Nudnik Project Report

Developed by: Eli Howitt & Malachi Machpod

Instructor: Arie Haenel

Contents

[Preface: 3](#_Toc92457429)

[Abstract: 3](#_Toc92457430)

[Introduction: 4](#_Toc92457431)

[Research 6](#_Toc92457432)

[Techniques: 8](#_Toc92457433)

[Function inlining: 8](#_Toc92457434)

[Notation and motivation for upcoming theory 10](#_Toc92457435)

[Junk code: Theory 10](#_Toc92457436)

[Permuting code: Theory (I) 12](#_Toc92457437)

[Permuting code: Theory (II) 12](#_Toc92457438)

[Permuting code: Implementation 13](#_Toc92457439)

[The techniques as a powerful trio 14](#_Toc92457440)

[The technique pipeline 14](#_Toc92457441)

[User manuel: Nudnik as a script 16](#_Toc92457442)

[User manuel: Nudnik as a an API 17](#_Toc92457443)

[Future Prospects: 18](#_Toc92457444)

[Summary 18](#_Toc92457445)

[Acknowledgements: 19](#_Toc92457446)

[Remarks: 19](#_Toc92457447)

[Appendix: 20](#_Toc92457448)

# Preface:

Nudnik is an anti-disassembly tool developed as a project assignment for the `Fundamentals of Software Security’ (FoSS) course at the Computer Science department of the Jerusalem college of Technology (JCT) during the first semester of the Hebrew calendar year 5782 (2021-2022).

## Abstract:

This document serves as a project report for the assignment of creating Nudnik. This report begins by covering the motivation behind anti-disassembly techniques, a high-level explanation of what some of such techniques do in practice, and some common examples. The report will then proceed by laying out the research we undertook in order to familiarize ourselves with the topic and our insights and decisions towards what became the 3 techniques we decided to implement. Next is a detailing and explanation of the techniques implemented and Nudniks technique-pipeline. Finally the report will end with a user manual for the Nudnik user interface and module, and future prospects of Nudnik.

Note: in this project the nickname Rob is given to the party trying to disassemble a program.

## Introduction:

What \*IS\* Nudnik?

Nudnik is an anti-disassembly tool written in Python 3.8 that accepts assembly 32 bit files as input, applies anti-disassembly techniques to said files and generates the resulting assembly files as ouput, where anti-disassembly refers to the attempt to make the process of static analysis of code more difficult (as opposed to anti-debugging which targets dynamic analysis of code via debugging or anti-virtual machine targeting the dynamic analysis of code within a virtual environment, which all fall under `anti-reverse engineering’).

An important property of anti-disassembly is that it can never really make disassembling the program “impossible”, only more difficult. This is true since as, if a program can run on a machine, therefore there is some well-defined process undertaken by the machine to perform the execution of the said program, so, assuming ‘Rob’ knows how said machine works, all that is left for Rob, is to follow the same process the machine would, and see what would theoretically happen had *he* executed the program. Fortunately (or unfortunately for Rob), there is still plenty to be done to make the above process more difficult, mainly by a) complexifying the said machine process, b) leverage unconventional usage of the machine, or c) by interfering with the tools and methods commonly used in disassembly. Nudnik’s techniques fall under `a’ and `c’.

In other words, the point of anti-disassembly is to make the task of disassembly, an already intense and frustrating task, more frustrating and more annoying. Hence the name Nudnik (`something very annoying’ in Hebrew).

Why anti-reverse engineering

The main cases where one might apply anti-reverse engineering techniques to their program are:

1. To hide **how** the program achieves what it does. This is useful to prevent analysis of the program revealing its vulnerabilities or to maintain some form of ownership over distributed proprietary software.
2. To obfuscate **what** the program does. This is useful for example in malware programs that wish to hide what changes are being made to the infected machine or in an attempt to remain undetected altogether.

For example, the following is taken from the Windows 10 License agreement (section c ‘restrictions’ sub vi, appendix 1):

*“ You may not: … reverse engineer, decompile, or disassemble the software, or attempt to do so…”*

An analysis of the specific techniques implemented by Nudnik is given shortly after the “Research” section.

## Research

To begin with, both of us were only novice reverse engineers ourselves and we have barely come into contact with anti-disassembly techniques, so naturally we started “Googling”. Most of the relevant sources are in one form or another, a listing of ~10 anti-reverse engineering techniques, overlapping at around five of the most common techniques. A list of such resources can be seen at Appendix 2.

We noticed the most common techniques target disassembly programs like Ghidra and IDA by taking advantage of the restrictions and assumptions of such programs. For example, a technique we will nickname `Fake jumps’, takes advantage of the assumption that every branch in a conditional jump instruction is attainable and therefore leads to legitimate code. The technique involves adding conditional jump statements with an unattainable branch(es) that leads to illegitimate code, for example data stored in the .text segment or instructions at an incorrect starting index. These branches will fool the disassembler into wrongfully parsing these branches as instructions and what they really are would be hidden from Rob. Such techniques motivated us to write a rough disassembler script that would break up the code into successive chunks as do disassembler programs. This script was not used in Nudnik but improved our understanding of how such programs work and what are their restrictions and assumptions.

In the process of researching the topic, we were exposed to many interesting facts and peculiarities in the topic of x86 Assembly. Being extreme and peculiar, these didn’t make it directly into Nudnik but further inspired us and expanded our horizon on the topic. For a list of such resources see Appendix 3.

We gathered all of what we have learned, together into a list of interesting topics we’d like to implement. The list composed of 14 techniques, 4 of which we came up with. We were going over the list and having trouble deciding which techniques to implement. We decided to analyze the weaknesses and solutions that Rob might employ to each technique. This analysis shed light on the fact that three of the techniques worked very well together: Function inlining, Junk code, and Permuting code. The details about these techniques and why how they `work well together’ are in the following section.

During development we decided to use the VSVC compiler, and to make changes to the code after compilation before assembly (i.e. on assembly code) this decision was made because it best fit the techniques that we had decided to implement, but there are also some interesting possibilities with making changes at different stages of building.

Another decision made that is worth pointing out, is that whilst the techniques we implemented are architecture-independent, we decided to build x32 programs since the assembly listing of the x64 VSVC compiler aren’t intended for further building and require some cleaning.

## Techniques:

## Function inlining:

The concept of a function is useful for the programmer since it means they can save themselves the work and risk of using the same or similar code in multiple places by creating a single abstraction of the code and jump to it every time it’s used. Similarly, though the use of functions is also convenient for Rob, since he only needs to understand what a function does once and then every time it is called he would already know what is happening. The goal of this technique is to take this convenience away from Rob by inlining (some of) the function calls, i.e. to replace a call to a function (effectively jumping to the function, executing it and jumping back) with the body of the function itself.

The implementation is pretty straight forward, but there are some interesting things to be mindful beyond copy-pasting:

* You must add the variables local to the callee’s .text segment to the callers .text segment.
* Names of variables and labels from the caller and callee might conflict and need to be resolved.
* Stack is assumed to include eip in callee code.
* Depending on the convention (generally speaking stdcall vs cdecl), the caller or the callee may `clean the stack’ and adjustments need to be made.

On the following page you will find the ways that we dealt with the above restrictions (ordered respectively), and an illustration of the technique.

How we dealt with the above (respectively):

* Added the variables.
* Swapped conflicting names with new ones.
* Added a stub push and pop to the stack (only popping with caller clean up i.e. cdecl style). This assumes the function doesn’t make explicit use of eip value expected on stack.
* We assumed all ‘ret imm’ where imm is nonzero, are callee clean up, i.e. stdcall style.

Before: After Function inlining:

func1 proc

<some code>

sub esp 4

<print code>

Add esp 4

<more code>

ret

func2 proc

<some code>

sub esp 4

<print code>

Add esp 4

<more code>

ret

func1 proc

<some code>

call print

<more code>

ret

func3 proc

<some code>

sub esp 4

<print code>

Add esp 4

<more code>

ret

print proc

<print code>

ret

func3 proc

<some code>

call print

<more code>

ret

func2 proc

<some code>

call print

<more code>

ret

## Notation and motivation for upcoming theory

Before we discuss the remaining two techniques, we should first observe that, any instruction can be boiled down to memory manipulation. We will find that it is therefore useful to divide the memory units an instruction can involve into the following:

* Each register
* Each flag bit (technically a register but worth treating differently)
* The rest [stack, data, heap…], we’ll call this `mem’ sometimes
* And then for each instruction to consider which of these it **changes** and which of these it **uses**. We will denote cx and ux as an instruction changing and an instruction using some memory unit x respectively.

There are details that we won’t go into (such as instructions that use and change x (e.g. add x, 2), or the special cases of mem unit), since not considering them still provides the main ideas. Those of you interested in the superfluous details should read the code.

This is true (consideration is useful) because the conditions describing when one can add junk instructions or when one can permute instructions depend solely on cx\ux representation of the instructions in the process being considered. These conditions are different and will be presented separately for each of the two techniques.

Note that parsing of instructions is limited to the instructions we support, currently around 10 most popular instructions (accounting however for >70% of assembly code according to appendix 4). For reasons that will become clear later, we ‘assume the worst’ for each instruction we don’t support and for control flow instructions and set them to change and use every memory unit.

## Junk code: Theory

In this technique we make the process of understanding the disassembled code longer, more complicated, and more daunting (which cannot be overlooked). We do this simply by increasing the size of the code by injecting instructions that will execute but won’t affect the effect of the code (not to be confused with code that would affect the effect of the code but would never actually run). We call the instructions ‘junk’.

The kind of junk we chose to support is such that changes a certain memory unit (another option is groups of instructions that cancel each other out like {swap(a, b); swap(a, b)} or { pusha; <code changing only registers>; popa}).

So when can you insert a junk instruction? The restricting property here is the effect of the instruction, i.e. what it **changes**. We don’t care what memory units it uses. Therefore, we will consider the junk instruction we want to add as what it changes and not what it uses, lets denote junk = cx.

When does the addition of an instruction that changes x not affect the program? Only when the next occurrence of x is such that also changes x. In our notation cx can only go before an existing cx, not some ux.

For example, if the following array represents some procedure then the red lines are location where cx can be inserted:

[ cx cy uy  ux  cx uz cx  cx  cy cz  uz  uy ux ux cx cz cx ux  cx]

Junk code: Implementation

Now that we have established a (necessary and sufficient under the scope of our code) condition for the ability to add junk code, whats left is to represent these constructs in code. The way we decided to implement this is to consider a 2d matrix `isNextUse’ where-

isNextUse[unit][index] = is the next occurrence of memory unit `unit’ started from `index’ and onwards, an instruction that uses `unit’

Computing this matrix is simple by looping through the instructions in reverse.

Next we loop again through the instructions (in forwards order) and for each instruction we maintain an array of memory units that can be changed before this instruction. We pass this array to a utility function that creates random junk code and we inject the result.

## Permuting code: Theory (I)

This technique involves permuting the instructions of a process in such a way that doesn’t affect the effect of the process. On its own, the technique doesn’t achieve much (occasionally making code look `strange’), but in combination with Function inlining and Junk code it is a powerful tool. The virtue of this combination will be explained following the sub section on this technique.

When can instructions be permuted? – firstly we point out that instructions involving unrelated memory units can trivially be permuted with respect to each other, therefore it will be useful to consider the instructions from the lens of a single memory unit , `x’. This order invariance property lets us ignore other units from x’s POV, therefore what x sees is [cx ux  cx cx  cx  ux ux cx cx ux  cx] (from junk code example).

Now observe that you can never permute any cx with respect to another ux. This motivates the following grouping:

We divide up this sequence into sets of cx and sets of ux alternating. In the example:

[{cx }{ ux } {cx cx  cx  } {ux ux } {cx cx } {ux  }{cx}]

The instructions in each set are almost order-invariant, there is still one more limitation to consider: one cannot separate an the last cx before a group of ux’s (this is because the last time you change x before using it will determine the value being used (obvious but easy lose the forest for trees here)). This motivates the finale change to our model - putting the last cx in a separate set. And in the example:

[{cx }{ ux } {cx cx  }{cx  } {ux ux } {cx }{cx } {ux  }{cx}]

Now we can safely say our model maintains sets of order invariant instructions when only considering x. The next page will generalize the use of this model to include multiple memory units at once.

## Permuting code: Theory (II)

Let us now consider our model for n memory units:

[{c1 c1 …}{c1}{ u1u1 …}{c1 c1 …}{ c1 }{ u1 u1 …}…]

[{c2 c2 …}{c2}{ u2u2 …}{c2 c2 …}{ c2 }{ u2 u2 …}…]

…

[{cn cn …}{cn}{ unun …}{cn cn …}{ cn }{ un un …}…]

\*reminder that the same instruction can use and change multiple memory units and can therefore appear in many of these models at once.\*

This is what our model would look like if we coloured the first set of instruction of each unit in green and the remainder in cyan (in general some unit could have its first set be that of uses but the idea is exactly the same).

Each memory unit is saying: “I`m ready for the instructions in my green set, but I am not ready for the instructions in my cyan set”, but since each unit’s model was only considering the instruction it is involved in; it is also implicitly ready for any of the instructions not in its model. Therefore, the only thing each unit cares about is ‘don’t use instructions in my cyan region’.

Taking everyone into consideration (as democracy would approve) we can express the set or order invariant instruction \*that are available to use right now\*:

available\_instruction = all\_instructions / U1<=i<=n(cyani) (U for union)

Lastly, we point out that this expression holds true considering the instructions of a process from **any** index onwards as long as the n models (cyani) is maintained.

This gives us enough loop through the indices of the process and build it from the ground up. The implementation is provided in the next page.

## Permuting code: Implementation

We begin looping through the instructions and computing the 2d matrix `unitChunks’ where unitChunks[unit] = stack representation of model of unit. Again, it is necessary to loop in reverse, in order to spot the `last cx’.

We then consider the integer array `notReady’ whos length is that of the number of instructions in the process being processed, and where notReady[idx] = number of memory units not ready for instruction at index `idx’.

We then set availableInstructions = allInstructions

Next loop through the second-through-last sets of each unit (the cyan portion) and update notReady. When notReady[idx] increases from 0 to 1 we remove it from availableInstructions.

Now that pre-computations are done, we can build a random, valid permutation of the instructions by iteratively:

* popping a random available instruction `ins’.
* Appending ins to our new list of instructions.
* Updating unitChunks that included ins in green region:
  + if their: green region is empty pop it and update notReady with new

green region (`I’ that become notReady[i] = 0 get added to the available instruction set).

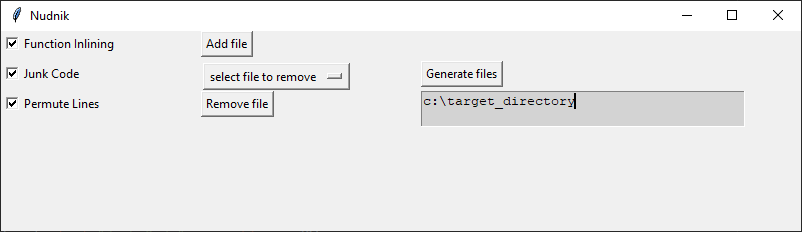
## The techniques as a powerful trio

Function inlining is powerful but when we were debating which techniques to implement we thought that its affect can be removed with a script that spots the same code section in many places and replaces them with a function (replaces being loosely defined here). This is where Permuting code comes in – if a function has been inlined in multiple places, then randomly permuting the instructions would produce different copies of the same function such that a comparison script wouldn’t spot it. Added to that random junk code all over, apply each technique a bunch of times in different orders and you get a nightmare to disassemble (or at least a nudnik ☺).

## The technique pipeline

While applying all three technique many times in many different orders is tempting, each time the techniques become more and more expensive timewise to apply, so we created a class `Techniques’ who’s role is to provide an ordered pipeline of techniques to be applied. This class is also in charge of deciding details like how much junk should be inserted in the relevant locations.

## User manual: Nudnik as a script



Executing the script `userInterface.py’ will present the above window. On the left are checkboxes determining which techniques you wish to apply. In the center a button ‘Add file’ which opens an explorer window and lets the user select files to add to the current list. Files can be removed from the list by selecting them from the drop-down menu ‘select file to remove’ and then pressing ‘remove button’. Finally, once a user has selected the correct list of files they desire to use, they enter the target directory for the generated files in the grey textbox and press ‘Generate files’ to generate the files in said directory.

## User manual: Nudnik as a an API

Of course, those who want to use Nudnik programmatically are free to use its modules in their own software. This is a brief overview of the intended use and structure of the modules’ contents:

In fileData.py you’ll find the class FileData which encapsulates the information of a given assembly file. It can be constructed with such a file and after changing its contents (either via the implemented techniques or by implementing other methods entirely) be exported to a .asm file with the `save’ method.

In techniques.py youll find the three techniques as functions that input and output a FileData object (Junk code is wrapped by ‘getJunkCodeFunction’ that generates it given a ‘junkSize’ – a measure of how much junk to inject).

Also, in techniques.py you’ll find the Techniques class whos role was explained earlier. Its intended use is to construct an instance with Boolean values representing which techniques this instance includes. Get the list of functions to apply to you FileData object with its ‘techniqueFunctions’ member.

# Future Prospects:

Nudnik does a great job doing but it could be doing its job better by:

* Supporting parsing for more instructions
* Supporting more junk instructions
* Have Techniques monitor the files and more intelligently decide which techniques to further apply

Nudnik can do more by:

* Supporting more architectures (at least 64 bit)
* Apply changes at different stages of build (e.g. mid compilation)
* Support more techniques
* Be converted to C for better performance and the ability to apply Nudnik to itself to hide its techniques

# Summary

Lots of effort went into making Nudnik a real nudnik. The three techniques it supports each have a different and necessary effect which when combined complement each other. Nudnik was a lot of fun developing and we are very proud of its results. That being said, Nudnik has lots of room for improvements and plenty of naughty techniques we would like to support in the future.

## Acknowledgements:

We’d like to thank our instructor Professor Arie Haenel for helpful guidance throughout the development of Nudnik, and for teaching us the most interesting and useful college course we’ve undertaken.

We would also like to point out our gratitude towards the Computer Science online communities which without their contributions on stack overflow, anti-debugging resources, x86 listings and detailings, etc. we would be in a lot of trouble! ☺

# Remarks:

A word of warning about using function inlining technique on recursive functions – for better or worse when function inlining repeatedly applied to recursive functions with recursive calls = a0 > 1grows **really fast**. If an represents the number of calls after applying the technique n times, then since each call is replaced with all a copy of the function containing an calls: an+1=an2=> an = a0^(2n) (easy proof by induction or by considering the series bn = loganything(an)).

Pythons built-in random.sample runs with O(n) on sets, therefore to select a random element from ‘availableInstructions’ during Permutation technique we used Amber’s data structure from appendix 5 to store the instructions.

# Even though Nudnik as it is can be boiled down to a pipeline of techniques inputting and outputting assembly files, we made the decision to provide an interface that would complete the entire build process for the user. Therefore Nudnik assumes the user (at least the user of the interface, not necessarily the module) has VSVC 2022 installed in the default locations.

# To be clear: we assume the input files have been compiled with VSVC 32 bit.Appendix:

1. Windows 10 License Agreement: <https://www.microsoft.com/en-us/Useterms/Retail/Windows/10/Useterms_Retail_Windows_10_English.htm>
2. List of anti-reverse engineering\disassembly technique listings:

<https://www.digitalwhisper.co.il/files/Zines/0x55/DW85-5-AntiDisasm.pdf>

<https://www.digitalwhisper.co.il/files/Zines/0x58/DW88-3-AntiReversing.pdf>

<https://youtu.be/iva16Bg5imQ?t=1558>

1. Sikorski, M., & Honig, A. (2012). 15. In Practical malware analysis: The hands-on guide to dissecting malicious software. essay, No Starch Press.
2. List of interesting somewhat related resources:

<https://www.youtube.com/watch?v=Wz_xJPN7lAY&ab_channel=Creel>  
<https://www.youtube.com/watch?v=HlUe0TUHOIc&ab_channel=Christiaan008>  
<https://youtu.be/KrksBdWcZgQ?t=1766>

1. Most common x86 instructions:

<https://www.strchr.com/x86_machine_code_statistics>

1. Fast random seed from set data structure source: <https://stackoverflow.com/questions/15993447/python-data-structure-for-efficient-add-remove-and-random-choice>