and examining the machine code in order to construct the most probable sequence of assembly instructions. This is done without actually executing the code, which is in contrast to dynamic disassembly, where the instructions are executed and identified upon execution. Both methods have their advantages and drawbacks. Static disassembly must guess which instructions are executed, but will on the other hand be able to cover all code. Dynamic disassembly, on the other hand, will know which instructions are executed, but it will only identify those instructions that are actually executed using the current input and environment.

Static disassembly can in turn be performed using a *linear sweep algorithm* or *recursive traversal*. A linear sweep algorithm starts with the first executable byte and then proceeds through the machine code, disassembling instruction by instruction. One drawback of this method is that the disassembler can not easily distinguish data from instructions in the code section of the executable. This can lead to errors if data is located in a stream of instructions. Another limitation is that when one instruction ends, the disassembler assumes that the next instruction follows it immediately. In recursive traversal, the actual control flow of the program is followed. If it

encounters an unconditional jump instruction, the disassembly proceeds at the target address. This will allow the algorithm to avoid errors based on data that is embedded in the code section. On the other hand, it may not always be easy, or even possible, to compute the correct target of a jump instruction. The techniques proposed in this paper will focus on linear sweep and dynamic disassembly. We will also discuss variants and extensions that aims at evading correct disassembly by recursive traversal.

B. Anti-Disassembly

Anti-disassembly is the process of deliberately trying to make it difficult to disassemble machine code. Several techniques have been proposed in the literature [4], [5], [19] and the most basic ideas are typically to take advantage of the limitations in linear sweep algorithms and/or recursive traversal. The fact that the linear sweep algorithm disassembles each instruction in a sequence, regardless of the control flow, can be exploited by adding junk bytes after an unconditional jump. These junk bytes will never be executed, but the algorithm will assume that they are part of the next instruction, resulting in a misalignment between the executed code and the disassembly listing. As long as the junk bytes constitute the beginning of an instruction and the following instruction starts after the end of the junk byte instruction, the disassembler will start producing the wrong code.

The following example will insert a junk byte with a value `0x9A` in order to confuse a linear sweep disassembler.

```
JMP foo          EB 03
DB 0x9A          9A   #This is the junk byte
foo:
XOR EAX,EAX      31 C0
XOR EBX,EBX      31 C3
INC EAX          40
INC EBX          43
INT 0x80         CD 80
```

This piece of code would be disassembled in the following way.

```
JMP foo+1                          EB 03
foo:
CALL DWORD 0x4340:0xC331C031       9A 31 C0 31 C3 40 43
INT 0x80                           CD 80
```

One limitation of using junk bytes as an anti-disassembly technique, is that the disassembly will resynchronize with the real code after a small number of steps since the x86 assembly instructions does not have fixed size instructions. This can be seen in the above example where the instruction `INT 0x80` is found in both disassembly listings.

Code overlapping can be seen as an extension of the junk-byte insertion approach, where code overlapping creates instructions that have bytes in two or more instructions. If the disassembler starts decoding at the wrong offset, it will either resynchronize within a few instructions, or it will mark an instruction as invalid if no possible instruction can be decoded from the starting byte. However, it is possible to craft instructions such that two different offsets will produce valid code, but only one will be executed at run time. By making sure that the instructions always overlap, the two execution paths will not resynchronize until it does so by design. Unfortunately, it is very difficult to find two valid execution paths with this behaviour since the start of one instruction always must be the end of another instruction.

One common technique to fool recursive disassemblers is to use *opaque predicates*. By having a conditional jump instruction where the condition is the same every time a program is executed, the conditional jump works effectively like an unconditional jump if the jump is taken, or a `NOP` instruction if the jump is not taken. Recursive disassemblers will often be fooled to follow the incorrect execution flow while disassembling. Below is an example of an opaque predicate.

```
MOV EAX, 0x1
TEST EAX, EAX
JZ foo+1
foo:
CALL something
```

Register EAX will always contain the value 1, and the conditional jump instruction will never be taken, but a recursive traversal disassembler will first evaluate the target of the branch instruction, starting with the second byte of the `CALL` instruction, and when evaluating the fallthrough instruction it will notice that the bytes are already disassembled and will not include this in its output.

To fool a dynamic disassembler the jump instruction must be able to evaluate to both true and false, depending on the circumstances. For instance if the executable finds itself running within a virtual machine, it may decide to jump to a piece of code with benign functionality, otherwise the malicious content will be executed. There are some interesting techniques for defeating dynamic disassemblers in this way, in which the instructions are generated during run-time and are based on the system environment [4], [5].

C. No-operation Instructions

No-operation (`NOP`) instructions can have many uses, despite the fact that the only effect it has on the CPU state is the update of the program counter. The most common use of the `NOP` instruction is for memory alignment of machine code, for the purpose of more efficient instruction handling by the CPU. Modern x86 instructions, for instance, fetches instructions at DWORD boundaries. If the target instruction of a branch would be in the middle of a DWORD, then the CPU would fetch the preceding WORD as well in addition to the target instruction. Another use is to prevent hazards in the CPU pipeline. There are illegitimate uses of the `NOP` instruction as well. They can be used to create a `NOP` sled, which eases exploitation of some buffer overflow vulnerabilities. `NOP`s has also been shown to be able to fool AV software by injecting them into the malicious code to change the signature of that malware [8].

The most commonly used `NOP` instruction for the x86 architecture is encoded within a single byte, `0x90`, and it is an alias for the instruction `XCHG EAX,EAX`, which simply switches the values between the registers in the two operands.

Compiled binaries often have multiple single-byte `NOP`s after each other to achieve memory alignment. If these `NOP`s

are executed, they will take one clock cycle each to execute. An alternative solution would be to replace multiple single-byte NOPs with one multi-byte NOP instruction.

NOP instructions on the x86 architecture should, according to Intels x86 manual [11], vary between one and nine bytes. In reality though, we can construct valid NOP instructions up to the maximum instruction size, 15 bytes, by using multiple instruction prefixes, but since these extra bytes are all static, it does not add any value to us. Below the recommended encodings of each multi-byte NOP instruction can be seen.

```
Instruction                          Encoding
NOP                                  66 90
NOP DWORD PTR [EAX]                  0F 1F 00
NOP DWORD PTR [EAX+00]               0F 1F 40 00
NOP DWORD PTR [EAX+EAX + 00]         0F 1F 44 00 00
NOP WORD PTR [EAX+EAX + 00]          66 0F 1F 44 00 00
NOP DWORD PTR [EAX+00000000]         0F 1F 80 00 00 00 00
NOP DWORD PTR [EAX+EAX + 00000000]   0F 1F 84 00 00 00 00 00
NOP WORD PTR [EAX+EAX + 00000000]    66 0F 1F 84 00 00 00 00 00
```

For the nine-byte NOP, the first byte (66) is an instruction prefix for overriding the operand-size. The following two bytes (0F 1F) is the opcode. The fourth byte (84) is the so called Mod R/M byte, which essentially describes the format of the operand. The last five bytes (00 00 00 00 00) describe the memory operand. Note that even though the NOP has a memory operand, when executed it does not access that memory in any way. This is simply how the NOP is represented in assembly code.

Since no memory is accessed, we can edit the last five bytes however we want and this would have no effect on how the instruction is executed compared to its recommended configuration.

III. A New Technique for Overlapping Instructions

In this section we give the requirements and the main ideas for our proposed way of overlapping instructions.

A. Requirements

In order to successfully overlap instructions to trick the disassembler into reconstructing the wrong execution path, as described in Section II-B, two requirements must be met.

1) The instructions must overlap each other and must never be aligned such that two instructions end at the same byte.
2) Both execution paths must consist of valid instructions.

Fulfilling both of these requirements is very difficult since an instruction in one path always puts heavy restrictions on the overlapped instruction in the other path. The proposed anti-disassembly technique will meet these two requirements by choosing instructions with certain properties that make it much more manageable to overlap and embed instructions. It will even allow us to choose the instructions in one execution path with much freedom.

In a situation where the program, depending on external or run-time properties, will execute one of the paths, it is crucial that both paths not only produce valid code but that the code is also executable. This will add a third requirement.

3) Both execution paths must consist of executable instructions.

By executable we mean instructions that will, to some extent, guarantee not to crash the program.

B. Overview of the Main Idea

The goal is to assemble a stream of bytes such that when decoding from two different offsets, two different sets of instructions would be found, i.e., two different execution paths. The two paths will be denoted Main Execution Path (MEP) and Hidden Execution Path (HEP) respectively. A static disassembler should only recover the MEP up until the point where the two paths converge. A dynamic disassembler, which will decode the actual executed instructions, will in the case when the MEP is executed recover the MEP. Clearly, in the situation when the HEP is executed, for example when the presence of a virtual machine is not detected, the HEP will be executed and recovered. As malware is often analysed in a VM, this adds an additional layer of obfuscation for the analyst.

The HEP is the most important execution path, since it should be able to hide arbitrary instructions. Therefore we will put as few restrictions as possible on it and allow it to be as flexible as possible. At the same time, the exact effects the MEP has are not important since its primary function is to hide the HEP. To be able to do this efficiently we identify instructions that have as many bytes that can be arbitrarily chosen as possible. The MEP will consist only of these instructions and they will be coded as XX YY ZZ.

XX represents instruction prefixes, the opcode and other static bytes part of the instruction that cannot be changed. YY includes the dynamic bytes, often describing a memory operand or an immediate value of the instruction. The bytes in YY should be large enough to be able to embed a large set of instructions. The YY bytes will form the most important part of the HEP. ZZ should, just as YY, be possible to have any value assigned to it with the only difference that the combination of ZZ followed by XX must encode to a valid and executable instruction. ZZ should preferably consist only of one byte, leaving as many free bytes for YY as possible. The combination of ZZ and XX is denoted the *wrapping instruction*. The wrapping instruction will be executed as part of the HEP and should have little influence over the hidden code. The wrapping instruction is used to glue together the HEP instructions and to separate the MEP and HEP by overlapping two consecutive MEP instructions. Finally, the last HEP instruction should end with ZZ, creating a convergence point for the different execution paths.
The MEP will be decoded and executed as

```
Instruction 1: XX YY ZZ
Instruction 2: XX YY ZZ
Instruction 3: XX YY ZZ
```

while the HEP will be executed as

```
Hidden instruction sequence 1:  YY
```

```
Wrapping instruction 1:          ZZ XX
Hidden instruction sequence 2:   YY
Wrapping instruction 2:          ZZ XX
Hidden instruction sequence 3:   YY ZZ
```

By starting execution in the first bytes of XX, MEP will be executed, while starting execution at YY will execute the HEP.

The next section will describe some instructions that could be used to achieve this.

IV. Suitable MEP Instructions

The largest instruction that has the most bytes that can be arbitrarily chosen and still assemble to a valid instruction is a MOV instruction where the source operand is a 32-bit immediate value and the destination operand is a memory address specified by a register and a 32-bit immediate value as offset.

```
Encoding: C7 80 10 20 30 40 50 60 70 80
Instruction: MOV DWORD PTR [EAX + 0x40302010], 0x80706050
```

This instruction allows the last eight bytes to be chosen arbitrarily and was used in [15] to embed a hidden instruction. While this is a valid instruction it is rarely executable since it will typically point to a memory location that is unavailable to the process, resulting in a program crash. Thus, it does not fulfil the third requirement in Section III-A, meaning that it is easily found using dynamic analysis. If VM detection is used by the malware, avoiding HEP execution inside VMs, an analyst using a VM would detect the crash, simplifying the analysis. Allowing executable instructions that do not risk the program to crash will put more restrictions on the possible instructions. We give four other instructions that can be used to this end.

All have several bytes that can be set arbitrarily and have the additional advantage of being executable without failing. These instructions all have four bytes available for the HEP, plus an extra byte for the wrapping instruction.

LEA. Load Effective Address will calculate the memory address in the second operand and store that value in the first operand. Since we can specify a memory operand here without actually doing any memory accesses, it can be used to insert any byte values in the last five bytes.

```
#Example, load address into AX
LEA AX,[EAX+EAX+0x80] 66 66 8D 84 00 80 00 00 00
```

CMOVcc. This instruction performs a MOV operation if a condition is met. For this instruction to be applicable for this technique, the condition must be chosen such that it always fails. Otherwise it may try to access memory that does not exist and result in a segmentation violation.

```
#Example, perform MOV if overflow flag is set
CMOVO AX,[EAX+EAX+0x80] 66 0F 40 84 00 80 00 00 00
```

SETcc. Similar to CMOV in that it will set a byte to the value 1 if a condition is met. It has the same problem as CMOV as any illegal memory accesses will result in a segmentation violation when the MEP is executed. Caution must be taken to assure the correct conditional is used.

```
#Example, perform SET if overflow flag is set
SETO BYTE [EAX+EAX+0x80] 66 0F 90 84 00 80 00 00 00
```

NOP. A No-Operation instructions can consist of several bytes since it can additionally include e.g., a memory operand. Since no memory is accessed when the instruction is executed, the bytes specifying this operand can be arbitrarily chosen.

```
#Example, 9-byte NOP instruction
NOP WORD PTR [EAX+EAX + 00000000] 66 0F 1F 84 00 00 00 00 00
```

Which instruction to use in the MEP can be situation dependent as they have different properties. Since the HEP instructions will influence the behaviour and effects of the MEP instructions, it is most convenient if the MEP has as little side effects as possible. In the remainder of the paper, we will use the 9-byte NOP instruction since it provides features that are not present in the other instructions.

- NOP only increments the program counter. The other instructions can affect the CPU state beyond the program counter.
- For CMOVcc and SETcc, an illegal memory access is likely to arise if the condition is not set to false.
- LEA will always update the value of its destination register.

Other instructions that can be used are PUSH DWORD, MOV EAX,DWORD and so on, but these limits the number of bytes for hidden instructions (length of YY) in the HEP to three and will thus not be described any further.

V. Assembling the Hidden Execution Path

By choosing multi-byte NOP instructions in the MEP, we have one valid and executable path. In this section, we show how to properly choose hidden and wrapping instructions such that the HEP is executable and easily manageable.

A. Hiding code in a linear stream of NOPs

Since the number of bytes at our disposal in a single 9-byte NOP instruction is quite limited, we must use multiple NOPs to be able to hide any meaningful piece of code. A wrapping instruction between two consecutive NOP instructions is needed to assure the correct execution flow of the HEP. The wrapping instruction will in most cases not be of any use for the HEP and should be chosen to influence the CPU state as little as possible.

To find all possible wrapping instructions, we generated a list of all instructions of the form (ZZ 66 0F 1F 84).

```
instruction := ZZ 66 0F 1F 84
for each possible value of ZZ
    disassemble(instruction)
```

In order to have a NOP for a wrapping instruction, ZZ would have to take up four bytes, leaving only one byte instructions

Category	Instruction	ZZ
I	CMP EAX, 0x841F0F66	3D
	TEST EAX, 0x841F0F66	A9
II	PUSH 0x841F0F66	68
	MOV EAX, 0x841F0F66	B8
	MOV ECX, 0x841F0F66	B9
	MOV EDX, 0x841F0F66	BA
	MOV EBX, 0x841F0F66	BB
	MOV ESP, 0x841F0F66	BC
	MOV EBP, 0x841F0F66	BD
	MOV ESI, 0x841F0F66	BE
	MOV EDI, 0x841F0F66	BF
III	ADD EAX,0x841F0F66	05
	OR EAX, 0x841F0F66	0D
	ADC EAX, 0x841F0F66	15
	SBB EAX, 0x841F0F66	1D
	AND EAX, 0x841F0F66	25
	SUB EAX, 0x841F0F66	2D
	XOR EAX, 0x841F0F66	35
IV	MOV AL, BYTE PTR [0x841F0F66]	A0
	MOV EAX, DWORD PTR [0x841F0F66	A1
	MOV BYTE PTR [0x841F0F66], AL	A2
	MOV DWORD PTR [0x841F0F66], EAX	A3
	CALL 0x841F0F66	E8
	JMP 0x841F0F66	E9

TABLE I: Possible wrapping instructions.

to fit in the HEP. Since we value the flexibility of larger hidden instructions, we will look for alternative wrapping instructions. There are no suitable instructions which have no influence on the state of the machine, when having ZZ consist of one byte. However, there are several other instructions that still can be used. Table I lists the possible wrapping instructions divided into four categories.

Category I includes instructions that change the EFLAGS register. These instructions are suitable when the HEP does not use any jumps or other instructions that relies on evaluation of information in the EFLAGS register. Only two instructions belong to this category, namely TEST and CMP.

```
TEST EAX, 0x841F0F66    A9 66 0F 1F 84
CMP EAX, 0x841F0F66     B1 66 0F 1F 84
```

CMP will use subtract to test the operands, while TEST will perform a logical AND operation. It can be noted that the TEST instruction is faster since the logical AND operation is executed faster than subtraction. If efficiency is important this should be taken into consideration. To form the TEST instruction properly for our needs we will have to assign the last byte of the first NOP instruction with the value 0xA9. This byte paired with the first four bytes of the following NOP instruction (66 0F 1F 84) will form the instruction TEST EAX,0x841F0F66. There are four bytes left within each NOP instruction that can include instructions from the HEP. See below for a stream of NOPs and its embedded HEP representation.

```
MEP:
NOP WORD PTR [ESI-0x56FFFE45]       66 0F 1F 84 66 BB 01 00 A9
NOP WORD PTR [ECX+ESI-0x7F32BF40]   66 0F 1F 84 31 C0 40 CD 80
```

The HEP displayed below will be executed if we start executing at the fourth byte of the first NOP. In the example, it is simply a call to the exit system call for any Linux OS with a return value of 1.

```
HEP:
MOV BX,0x0001                       66 BB 01 00
TEST EAX,0x841F0F66                 A9 66 0F 1F 84
XOR EAX,EAX                         31 C0
INC EAX                             40
INT 0x80                            CD 80
```

Category II includes instructions that change the values of general purpose registers or valid memory, like the stack, without updating the EFLAGS register. Using the PUSH instruction or any of the MOV instructions with the source operand being an immediate value does not alter the EFLAGS register. Thus, the wrapping instruction can be placed between a comparison instruction and the instruction evaluating the EFLAGS register, without changing the semantics of that evaluation. Instead, when using these instructions, we must take care to limit the use of the affected register in the rest of the HEP. As an example, MOV EBP,0x841F0F66 can be used as a wrapping instruction. This will limit the use of register EBP in the rest of the HEP. As HEP will mostly be custom assembly code, EBP might not be used as much here as in compiled code, which makes this instruction particularly interesting.

Category III includes instructions that both change the EFLAGS register and updates registers or memory. These instructions have no apparent advantages over those in categories I and II since they have the limitations of all instructions in those categories. Some of them could be used though, e.g., by using the XOR instruction. Then every other time it is used, EAX will be restored to its original value. ADD and SUB can also be used together to restore the value of EAX if used together.

Category IV includes instructions that cannot be guaranteed to be executable, due to the possibility of illegal memory access.

We have also generated wrapping instructions for the 8-byte and 10-15 byte NOP, but found that the 9-byte NOP gives the best wrapping instructions in terms of maintaining a controllable CPU state.

The main limitation of the (non-wrapping) HEP instructions is the maximum instruction length of four bytes. Except for the last instruction, all instructions are required to be four bytes or less. Still, this requirement can be significantly relaxed as many instructions larger than four bytes can be broken down to multiple instructions of size four or smaller. Below is an example of a MOV instruction that is too large to fit within the HEP.

```
MOV EAX,0x12345678  B8 78 56 34 12  #5 bytes long
```

The five byte MOV instruction can be translated into two 4-byte instructions and one 3-byte instruction.

```
MOV AX,0x5678       66 B8 78 56     #4 bytes long
SHL EAX,0x10        C1 E7 10        #3 bytes long
MOV AX,0x1234       66 B8 34 12     #4 bytes long
```

This reduction in size of instructions significantly increases the flexibility in the choice of HEP instructions, providing e.g., malware authors additional power and possibilities.

VI. Additional Practical Considerations

As a complement to the approach discussed above, it is possible to take additional measures in order to evade analysis.

This section will discuss how the proposed code overlapping technique can be extended in various ways, increasing difficulty of detecting the presence of the HEP.

A. Hiding code in scattered NOPs

As demonstrated in Section V-A, a large piece of code can be put into the HEP, while at the same time allowing the main execution path to be not only valid assembly code, but also represent code that is executable without crashing the program. Thus, in initial analysis attempts using a linear sweep disassembler, it is not straight forward to identify the use of a HEP. Still, the potentially long linear NOP stream that is present in the disassembly will probably stand out and be suspicious. We can improve the stealth of the HEP by scattering NOPs throughout the program. Correct execution flow can be maintained by allocating the last two bytes of each NOP with an unconditional jump instruction to the next hidden instruction embedded within another NOP.

Thus, we can have several short streams of linear NOP instructions, possibly only one at a time, where the last of the instructions contains one instruction with a maximum size of three bytes, followed by the unconditional jump. In this instruction there will only be three bytes available to assign a hidden instruction, limiting the number of instructions that can be used even more than before. It should be noted though that it is still possible to break down some larger instructions to fit this technique. Below is the ADD example from earlier which can be further reduced in size to two and three-byte instructions.

```
#MOV EAX,0x12345678
MOV AL,0x78       B0 78     #2 bytes long
SHL EAX,0x8       C1 E7 08  #3 bytes long
MOV AL,0x56       B0 56     #2 bytes long
SHL EAX,0x8       C1 E7 08  #3 bytes long
MOV AL,0x34       B0 34     #2 bytes long
SHL EAX,0x8       C1 E7 08  #3 bytes long
MOV AL,0x12       B0 12     #2 bytes long
```

By dividing instructions like this, it is possible to use only single NOPs scattered throughout the program. The two-byte JMP instruction can jump forward 127 bytes or backwards 128 bytes in the code, which means that two consecutive NOP instructions, from the HEPs perspective, have a limit on how far away from each other they can be.

This approach has the additional advantage of the possibility to embed hidden instructions in NOPs already existing in compiled binaries. It is not only limited to occurrences of single 9-byte NOP instructions, but it can also be used when there are clusters of single-byte NOP instructions, as these could be converted to multi-byte NOP instructions without changing the semantics of the program. Below is an example of the scattered NOP technique.

```
NOP WORD PTR [ECX+ESI*4+0x4EEB01]   66 0F 1F 84 B1 01 EB 4E 00
...

NOP WORD PTR [ECX+ESI+0x21EB40C0]   66 0F 1F 84 31 C0 40 EB 21
...

NOP WORD PTR [EBP+ECX+0x80]         66 0F 1F 84 CD 80 00 00 00

.nop1:
MOV BL,0x01                         B1 01
JMP .nop2                           EB 4E
...

.nop2:
XOR EAX,EAX                         31 C0
INC EAX                             40
JMP .nop3                           EB 21
...

.nop3:
INT 0x80                            CD 80
```

Note that all bytes in the first NOP are not filled, but since the last byte is preceded by an unconditional JMP, this byte will never be reached and will not cause a problem.

Using scattered NOPs requires a more detailed analysis than when using a linear stream of NOPs.

B. Normalization of MEP instructions

So far both a linear NOP stream and the scattered NOPs works very well to hide an alternate path of execution. An analyst will only see the NOPs in a disassembly listing at first. An experienced analyst may however find it odd to see NOP instructions not conforming to Intel's recommendations and might start disassembling from within a NOP instruction and discover the hidden code.

We want some way to make the NOPs look legitimate and there is only one legitimate representation of the 9-byte NOP instruction. To achieve this normalization we will have to use self-modifying code to generate the correct bytes during runtime. The idea is that during static analysis, an analyst will only see the multi-byte NOP instructions in their legitimate representation and when it is time for the hidden instructions to be executed, a decoding routine will be called to generate the HEP and finally jump to it and continue execution.

To be able to decode the HEP correctly we need to store a key for it somewhere. If we store it as it is, it is more likely it will be discovered as they disassemble to valid instructions. XOR, ADD, MOV, OR and similar operations thus cannot be used by the decoder without revealing the instruction bytes in the data section. Instead we propose to use the SUB operation because we can store a value of 0x100 - 0xXX for each key byte where 0xXX is one byte of the hidden instruction sequence.

The addition of self-modifying code allows the NOP instructions to look legitimate and only if the HEP is about to be executed, the HEP will be embedded in the NOP stream. Otherwise the NOPs will look like normal.

To re-create the HEP, we need a key and a decoding routine. For our proof-of-concept application [13] the decoding routine which is included in the application looks like this:

```
CALL foo
foo:
POP EAX                         #Retrieve EIP
ADD EAX,offset_to_nop           #Point to first byte of NOP stream
MOV ECX,nop_stream_size         #Number of bytes in NOP stream
LEA ESI,DWORD PTR [EBP+key]     #Address to key in memory
bar:
MOV EDX,BYTE PTR [ESI+ECX]      #Get key-byte
SUB BYTE PTR [EAX + ECX],EDX #Subtract key-byte from code-byte
DEC ECX
CMP ECX,0xFFFFFFFF
JNZ bar
ADD EAX,4                       #Adjust jump target to hit HEP
JMP EAX                         #Jump to HEP
```

To hide the decoding routine, we could embedd it within NOPs as well. If the HEP is larger than the decoder routine

we would have less NOPs that look suspicious.

VII. Detection

In this section we will present how well the technique holds up against binary analysis tools. We will test it against some different disassemblers and also a binary analysis framework called BAP, which turns the instructions into an intermediate language representation. We will also suggest solutions for how this technique could be detected.

The proof-of-concept application used as an example for testing is a backdoor [6] hidden within a simple "hello world" program. The executable and details on how the backdoor is hidden can be found at [13].

A. Anti-analysis

The following three analysis tools have been used in the testing.

Objdump [2] is a utility, part of GNU binutils, and its disassembler employs a very basic linear sweep disassembly algorithm and is a natural starting point for testing how appropriate this technique is against these types of disassemblers.

IDA pro [3] is probably the most widely used and advanced disassembler, and for the technique described in this paper to be a viable option for hiding code, it should hold up against an adversary with IDA pro.

BAP [1] is short for Binary Analysis Platform, and one feature of this tool is that it creates an intermediate language representation of the assembly code in the target binary.

The disassembly listings consist of five NOP instructions from the example program's MEP.

1) Objdump: When testing on the objdump utility, the results are very straightforward. Since it uses a linear sweep disassembly algorithm, no branch instruction will be followed and the end result is that the MEP will be shown and the HEP will be hidden.

```
58a:  66 0f 1f 84 6a 01 66  nop  WORD PTR [edx+ebp*2-0x566f99ff]
591:  90 a9
593:  66 0f 1f 84 6a 02 66  nop  WORD PTR [edx+ebp*2-0x566f99fe]
59a:  90 a9
59c:  66 0f 1f 84 89 e1 66  nop  WORD PTR [ecx+ecx*4-0x566f991f]
5a3:  90 a9
5a5:  66 0f 1f 84 cd 80 66  nop  WORD PTR [ebp+ecx*8-0x566f9980]
5ac:  90 a9
5ae:  66 0f 1f 84 89 c2 66  nop  WORD PTR [ecx+ecx*4-0x566f993e]
5b5:  90 a9
```

2) IDA pro: With IDA, the adversary has more options to deal with a binary showing a suspicious disassembly output. The following is what is output by default.

```
.text:080485F6 66 0F 1F 84 6A 01 66 90 A9
                       nop     word ptr [edx+ebp*2-566F99FFh]
.text:080485FF 66 0F 1F 84 52 0F 1F 00 A9
                       nop     word ptr [edx+edx*2-56FFE0F1h]
.text:08048608 66 0F 1F 84 89 E1 66 90 A9
                       nop     word ptr [ecx+ecx*4-566F991Fh]
.text:08048611 66 0F 1F 84 CD 80 66 90 A9
                       nop     word ptr [ebp+ecx*8-566F9980h]
.text:0804861A 66 0F 1F 84 31 C0 66 90 A9
                       nop     word ptr [ecx+esi-566F9940h]
```

The MEP is shown, just as we wanted. The reason the MEP is shown is because we used an opaque predicate to execute HEP. With IDA we can however define some bytes as data and start disassembly from the first byte of the HEP. When doing so the following disassembly output is listed.

```
.text:080485FA 6A 01           push    1
.text:080485FC                 db      66h
.text:080485FC 66 90           nop
.text:080485FE A9 66 0F 1F 84  test    eax, 841F0F66h
.text:08048603 52              push    edx
.text:08048604 0F 1F 00        nop     dword ptr [eax]
.text:08048607 A9 66 0F 1F 84  test    eax, 841F0F66h
.text:0804860C 89 E1           mov     ecx, esp
.text:0804860E                 db      66h
.text:0804860E 66 90           nop
.text:08048610 A9 66 0F 1F 84  test    eax, 841F0F66h
.text:08048615 CD 80           int     80h
.text:08048617                 db      66h
.text:08048617 66 90           nop
.text:08048619 A9 66 0F 1F 84  test    eax, 841F0F66h
.text:0804861E 31 C0           xor     eax, eax
.text:08048620                 db      66h
.text:08048620 66 90           nop
```

If the analyst knows what he is looking for, the HEP will be exposed. To combat this one or more of the techniques listed in Section VI can be used to further confuse the analyst. For instance, by using the scattered NOPs technique the NOPs will not stand out as much as if they are clustered. If the NOPs are normalized, they will not look as suspicious either, even if gathered in a large cluster.

3) BAP: With BAP the interesting thing is how our MEP and HEP are represented in its intermediate language representation. The following output is from BAP

```
addr 0x8048377 @asm ``nop''
label pc_0x8048377
addr 0x8048378 @asm ``nop''
label pc_0x8048378
addr 0x8048379 @asm ``nop''
label pc_0x8048379
addr 0x804837a @asm ``nop''
label pc_0x804837a
addr 0x804837b @asm ``nop''
label pc_0x804837b
```

Note that each NOP is listed as being one byte. BAP uses the MEP in its intermediate language representation and the HEP is completely removed. Any further analysis in BAP will not reveal anything about the HEP.

B. Detection Algorithm

A general method to find hidden code is to look for long sequences of instructions, all of which are unaligned from instructions in the main execution path. The following algorithm can be used to do this.

```
find_HEP(threshold)
    foreach instruction in text-segment
        for i := 1; i < instruction.size; i++
            count := 0
            hidden := disassemble(instruction+i)
            while not in_MEP(hidden) and valid(hidden)
                count++
                hidden := disassemble_next(hidden)
            if count > threshold
                add_to_HEP(instruction+i)
```

This algorithm will try to assemble a stream of bytes into a valid instruction from every possible offset within all instructions of the MEP in the code section of an executable. It will continue to assemble instructions until it has assembled an instruction that is in the MEP or is an invalid instruction. It will look for a certain number of instructions, the threshold, before it considers the instruction stream as a HEP. Choosing this threshold value is tricky since legitimate code can still include a great number of hidden instructions in a row before the disassembly resynchronizes to the MEP. [16] shows that the greatest number of instructions found, which weren't part

of the MEP, is 27 instructions out of 360,000. This was found in the compiler gcc.

The larger the HEP is, the easier it is to find, as we can set the threshold to a higher number, thus increasing our chances of avoiding false positives. Setting the threshold too high could make the HEP go undetected as well. The best way to avoid false negatives is to start with a high value and then decrease the threshold by one until something is found. It is when the HEP is small that it becomes difficult to detect, since the smaller the HEP is, more false positives will arise and it is up the analyst to manually go through all instructions and filter out the false positives.

This algorithm works for all kinds of MEP instructions, not just `NOP`s, as it considers all possible offsets in all instructions as a possible starting point for hidden instructions. It even works for a combination of several different MEP instructions.

The scattered `NOP`s variation of our technique poses a problem for the algorithm, as only two instructions, the hidden instruction and the `jmp` exists linearly. To detect this we need to alter the algorithm so that it follows all unconditional jumps. It should also take into consideration that the `NOP`s may not be in order of the execution flow of the MEP. The last `NOP` instruction in the MEP could for instance be the entry point of the HEP. The jump instruction could also be a conditional in which case the algorithm would have to be modified to follow both paths.

The algorithm was implemented as an IDA pro script and tested against our example application. The script can be found at [13].

We started with an initial threshold of 100 and the backdoor was found and was reportedly 154 instructions long. The largest false positive that was reported was 12 instructions long.

If we apply the algorithm on the code when applying the self-modifying code technique to normalize the `NOP`s, it does not detect the HEP anymore. If we were to hide the decoding routine in `NOP`s the threshold would need to be set to a significantly lower value than 100 to be detected. This means that the use of this additional obfuscation could delay detection further, given that the decoding routine has a smaller number of instructions compared to the HEP.

VIII. Related work

There have been much work involving anti-disassembly techniques. Some of the more innovative ideas involves generating instructions during run-time based on system information so that different instructions are generated depending on the system it is executed on. These instructions are generated with cryptographic hash functions in [5] and pseudo-random number generators in [4]. The idea of embedding hidden instructions within a larger instruction is described in [15]. The first mention of overlapping instructions as a means to complicate disassembly was first described in [9]. Special cases of overlapping instructions has been used as a way to increase tamper-resistance of binaries [12]. It was also mentioned in a dissertation [14] as a problem when developing binary-analysis techniques and tools. Overlapping instructions also have a use in return-oriented [18] and jump-oriented [7] programming scenarios, where a greater amount of gadgets (small snippets of instructions) can be found using the technique. Especially in jump-oriented programming is overlapping instructions beneficial, because the necessary types of instructions needed are scarce, but the op-code byte for it is rich.

The use of `NOP` instructions for unconventional purposes has been shown to have many applications as well. The most famous example may be the `NOP` sled used to ease the exploitation of buffer overflow vulnerabilities [17]. Another example is when the insertion of `NOP` instructions in strategic places in a malicious program's executable code could prevent AV-software from detecting it [8].

Improvement of disassembly methods, like the differentiation of data or junk-bytes from executable code, has also been considered in [20].

We have shown the use of self-modifying code as an additional layer of obfuscation to our technique. There have been research on techniques for reverse-engineering self-modifying code in [10].

IX. Conclusions

We have presented a new technique for anti-disassembly. By using and extending ideas from code embedding and code overlapping we have shown how to overlap two execution paths that are both executable. This does not only complicate analysis using static disassembly with a linear sweep algorithm, but will also make it more difficult to use dynamic analysis since both paths can be executed. Combining this technique with e.g., opaque predicates, self-modifying code and VM detection mechanisms has potential to significantly delay correct disassembly and analysis of e.g., malware, hidden decryption routines and license validation code. Furthermore, we give an algorithm for discovering the hidden execution path by attempting disassembly of code that is offset a number of bytes from the main execution path. This algorithm can successfully and automatically discover malware that uses our proposed technique, potentially saving both time and resources for an analyst.

References

[1] Bap: The next-generation binary analysis platform. http://bap.ece.cmu.edu.

[2] Gnu binutils. https://www.gnu.org/software/binutils/.

[3] Hex-rays ida pro disassembler. https://www.hex-rays.com/products/ida/index.shtml.

[4] John Aycock, Juan Manuel Gutierrez Cardenas, and Daniel Medeiro Nunes de Castro. Code obfuscation using pseudo-random number generators. *2012 IEEE 15th International Conference on Computational Science and Engineering*, 3:418–423, 2009.

[5] John Aycock, Rennie deGraaf, and Jr Jacobson, Michael. Anti-disassembly using cryptographic hash functions. *Journal in Computer Virology*, 2(1):79–85, 2006.

[6] Geyslan G. Bem. shell_bind_tcp.asm, 2013. https://github.com/geyslan/SLAE/blob/master/1st.assignment/shell_bind_tcp.asm.

[7] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11*, pages 30–40, 2011.

[8] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[9] Frederick B. Cohen. Operating system protection through program evolution, 1992. http://all.net/books/IP/evolve.html.

[10] Samuya Debray and Jay Patel. Reverse engineering self-modifying code: Unpacker extraction. 3:131–140, 2010.

[11] Intel. *Intel(R) 64 and IA-32 Architectures Software Developer Manuals*. Intel, June 2013.

[12] Matthias Jacob, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th workshop on Multimedia & security*, MM&Sec '07, pages 129–140, New York, NY, USA, 2007. ACM.

[13] Christopher Jämthagen. Hidden execution paths project website, 2013. http://www.eit.lth.se/index.php?uhpuid=dhs.cej&hpuid=864&L=1.

[14] Johannes Kinder. Static analysis of x86 executables, 2010.

[15] Charles LeDoux, Michael Sharkey, Brandon Primeaux, and Craig Miles. Instruction embedding for improved obfuscation. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 130–135, New York, NY, USA, 2012. ACM.

[16] 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*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[17] Aleph One. Smashing the stack for fun and profit, 1996. http://phrack.org/issues.html?issue=49&id=14#article.

[18] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[19] Michael Sikorski and Andrew Honig. *Practical Malware Analysis, the Hand-On Guide to Dissecting Malicious Software*. no starch press, 2012.

[20] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 6913 of *Lecture Notes in Computer Science*, pages 522–536. Springer Berlin Heidelberg, 2011.