R. K. Lebedev, Использование переключения режимов x86 для защиты программного кода, *ПДМ*, 2023, номер 3, 104–120

DOI: 10.17223/20710410/61/6

УДК 004.056 DOI 10.17223/20710410/61/6

USING X86 MODE SWITCHING FOR PROGRAM CODE PROTECTION

R. K. Lebedev

Novosibirsk State University, Novosibirsk, Russia

E-mail: n0n3m4@gmail.com

A novel program code obfuscation approach involving the x86 mode switching is proposed in the paper. The details and existing applications of x86 mode switching are reviewed, as well as the possible consequences of using this switching to the reverse engineering tools. Based on this approach, a few specific methods are proposed and evaluated against the most popular reverse engineering tools of various purposes, including disassemblers, decompilers, binary instrumentation and symbolic execution tools. A method of seamless integration of these machine code level obfuscations to the C, C++ and possibly other compilers is also proposed.

Keywords: *code protection, reverse engineering, obfuscation, x86 mode switching, disassembly, decompilation, symbolic execution.*

ИСПОЛЬЗОВАНИЕ ПЕРЕКЛЮЧЕНИЯ РЕЖИМОВ X86 ДЛЯ ЗАЩИТЫ ПРОГРАММНОГО КОДА

Р. К. Лебедев

Новосибирский государственный университет, г. Новосибирск, Россия

Предложен новый подход к обфускации программного кода с использованием переключения режимов x86. Рассмотрены детали и существующие применения переключения режимов x86, а также возможные последствия его использования для инструментов реверс-инжиниринга. На основе этого подхода предложено несколько методов, проверенных против различных наиболее популярных инструментов реверс-инжиниринга, включая дизассемблеры, декомпиляторы, инструменты бинарной инструментации и символьного исполнения. Предложен метод бесшовной интеграции обфускаций на уровне машинного кода в компиляторы C, C++ и, возможно, других языков.

Ключевые слова: *защита кода, реверс-инжиниринг, обфускация, переключение режимов x86, дизассемблирование, декомпиляция, символьное исполнение.*

1. Introduction

Program code protection is an important problem to solve currently due to the increasing spread of information technology in our life. Programs can be considered as important as hardware devices, so it is desirable for their authors to be able to leave some of the implementation details unknown to prevent intellectual property theft and software patent violations.

One of the commonly used approaches to the code protection is software obfuscation — the process of program code replacement with some equivalent, but less readable one. It's applicable both on source code and machine code levels. Although absolute code protection

(i.e., turning a program into a black box) is theoretically impossible in general [1], in practice, there are various methods that can greatly increase the time and skill required for reverse engineering, making it impractical.

Machine code level obfuscation is more practical for protecting the commercial software, because it usually comes in a compiled executable form (unless it's written in a language that cannot be compiled). Furthermore, compilation itself can be considered as a kind of code protection, for example IL2CPP AOT compiler of Unity engine makes games significantly harder to decompile compared to the original .NET bytecode, so there is even a project dedicated solely to revert this process [2].

Obfuscation methods can be divided into two types: generic and architecture-dependent.

Generic methods are those that can be performed at any level, including source code, and can be applied to some extent to any platform on which the program can run. This makes these methods particularly useful for cross-platform software development because there are at least two popular processor architectures on the market: ARM and x86. The aforementioned advantage is also significant for web applications written in JavaScript and WebAssembly.

Here are some examples of these generic obfuscation methods:

— Control flow graph flattening is an approach that aims at the branches and loops of the program, making the graph representation of the program "flat" and making it much more difficult to study. One of the most common ways to implement it is to split the program in basic blocks, which are branch or loop destinations, and place them as the clauses of the switch-like structure controlled by some state in the loop. Branching, looping and any other control flow altering is done by changing this state, effectively turning the program into a finite-state machine [3].
— Opaque predicates — insertion of some conditional branches whose condition is always constant, but is unclear during static analysis, thus being "opaque". A branch that is never taken can contain a fake code that may mislead the analyst: incorrect implementation of the block, unrelated function calls or just random bytes. This technique also makes control flow graph of the program more complex, independent of the fake code used [4].
— Constant obfuscation is the replacement of numeric constants and literals with some runtime-computed value, for example, a mathematical formula or string decryption routine. Prevents some simple static analysis methods like strings extraction [5].
— Instructions Substitution is the replacement of standard operations such as arithmetic or Boolean with some other instructions that produce the same result, but are written in a less readable form. A simple example of such transforms for some Boolean operations is the use of De Morgan's laws, and some compilers may actually use similar transforms for optimization [6].

While generic obfuscation methods are quite popular in the existing obfuscators, such as Obfuscator-LLVM and Tigress [7, 8], they have a significant downside: machine code remains completely usual because it is generated by the compiler from source or intermediate representation like LLVM IR. This means that any of the existing reverse engineering tools are still applicable: disassemblers, debuggers and symbolic execution tools are available to the attacker. The code may still be resilient to attacks on its own, but neutralising these tools at the lower level may assist in protecting the program even more.

Here architecture-dependent obfuscation comes useful: it is possible to generate some machine code that generic reverse engineering tools will process improperly or won't process

at all. This is possible because some processor instruction sets are extremely complex, and reverse engineering tools may focus only on the machine code constructions that compilers commonly use. As a variable-length CISC (complex instruction set computer) instruction set, x86 is an architecture, where these methods can be particularly efficient. Furthermore, x86-64 processors retain backward compatibility with 32-bit and 16-bit modes, which complicates things more.

Here are some examples of architecture-dependent obfuscation methods for x86:

— Self-modifying code is a code that modifies itself at runtime. This technique makes it possible to hide entire program logic from static analysis, since the executing code may differ tremendously from the code available in the program image. Packers and protectors, quite popular types of tools to do software protection, also use this approach [9]. This method is only applicable to the von Neumann architecture, and some operating systems may add additional data-code separation, making it impossible to be used.

— Instruction overlapping is a method that uses jumps to the middle of specially prepared program instructions. As x86 uses variable length instructions encoding and doesn't require program counter alignment, it is possible to effectively hide one instruction inside another: for example, in the case of a long (e.g. 64-bit) constant load, constant itself may contain some meaningful machine code. So jumping in the middle of instruction will execute this hidden instruction while it remains invisible to the disassembler, unless it is manually disassembled at the right start position [10]. Furthermore, it is possible to craft machine code that is impossible to disassemble correctly at all. This occurs because one byte is shared between two instructions, both of which are actually executed in the program [11].

— Rare instructions use is a method that involves replacing common instructions with their less common alternatives, e.g. instructions from x86 extensions which could be unsupported in disassemblers and decompilers. For example, constant loading may be implemented via AES-NI instructions instead of plain MOV call, that effectively prevents decompiler attempts to simplify the code and breaks symbolic execution in some tools [12].

The obfuscation method proposed in this paper is also architecture-dependent.

2. Basics of x86 mode switching

Modern x86 processors are able to switch between 64-bit and 32-bit execution modes during program runtime. This feature is particularly useful for running 32-bit programs on 64-bit operating systems, because kernel is always running in the native mode. A transition should happen at system calls to execute 64-bit kernel code in the kernel mode, while the reverse transition should happen to continue executing the program in the 32-bit mode. One of the famous uses of this approach is the WOW64 (Windows 32-bit on Windows 64-bit) subsystem of Microsoft Windows, which allows 32-bit applications to run on 64-bit operating system [13].

This switch is implemented by choosing a proper descriptor in x86 GDT (Global Descriptor Table), which is initialized and set by the operating system, its scheme is available in the Fig. 1.

The interesting bits of the GDT descriptor are L and D bits: when L bit is set, the code is considered to be 64-bit, and the D bit should be unset. D bit set means that the running code is 32-bit, while if both bits are unset, the code is considered to be running in a legacy 16-bit mode. In both Windows and Linux there are corresponding 32-bit and 64-bit user-mode descriptors that have indexes 4 and 6 in GDT, respectively [14, 15].

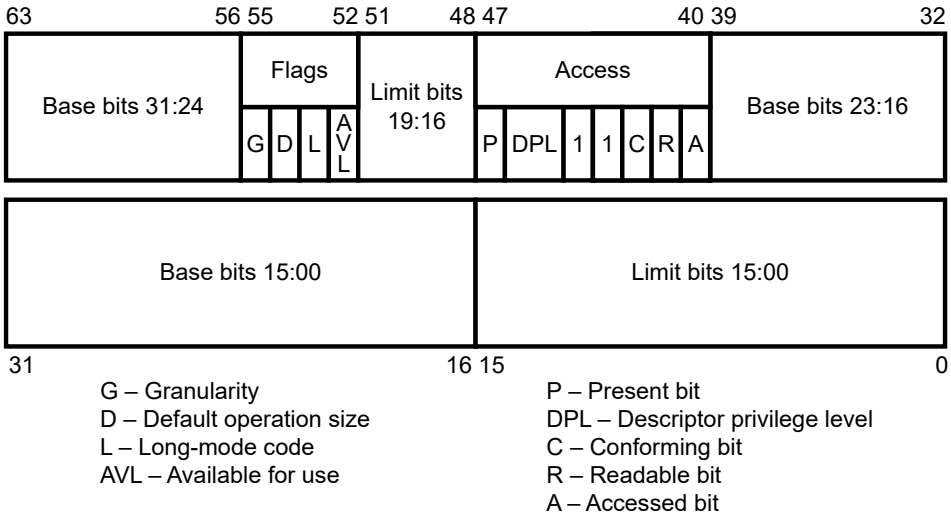

Fig. 1. Code-Segment Descriptor

Modern x86 operating systems are always running in the protected and long modes only. In these modes, a GDT descriptor can be chosen by loading a new segment selector value, that is stored in a Code Segment register (CS), to a proper value. A scheme of this selector is available in the Fig. 2.

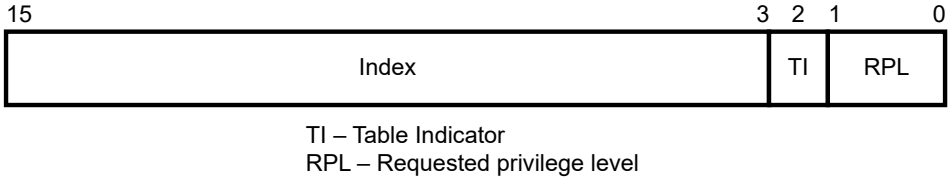

Fig. 2. Segment Selector

To choose another descriptor, one can just change an Index part of the segment selector leaving TI and RPL the same. While the values of TI and RPL can be retrieved from the running OS, according to the documentation for regular user-mode programs they should be TI = 0 (as GDT descriptor is selected, not LDT) and RPL = 3 (because user-mode programs are running in the "ring 3", the least privileged mode).

Therefore, to select a GDT descriptor by index, we should set CS according to the following simple formula:

$$\mathrm{CS} = \mathrm{Index} \cdot 2^3 + 3.$$

So we can compute the CS values for 32-bit and 64-bit execution modes on Windows and Linux using the respective GDT descriptor indexes 4 and 6:

$$\mathrm{CS}_{\text{32-bit}} = 0\text{x}23, \quad \mathrm{CS}_{\text{64-bit}} = 0\text{x}33.$$

The last thing required to switch between 32-bit and 64-bit modes on x86 is the ability to set the CS register. According to the x86 platform documentation, in user-mode this can be done by the following instructions only [16, ch. 3.4.3, vol. 3A]:

1) Far Jump and Far Call: These instructions act like regular JMP and CALL instructions, with the only difference: they not only change the instruction pointer, but also change the code segment. They are available in both 32-bit and 64-bit mode, but immediate values for the address are allowed in 32-bit mode only (EA cp and

9A cp opcodes for JMP and CALL respectively), in 64-bit mode it is required to store an address in memory first. Far Call also pushes the code segment on the stack in addition to the return address.

2) Far Return: This instruction is a variant of a regular RET instruction that can also change the code segment being complimentary to the Far Call instruction. It takes both the target address and the code segment from the stack upon return.
3) Interrupt Return: While this instruction is designed for exception and interrupt handling, in fact it is quite similar to the Far Return: it does the same thing, but also restores EFLAGS register from the stack upon return.

3. Applications of 32-bit x86 mode for obfuscation

3.1. General lack of support in the reverse engineering tools

Mode switching is not common in the regular programs, so there is little demand for supporting it in the reverse engineering tools.

For example, none of the two most popular disassemblers, IDA and Ghidra, can automatically detect x86 mode switching, so the user has to specify 32-bit regions manually. Furthermore, Ghidra is completely lacking mixed mode support [17], so a binary with such switches will have to be divided in parts to be analyzed, that is rather time consuming and inconvenient. IDA does have mixed mode support, which can be used by creating new segments in the program, but it is impossible to use the decompiler with these segments, because IDA requires a correct disassembler version (32-bit or 64-bit) to be launched to decompile respective binary: if the binary uses both modes, this is clearly impossible.

Therefore, just adding mode switching along with some important 32-bit code to the app alone may make it much harder or even impossible to analyze with the reverse engineering tools even if the attacker is aware of it being used in the program.

3.2. Machine code mismatch

Assuming that reverse engineering tools are complicated to use with the mixed mode binaries, one may try to disassemble any machine code as if it was 64-bit, this is also what IDA and Ghidra do by default. This usually works, because most encodings of the instructions remain unchanged in the 64-bit instruction set. Still, there are some exceptions that may be made dangerous to the reverse engineering tools, because these tools will not just fail to disassemble the code, but instead silently misunderstand it, possibly leaving the attacker with incorrect decompiled code.

One of these exceptions is the family of one byte 4* (hex) machine codes [16, ch. B.1.2, vol. 2D]:

— On 32-bit these machine codes are used for encoding the INC and DEC instructions, one variant per register (16 total).
— On 64-bit they have been changed to encode the REX prefix. This is not an instruction on its own, instead, it modifies the behavior of the instruction following it. Lower nibble of its encoding is used as a bitmask with the following bits:
    1) W — operand size override, makes operand size 64-bit;
    2) R — extension of the ModR/M reg field, allows instruction use additional registers introduced in 64-bit mode (R8-R15);
    3) X — extension of the SIB index field;
    4) B — extension of the ModR/M rm field or SIB base field, depending on the instruction.

If an instruction doesn't require a prefix, for example, it is a no-op, the prefix is silently ignored independent of the flags specified [16, ch. 2.2.1, vol. 2A].

This ambiguity may be utilized to create machine code that executes different depending on the mode it is executed in:

1) First, zero out some register, e.g. EAX by using XOR EAX, EAX. This instruction has the same machine code in both modes.
2) Place a 40 (hex) machine code. It will decode as INC EAX in 32-bit mode and as REX prefix otherwise. Here the difference appears: in 32-bit mode EAX will be set to one, while in 64-bit mode it still remains zero.
3) Place a conditional jump depending on the zero flag, e.g. JZ label. Both XOR and INC instructions affect zero flag, so it will be ZF=1 in 32-bit mode and ZF=0 otherwise, so the execution mode will define whether the jump is taken. REX prefix will not affect this instruction in 64-bit mode.

An exact machine code illustration is available in the Table 1.

Table 1

Disassembly ambiguity

32-bit disassembly	Machine code, hex	64-bit disassembly
XOR EAX, EAX	31 C0	XOR EAX, EAX
INC EAX	40	REX JZ 0x1337
JZ 0x1337	0F 84 37 13 00 00	

This jump may be utilized to mislead the reverse engineer by forwarding the reverse engineering tools to the wrong code, creating an enhanced machine code-level version of the opaque predicates protection method.

3.3. Deprecated instructions

Some of the 32-bit x86 instructions are interesting for obfuscation on their own, but were removed from the 64-bit instruction set. The ability to switch to 32-bit x86 mode provides a possibility to use these instructions in the modern 64-bit programs.

One of the notable removed instruction sets is the Intel BCD instruction set, which contains six instructions, including AAM and AAD. These instructions are used to correct the result of binary-coded decimals multiplication and division, respectively. As noted in [18], although they are designed for binary-coded decimals, any other non-10 base can also be provided in the machine code form of these instructions, including zero. Due to assembly source form only supporting decimal base for AAM and AAD instructions, reverse engineering tools may be expected to miss other bases support.

According to the documentation, these instructions have the microcode implementations shown in the Listings 1 and 2 [16, ch. 3.3, vol. 2A].

```
1 IF 64-Bit Mode
2     THEN
3         #UD;
4     ELSE
5         tempAL := AL;
6         AH := tempAL / imm8;
7         /* imm8 is the second byte of machine code 0xA by
      default for AAM mnemonic */
8         AL := tempAL MOD imm8;
9 FI;
```

Listing 1. AAM implementation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL := AL;
        tempAH := AH;
        AL := (tempAL + (tempAH * imm8)) AND FFH;
        /* imm8 is the second byte of machine code 0xA by
    default for AAD mnemonic */
        AH := 0;
FI;
```

Listing 2. AAD implementation

Using this information, one may easily implement some basic transforms of AX register. One of the common register operations is zeroing, usually implemented as XOR EAX, EAX for EAX register. Here we propose another implementation, done with AAM, AAD and BSWAP use, it consists of the following steps:

1) AAM with base 1, which will set AL to zero (as anything MOD 1 is zero) and AH will get the original AL value;
2) AAD with base 0, which will set AH to zero and AL will be increased by AH multipled by zero, so it will keep its value;
3) BSWAP EAX, this is required because AAM and AAD instructions can only operate on the lowest 16-bit half of the EAX register, so we swap these halves and continue with another register half;
4) repeat steps 1-2 for another register half.

This implementation can also be used to zero other registers if needed, with an additional pair of XCHG instructions required to swap target register with EAX and back. An example implementation of this approach is available in the Listing 3.

```
/* Any register may be used here, or this part may be
   omitted if EAX is to be zeroed */
XCHG EBX, EAX
/* Notation AAM (AAD) N is not official, but is supported
   in some assemblers, N is used as imm8 value */
AAM 1
AAD 0
BSWAP EAX
AAM 1
AAD 0
XCHG EBX, EAX
```

Listing 3. Register zeroing using AAM and AAD

4. Implementation of the proposed obfuscation methods

4.1. Mode switch scope and limitations

As with regular pure 32-bit programs, a processor running a part of a program in the 32-bit mode is limited to only have 32-bit memory accesses and 32-bit registers, this puts significant limitations on the method applications, which we will review:

1) First of all, the code running should reside in the lower 32 bits of virtual memory. If executable is not built as a PIE binary (for Linux) or has no dynamicbase option set (for Windows), it is possible to set an executable base in the appropriate area. Furthermore, for Linux it is being done automatically, because the default base address for most Linux compilers on x86-64 is 0x400000, for Windows, though, it is 0x140000000, which is beyond 32-bit range and should be changed. For PIE executables this method can be still used as long as there is no need to protect the whole executable and application can map appropriate memory pages using mmap or VirtualAlloc calls.
2) Protected part of the code shouldn't do any memory accesses to the heap or stack. Both heap and stack are usually allocated above 4GB range, so they cannot be accessed with 32-bit pointers. However, this limitation may not apply to the static memory if it is allocated near executable code, e.g. in non-PIE Linux binaries, where data usually starts at 0x600000.
3) Protected part of the code shouldn't do 64-bit arithmetics or use newer instructions introduced in the 64-bit ISA. However, operations that are known to zero out the upper half of 64-bit registers are allowed because x86-64 has automatic zero expansion for operations involving 32-bit registers [16, ch. 3.4.1.1, vol. 1]. One of the notable exceptions is register zeroing, which is commonly implemented as XOR Exy, Exy instead of XOR Rxy, Rxy.
4) Protected part of the code shouldn't call any 64-bit functions and is recommended to not use system calls. While it is possible to do syscalls from 32-bit mode sometimes, for example, in Linux via x32 ABI, this may require changes in arguments and system call numbers.

While these limitations seem rather strict, they are not very different from the limitations put by virtualization — a method commonly used for protecting the most sensitive code parts [19]. Therefore, methods are still acceptable for math and cryptographic applications, like license checking.

4.2. Mode switch implementation

The most straightforward way to change execution mode is using Far Jump instruction considered in the Section 2. In 64-bit mode, it has to be used as indirect jump with address stored in memory and pointed by register, while in 32-bit mode it is also available as an immediate jump. During experiments it turned out that the stack pointer register (RSP) gets truncated to 32 bits regardless of stack usage in the code, so it can be safely used as a scratch register for jumping from 64-bit mode, as it gets corrupted anyway. Its value can be saved in the RBP register, which doesn't get truncated and is closely related to the stack, which shouldn't be used in 32-bit mode anyway to comply with the mode switch restrictions. Original RBP value can be stored on the stack, while 64-bit indirect jump location and code segment value may be placed anywhere in the executable file. An implementation of this method is available in the Listing 4.

```
1 PUSH RBP
2 MOV RBP, RSP
3 MOV RSP, offset bit32_data
4 JMP fword ptr [RSP]
5 bit32:
6 /* The following code is executed in 32-bit mode */
7 .code32
```

```
 8 XOR EAX, EAX
 9 /* Return to the 64-bit mode */
10 JMP 0x33:offset bit64
11 bit64:
12 .code64
13 MOV RSP, RBP
14 POP RBP
15 /* Code execution continues as regular */
16
17 /* Location and CS value used for indirect far jump */
18 .section .rodata
19 bit32_data:
20 .int offset bit32
21 .short 0x23
```

Listing 4. Mode switch via Far Jump

4.3. Stealth mode switch

During the evaluation of the Far Jump method, it was found that some reverse engineering tools are unable to process this instruction at all. While this still renders them useless, it doesn't allow one to apply the more advanced protection — fake disassembly and decompilation, as proposed in the Section 3.2. While other methods noted in the Section 2 produce nearly the same results, there is one extra method that can be used — exception handlers. Both Linux and Windows expose the processor context to the exception (signal) handlers, so registers can be modified directly, and this also includes the CS register. This eliminates the problems that can appear with rare instruction forms like Far Jump or Far Return, because they are not used anymore.

It should still be noted that exception (signal) handler should preserve the stack pointer address, and the stack used while calling the handler should differ from the current program one, because the stack pointer is invalid due to truncation when the program is in the 32-bit mode. In Linux, this can be done by using the sigaltstack system call that allows one to specify the separate stack for signal handling.

However, causing exceptions in the code may look suspicious to the analyst and cause problems in some reverse engineering tools, as they may seem to be unhandled. For example, using the traditional UD2 instruction to throw an exception results in IDA setting function end as soon as this instruction is encountered.

In this paper, we propose a stealth method to cause exceptions that doesn't affect program listings, that is based on the optimizers abuse. Usually, reverse engineering tools simplify the code during processing to remove the junk code. While this is efficient against constant obfuscation, this gives us a valuable opportunity: we can add an operation that looks like a no-op to the optimizer, but still causes exception and so triggers our exception handler.

The easiest way to achieve this is using some meaningless arithmetic operation, for example, XOR some memory with zero. It is well known that applying XOR with zero to the value doesn't alter it, so the optimizer may remove this instruction if it doesn't care about possible side effects. The real processor, on the other side, will still try to read the value from memory, and trigger an exception if the address is invalid.

An example implementation of this method is available in the Listings 5, 6 and 7.

```
1  static void sighandler(int sno, siginfo_t *si, void *data)
2  {
3      static uintptr_t stack;
4      ucontext_t *uc = (ucontext_t *)data;
5      // Adjust RIP to skip the length
6      // of xor byte ptr [0x1337], 0
7      // It's 8 in 64-bit mode
8      if (uc->uc_mcontext.gregs[REG_CSGSFS] & 0x10)
9          uc->uc_mcontext.gregs[REG_RIP] += 8;
10     // It's 7 in 32-bit mode
11     else
12         uc->uc_mcontext.gregs[REG_RIP] += 7;
13     // We're switching from 64-bit, save stack ptr
14     if (uc->uc_mcontext.gregs[REG_CSGSFS] & 0x10)
15         stack = uc->uc_mcontext.gregs[REG_RSP];
16     // Restore otherwise
17     else
18         uc->uc_mcontext.gregs[REG_RSP] = stack;
19     // Switch 0x23 <-> 0x33
20     uc->uc_mcontext.gregs[REG_CSGSFS] ^= 0x10;
21 }
```

Listing 5. Signal handler implementation

```
1  static void prepare_sighandler()
2  {
3      // Setup an alternative stack for signal handler
4      // It doesn't matter if it is accessible from 32-bit
       address space
5      char * stack = malloc(SIGSTKSZ);
6      stack_t ss = {
7          .ss_size = SIGSTKSZ,
8          .ss_sp = stack,
9      };
10     sigaltstack(&ss, 0);
11
12     // Specify signal handler
13     struct sigaction action_handler;
14
15     action_handler.sa_sigaction = sighandler;
16     action_handler.sa_flags = SA_ONSTACK | SA_SIGINFO;
17
18     sigaction(SIGSEGV, &action_handler, NULL);
19 }
```

Listing 6. Signal handler setup

```
1       ...
2       XOR byte ptr [0x1337], 0
3       /* The following code is executed in 32-bit mode */
4       .code32
5       ...
6       /* Return to the 64-bit mode */
7       XOR byte ptr [0x1337], 0
8       .code64
9       ...
```

Listing 7. Stealth mode switch invocation

5. Software implementation

Being architecture-dependent, the proposed methods are difficult to be implemented at the source code or intermediate representation level. Therefore, machine code and assembly transforms are the only desirable solutions available. While machine code transformation is the most direct way to do machine-level obfuscations, it comes with major tradeoffs:

1) When working with machine code in the object file form, software should be able to distinguish between code and data reliably, and find instruction boundaries for variable length instruction encodings. This is a complicated task, especially provided that for machines with variable length instructions, it may be impossible to disassemble machine code correctly at all, as noted in the Introduction section.
2) Adding any extra code, or replacing existing code with a code of a different length, is also extremely complex. While removing code (and replacing with a smaller one) may be safely implemented by adding a corresponding number of one-byte NOP instructions, adding extra bytes to the code requires recomputing all relative offsets in the code, which include not only instructions but also data, which may have addresses stored in an unpredictable format.

Considering these problems, assembly transform was chosen as an implementation approach. It has other possible issues to consider, but they turned out to be manageable as long as GNU Assembler is used:

1) Assembler may be unable to produce mixed mode machine code, supporting only one variant of x86 at a time. While this may also be managed by carefully choosing instructions, so they match in both architectures, GNU Assembler provides a easier solution by supporting .code32 and .code64 directives. These directives allow to switch between 32-bit and 64-bit variant of x86, respectively.
2) Some machine code has no assembly representation, or there are multiple machine codes equivalent to the same instruction. This may be solved by manually writing machine code byte by byte using .byte directive.

For the easier integration with existing compilers and build systems, it was chosen to implement obfuscation as a GNU Assembler proxy, which transforms the input and then passes it, along with command-line flags, to the original GNU Assembler, a scheme of this process is available in the Fig. 3. This allows the tool to be used with any compiler that supports the use of an external assembly, including the most popular open source compilers GCC and Clang.

The proxy was implemented in Python programming language, processing the assembly code as text, with no additional assembly parsers involved. This decision was made considering the relative simplicity of the assembly language and the transformations applied.

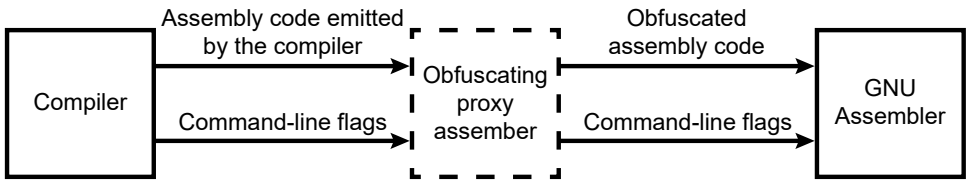

Fig. 3. Obfuscating proxy diagram

It was decided to support the Intel x86 syntax only (not AT&T which some of the compilers emit by default), but this is actually a limitation of the obfuscator prototype used in this work, there is little problem in implementing support for other syntaxes if needed.

Another desirable property of the obfuscator is the ability to fine-tune it right from the source code of the program, specifying which obfuscations should be applied to a particular function or a place in the program. Some of the obfuscators, e.g. Tigress, are using command-line flags for this kind of control, so the user has to specify all the functions he would like to have obfuscated in these flags. This approach was found to be impractical for the two main reasons. First of all, it is complicated to pass command-line flags to the external assembler: while compilers may have options to pass flags to assembler, they don't really have to (there's still an option to use environment variables, but this is harder to debug due to their automatic inheritance). And secondly, this is boring if the project is large enough.

So another approach was proposed: introduce special macros that user can utilize right in the source of the program to configure how it will be obfuscated. While the obfuscator doesn't have direct access to the source code, because it works on the lower level, we may define these macros to emit some inline assembly our tool may process. To our luck, inline assembly (at least in C/C++) gets emitted right in the place where it was defined, so this may even be a middle of the function, giving the most precise control possible.

To not introduce unnecessary compatibility breakage, it was decided to make these macros emit assembly comments keeping backward compatibility with the original assembler if the user would like to build an unobfuscated copy without having to bother about any compilation defines. Example of macros implementing this approach is available in the Listing 8, example of the usage is available in the Listings 9 and 10.

```
1 #define OBF_DEF_LABEL(name) \
2     asm ( "#Obf_label " #name ":" )
3 #define OBF_FAKE_FUN(funname) \
4     asm ( "#Obf_fake_fun " #funname )
5 #define OBF_FAKE_JUMP(funname, retlabel) \
6     asm ( "#Obf_fake_jumpback " #funname " " #retlabel)
```

Listing 8. Macros emitting assembly comments

```
1     printf("This code is entirely visible\n");
2     OBF_FAKE_JUMP(fakefun, retlabel);
3     printf("This is the code to be hidden\n");
4     OBF_DEF_LABEL(retlabel);
5     printf("Code continues\n");
```

Listing 9. A code fragment using proposed macros

```
1       lea rdi, .LC0[rip]
2       call    puts@PLT
3       #Obf_fake_jumpback fakefun retlabel
4       lea rdi, .LC1[rip]
5       call    puts@PLT
6       #Obf_label retlabel:
7       lea rdi, .LC2[rip]
8       call    puts@PLT
```

Listing 10. An assembly code generated by the compiler

6. Results

6.1. Testing methodology

To check the impact of the proposed methods, the following test cases were chosen:

1) A 64-bit program that implements the method described in the Section 3.2 using a Far Jump instruction described in the Section 4.2. Tools that do not support mode switching at all will also automatically fail this test.
2) A 64-bit program that implements the method described in the Section 3.2 using a stealth mode switch proposed in the Section 4.3.
3) A 32-bit program that implements the method described in the Section 3.3. The program is chosen to be built as 32-bit instead of 64-bit so that the tools that fail the previous tests may still pass it independently.

The following reverse engineering tools were chosen for testing:

1) IDA Pro — a popular commercial interactive disassembler with a decompiler [20]. Testing is done both for a disassembler and a decompiler.
2) Ghidra — a popular open source disassembler with a decompiler [21]. Testing is done both for a disassembler and a decompiler.
3) Valgrind — a dynamic binary instrumentation framework [22]. During the test, it is required to execute a program correctly.
4) Angr — an open source binary analysis platform for Python commonly used as a symbolic execution tool [23]. During the test, it is required to reach a particular path in the program. It was tested with two different binary lifters, default PyVEX (based on Valgrind) and P-code (based on Ghidra).

The program chosen for the testing is available in the Listing 11. It imitates a very simple license check with the use of obfuscation macros implementing the method used in tests 1 and 2, OBF_FAKE_JUMP macro inserts a jump intended to be seen by a decompiler only, while OBF_DEF_LABEL defines a label where execution should return to. Due to the availability of two different code paths, this code is also applicable for using with Angr, and an explicit register zeroing also makes it possible to test the method from the test 3, so this single code covers all scenarios required. The environment used for all tests was Linux with the GCC compiler, and all 64-bit tests were compiled with "-no-pie" flag specified.

```
1 #include <stdio.h>
2 #include <obf_defines.h>
3
4 static int user_input;
5 static int check_result;
6 int main()
```

```
 7 {
 8     puts("Please enter license code:");
 9     scanf("%d", &user_input);
10     OBF_FAKE_JUMP(fakefun, retlabel);
11     // This is the real code supposed to be hidden
12
13     // We make sure that zeroing is done with a register
14     // so we can also apply a method described in 2.3
15     register int valid_value = 0;
16     for (int i = 0; i < 10; i++)
17         valid_value += i * i;
18     check_result = valid_value == user_input;
19     OBF_DEF_LABEL(retlabel);
20     if (check_result)
21         puts("License check passed");
22     else
23         puts("License check failed");
24     return 0;
25 }
26
27 void fakefun()
28 {
29     // Fake code supposed to be seen by the decompiler
30     int valid_value = 0;
31     for (int i = 0; i < 1337; i++)
32         valid_value += i * i;
33     check_result = valid_value == user_input;
34 }
```

Listing 11. The code used for testing

6.2. Far Jump experiment results

The following results were observed for the program built with "machine code mismatch" obfuscation enabled with Far Jump mode switching implementation:

— IDA Pro has successfully identified the Far Jump instruction, but it was unable to find its destination, even though it was written in plain bytes. The code following this instruction was considered to be data. After correcting this mistake, the code was disassembled as a pair of XOR and JMP instructions, with no signs of REX prefix or INC instruction. The decompiler has been unable to follow the function past the Far Jump instruction.
— Ghidra has successfully identified the Far Jump instruction and its destination. The code following it was also disassembled, but the mode change was not noticed, so INC instruction wasn't found and REX prefix was omitted. Interestingly enough, Ghidra has also been able to disassemble an immediate Far Jump (EA opcode) correctly despite it being disallowed in 64-bit mode. The decompiler has been able to decompile the whole function "correctly" respectively to the generated disassembly, effectively making a decompiler show the fake program listing provided by the obfuscation.
— Valgrind has been unable to execute the program past the first (64-bit) Far Jump instruction, reporting "unhandled instruction bytes" error, with its bytes followed.

— Angr has been unable to execute the Far Jump instruction with an IR decoding error when PyVEX was used. After switching to the P-code lifter, the execution has succeeded, but the path found belonged to the fake code inserted by the obfuscator.

6.3. Stealth mode switch experiment results

The program was built in the same way as in the previous experiment, except that it used the implementation of stealth mode switching, and this change had a big impact on the results:

— Both IDA Pro and Ghidra have successfully disassembled the code in a 64-bit mode, ignoring the REX / INC byte. Both decompilers have successfully decompiled the fake function they were intended to show by the obfuscation, completely ignoring the XOR operation causing the memory error, that is used for the stealth mode switching.
— Valgrind has been unable to execute the program past mode switch with multiple "unhandled instruction bytes" and "invalid read" errors reported. Instead of crashing, the program has stuck.
— Angr has successfully executed the code and found a path in the fake code with PyVEX. In the P-code mode, execution has failed with error "CALLOTHER emulation not currently supported".

6.4. Deprecated instructions experiment results

In this experiment, the program was built with "deprecated instructions" obfuscation enabled in 32-bit mode, so there was no mode switching in the program. The following results were observed:

— IDA Pro has successfully disassembled the AAM and AAD instructions, along with their machine code-level base arguments. The decompiler has failed to optimize this construction, leaving it in the inline assembly form.
— Ghidra has successfully disassembled the AAM and AAD instructions, along with their arguments. The decompiler has been able to optimize the construction to the variable zeroing, beating the obfuscation.
— Valgrind has been unable to execute the program past the first non-standard AAM instruction, reporting "unhandled instruction bytes" error, with its bytes followed.
— Angr has been able to find a correct answer with both lifters. This is surprising, provided that Valgrind failed to execute the non-standard AAM / AAD forms, but as seen in the warning logs, this support has been manually implemented in PyVEX in 2022, according to the commit history [24].

6.5. Results summary and evaluation

Systemized results of the experiments are available in the Table 2.

Table 2

Observed results

Tool	Far Jump	Stealth	Deprecated instr.
IDA Pro	Failure	Fake code	Success
IDA Pro (dec.)	Failure	Fake code	Failure
Ghidra	Fake code	Fake code	Success
Ghidra (dec.)	Fake code	Fake code	Success
Valgrind	Failure	Failure	Failure
Angr (PyVEX)	Failure	Fake code	Success
Angr (P-code)	Fake code	Failure	Success

As can be seen in the table, mode switching defeats all the tools evaluated. Stealth mode switching is particularly efficient against reverse engineering, because it allows the attacker to see the fake code instead of a real one in most tools, as fake code is much more dangerous to the reverse engineer than a not working tool, because it draws no attention and possibly makes the attacker waste time on it.

Deprecated instructions obfuscation is not as efficient against modern tools, though: it still breaks IDA Pro decompiler, making reverse engineering inconvenient, but has little impact on the other tools. However, it may be effective against less popular and less actively supported tools than those supported at the time of writing, since even Angr only recently received proper support for these instructions (in PyVEX and general P-code support).

7. Conclusion

A novel obfuscation approach is presented, that is not only efficient against the most popular reverse engineering tools, but also has some unique properties.

First, it allows one to build a code that looks different from a disassembler and decompiler output dramatically without using any runtime code modification, an approach commonly used for this task in packers and protectors. This opens an opportunity to use this method on the most protected operating systems, which deny any modification of executable code in runtime. It is also extremely useful against casual reverse engineers that rely on the decompiler only: they have zero chance to know what is actually going on in the protected parts of the program, because this information is not shown in the decompiler output at all.

Secondly, fixing reverse engineering tools to fight this approach is complicated and most possibly time-consuming, because it involves implementing a possibility to switch architectures in the single program. While this was implemented in the past for the ARM-Thumb transitions, a motivation to do so for x86 is incomparably less, as this feature is not used in the real-world applications.

The deprecated instructions use doesn't look as promising on its own in a 32-bit mode, but it is still useful in combination with mode switching, because forcing the code to be disassembled as 64-bit will make these instructions impossible to disassemble for a standard-compliant disassembler, confusing the attacker.

While there is no performance evaluation of the methods in this paper, it is possible to eliminate most of the possible slowdowns with the ability to fine-tune methods usage straight from the source code, thanks to the proposed obfuscation control approach. Nevertheless, it is possible to approximate the time of a mode switching by analyzing the code to be equal to a few context switches worst case, which is barely noticeable unless used in heavy cycles.

REFERENCES

1. *Barak B., Goldreich O., Impagliazzo R., et al.* On the (im)possibility of obfuscating programs. LNCS, 2001, vol. 2139, pp. 1–18.
2. `https://github.com/SamboyCoding/Cpp2IL` — Cpp2IL: Work-in-progress tool to reverse unity's IL2CPP toolchain, 2023.
3. *Wang C., Davidson J., Hill J., and Knight J.* Protection of software-based survivability mechanisms. Proc. Intern. Conf. Dependable Syst. Networks, Goteborg, 2001, pp. 193–202.
4. *Collberg C., Thomborson C., and Low D.* Manufacturing cheap, resilient, and stealthy opaque constructs. Proc. 25th ACM SIGPLAN-SIGACT Symp. POPL'98, San Diego, California, USA, 1998, pp. 184–196.

5. *Collberg C., Thomborson C., and Low D.* Breaking abstractions and unstructured data structures. Proc. Intern. Conf. Computer Languages, Chicago, IL, USA, 1998, pp. 28–38.
6. *Warren H. S.* Hacker's Delight, Second Ed. Addison-Wesley, 2012. 512 p.
7. *Junod P., Rinaldini J., Wehrli J., and Michielin J.* Obfuscator-LLVM — software protection for the masses. IEEE/ACM 1st Intern. Workshop Software Protection, Florence, Italy, 2015, pp. 3–9.
8. `https://tigress.wtf` — the tigress c obfuscator, 2023.
9. *Ugarte-Pedrero X., Balzarotti D., Santos I., and Bringas P. G.* SoK: deep packer inspection: A longitudinal study of the complexity of run-time packers. EEE Symp. Security and Privacy, San Jose, CA, USA, 2015, pp. 659–673.
10. *Jamthagen C., Lantz P., and Hell M.* A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. Workshop Anti-malware Testing Research, Montreal, QC, Canada, 2013, pp. 1–9.
11. *Cohen F. B.* Operating system protection through program evolution. Computers and Security, 1993, vol. 12, no. 6, pp. 565–584.
12. *Lebedev R. K. and Koryakin I. A.* Primenenie rasshireniy arkhitektury x86 v zashchite programmnogo koda [Application of x86 extensions for code protection]. Prikladnaya diskretnaya matematika. Prilozhenie, 2021, no. 14, pp. 138–140. (in Russian)
13. `https://wbenny.github.io/2018/11/04/wow64-internals.html` — WoW64 internals, 2018.
14. `https://community.osr.com/discussion/246643` — Understanding Win 7 x64 GDT/LDT, 2013.
15. `https://github.com/torvalds/linux/blob/master/arch/x86/kernel/cpu/common.c` — Linux Kernel (GitHub), 2023.
16. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. December 2022. 5060 p.
17. `https://github.com/NationalSecurityAgency/ghidra/issues/510` — Allow Different Instruction Sets for Different Memory Sections (Ghidra, GitHub), 2023.
18. Assembly language is too high level. DEF CON 25, 2017. `https://media.defcon.org/DEFCON25/DEFCON25presentations/DEFCON25-XlogicX-Assembly-Language-Is-Too-High-Level.pdf`
19. *Collberg C., Thomborson C., and Low D.* A Taxonomy of Obfuscating Transformations. Technical Report. Department of Computer Science, The University of Auckland, 1997, no. 148. `https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf`.
20. `https://hex-rays.com/ida-pro` — Hex Rays — State-of-the-art binary code analysis solutions, 2023.
21. `https://github.com/NationalSecurityAgency/ghidra` — Ghidra Software Reverse Engineering Framework (GitHub), 2023.
22. *Nethercote N., and Seward J.* Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not., 2007, vol. 42, no. 6, pp. 89–100.
23. *Shoshitaishvili Y., Wang R., Salls C., et al.* SOK: (State of) The art of war: Offensive techniques in binary analysis. IEEE Symp. Security Privacy (SP), San Jose, CA, USA, 2016, pp. 138–157.
24. `https://github.com/angr/pyvex/commit/46049a14985a8d78c6679d75f103540b94c22bc5` — Add generalized aam and aad instructions for x86, angr/pyvex (GitHub), 2022.