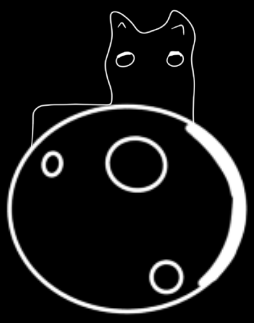


SkyPenguinLabs

Samples

How Mathematics Is Applied To Reverse Engineering

SPL-REC2

Powered by SkyPenguinLabs

Introduction

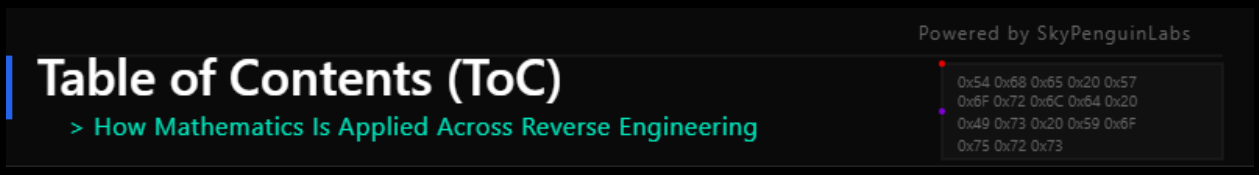
> How Mathematics Is Applied Across Reverse Engineering

0x54 0x68 0x65 0x20 0x57
0x6F 0x72 0x6C 0x64 0x20
0x49 0x73 0x20 0x59 0x6F
0x75 0x72 0x73

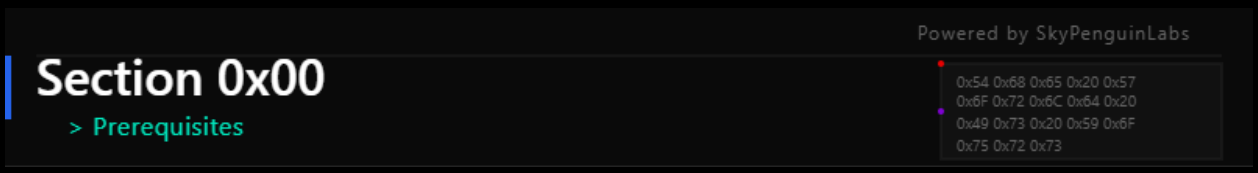
Welcome to the first official paid theoretical lesson hosted and sold by SkyPenguinLabs! In this course, you will be learning how mathematics is applied across many different areas of the reverse engineering spectrum, and will be learning how to utilize mathematics in a day-to-day routine.

> Readers Note: The goal of this course is to make sure that the reader understands the concept of our courses, the goal of offering a free course, alongside some information to aid them in their reverse engineering road path. While SkyPenguinLabs understands that you can not teach reverse engineering & expect everybody to pick up and apply these skills, we aim to ensure that the reader walks away having grasped at least something applicable to their existing path.

The following table of contents is listed on the next page which explores this course contents.



While these sections may seem small, they act as their own individual chapters or rather modules for the course content. This is how we are going to layout the other documents as well :D

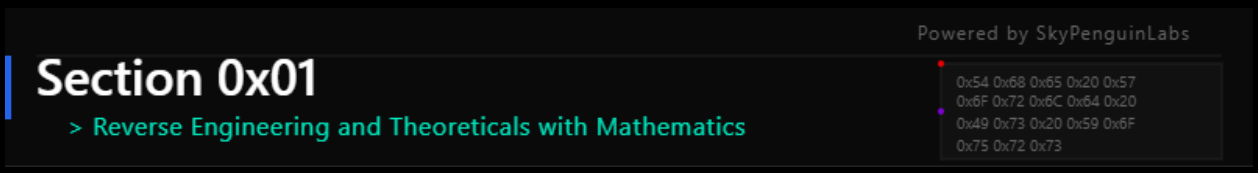


Before reading this article, it is important to know that you may come across references that are unknown and not further explained in the course. These references will be highlighted like **this** to remind you that it's worth noting down, and grabbing further context if you misunderstand before reading.

Of course, this is not fit for everybody, but it works for a general mass of people, so we are working with it.

Additionally, this article covers multiple spectrums of reverse engineering spaces but does not introduce you to what reverse engineering in those spaces is like as this course is not designed to be an introductory course and it is important that you as a reader keep this in mind as we would dislike misdirecting or having a reader interpret information in an incorrect way due to the way the course was perceived.

Additionally, since this is a free course, entertain the idea that these are the first few courses being put out and the formatting of the courses is deemed to change somewhere down the line throughout production and we are always open to community feedback on the readability, structure, and overall functionality of the document as a reader.



There is something I must state before going further into this, and it is simply that when we discuss the applications of mathematics across the reverse engineering space as a whole, we are grouping in anything from software reverse engineering, all the way down to network protocol reverse engineering & hardware reversing. The reason I state this is because mathematics is going to be used in every portion of reverse engineering, and focusing on just the broad implementations of it becomes impractical. Thus, this section covers applications from the most commonly discussed fields: *Software Reverse Engineering, Hardware Reverse Engineering, Cryptographic Reversing, and Network Protocol Reversing.*

> [Q] - *That being said, what exactly does mathematics have to do with reverse engineering? And I do mean, asides from the traditional instruction arithmetic?*

Well, to answer this question means really understanding what goes into the field of reverse engineering. Since it is such a broad field, we have to focus on very specific areas which have their own niche subsets of tools which we can explore to give us a general idea of what is used, where, and how.

The following sections below go through the 3 most common fields in the reverse engineering space, and cover how mathematics aids that field.

1. Software Reverse Engineering

Being the first, most talked about field in the reversing space, especially when malware reversing is brought into the circle, mathematics is quite undermined here, and while used, often not deeply understood even when being used because its not something every software reverse engineer is going to use.

Okay, let me actually explain what I mean here.

Often, when reverse engineering software, mathematics is not always going to be the most immediately applicable thing to

surface level reverse engineers. However, when the same reverse engineer attempts to understand the tools used to reverse engineer the software, we enter mathland like a motherf*cker!!!

But seriously, it's actually not that bad. To back the observation, I would like to propose that we cover 5 algorithms which are used within the IDA-Pro reverse engineering framework specifically in the components which involve disassembling the applications code, and pseudocode representations. Since some of the courses we will be releasing also cover optimization of applications, I figured I would also talk about some of the algorithms that may be used to also optimize pseudocode decompilers to make loading MASSIVE applications easier on the host.

These 3 major components are shown below.

Component Name	Description
Disassembler	*The frameworks application disassembler*
Pseudocode Decompiler	*The frameworks pseudo code translator that wraps around the disassembler*
CFG Graph	*IDA Pro's CFG Logic graph which breaks down the control flow of the logic in the program*

The sections begin on the next page.

IDA-Pro's Disassembler

If you are new to disassemblers, this next line might not make any sense, for those who aren't, it will. Unlike traditional disassemblers which use a **linear sweep** approach (*for beginners: essentially converting binary code to assembly code sequentially*) IDA relies on a recursive descent algorithm which follows paths in the program's execution.

Now, for a beginner, what I said makes no sense and you are probably wondering - how and why do disassemblers require algorithms?

Okay, think about this, seriously. Developers write code, say a language like C++, to develop Desktop applications. These apps are quite huge, lots of resources = more code, lots of libraries = more code, and lots of other localized thickness that increases the code size.

When the raw code gets thrown through the compiler, it gets translated into a representation and structure the compiler understands, which is then used to finalize the executable code as binary code. There is a LARGE amount there that is missed in the middle, and when you touch frameworks like LLVM, compilation becomes a multi-layered, and extremely over-complex clusterf*ck of code that eventually gets compiled to bytecode converted to machine code at runtime (*JIT*) or just being compiled to machine code.

So as you can imagine, we can not just write a simple program that opens the file, reads text data or the 0's and 1's and convert it back to tokens in widely used standard libraries based on basic pattern recognition. Even in scenarios where you can (*such as reversing custom bytecode implementations*), the implementation becomes EXTREMELY slow without an optimized algorithm that takes another angle of parsing data and viewing data in the application. And yes, you may be asking - is there not anything better? There is, but right now, Research from the University of Toronto and other institutions has shown that this approach achieves approximately 95.83% instruction

recovery accuracy, significantly higher than simpler methods.

This is because, when we tell a framework such as IDA or any program designed to open a binary file and read its contents, it needs to take the stream of data (*however the library or programs code read it into memory*) and tell the program how to tokenize it, parse it, structure it, format it, categorize it, analyze it, and ALSO fix any errors inside that occur during those processes in a reasonable timeframe. If it does not properly tell the program how to read the stream of data, it's just that, data and the program will spit out junk.

> **Informational Note:** To see the *'junk'* run `cat` on an ELF/PE file and see what I mean. Some data may be visible, but it's very, very minimal.

Let alone, IDA needs to first start DETECTING what type of file you are actually trying to disassemble, which in of-itself for how much IDA supports is a load of work! Truly. I respect it, hats off to THAT much substance abuse.

Now, back to the algorithm, the recursive descent algorithm works as a graph building process which explores the programs control flow paths. Starting with the program's entrypoint, decoding each byte from that point forward as an instruction in the detected architecture (***detecting architectures***). And yes, it **assumes** that ALL bytes are part of instructions (*as some SMEs may figure, without optimization or proper testing can result in issues*). When the algorithm detects control flow related instructions such as jumps, calls and returns, the program traces it. Almost like a web crawler on a web application which recursively traces through web resources to find and scan relevant files. Instead, here it recursively analyzes the target addresses of the instructions, descending into the code paths, the distinguishing between code and data which handles the possible misinterpretation issue that stems from assuming all bytes are part of instructions, and finally it handles non-linear code paths and any complex control flow

structures and related errors before finalizing the task. This algorithm also has a defined scope, the disassembly scope is anything within the logical references of the instructions. As mentioned, from the entrypoint down and anything cross referenced within logic.

To enhance the decompilation, IDA uses heuristics and value-set analysis to handle indirect jumps, which are jumps to addresses calculated at runtime, by tracking possible values of variables. IDA also identifies and analyzes function prologues/epilogues to discover new function entry points, further improving code coverage which assists with the scope extension as well. But this comes from optimizing the algorithm itself, and its implementation, rather than coming from the root functionality of the algorithm.

I find this quite neat, because if you check the algorithm out here, it's quite in-depth, and you can imagine the amount of sleepless nights the engineers endured for the industry haha. But just wait, because this is just the start LOLLLL! Gets me super hyped.

IDA-Pro's Pseudocode Decompiler & CFG Generation

Before we proceed, I must make sure the reader understands the fundamental difference between a decompiler and a disassembler, because man, I have heard way too many people mix it up or think it's the same.

While a disassembler converts binary code to assembly language, a decompiler takes this process much further by attempting to recover high-level code constructs but rather an IL or Intermediate Language which is interpretable in other logical senses. This is the "high level language" representation of the decompiled code.

Decompilers also are not just existent for your run of the mill low level software either, it's also for high level applications, such as those built out using frameworks like .NET with C# or Javascript with Electron. Some decompilers manage better than others because of the algorithms they choose to use. I

use the word _*algorithms*_ because similar to IDA, they use more than one algorithm to improve the process, and may collaborate with others to achieve higher results.

Simply, Hex-Rays is IDA Pro's decompilation engine, and it represents one of the more publicly available, advanced implementations of custom decompilers.

Understanding the decompiler IDA Pro uses is quite a pain if you know even a mid-sized amount of mathematics so we will not be covering every single topic of the decompiler, rather the primary aspects which incorporate algorithms to solve their problems.

In this case, the decompiler works through multiple steps of processes which incorporate different algorithms. The processes are listed below.

- **Microcode Generation -** Before the decompiler is able to start converting the disassembled program's assembly into a high level language, IDA transforms the assembly into microcode which is an intermediate representation of the assembly instructions on the surface. Microcode is represented with an instruction, and a set of operations (*microps*) for that micro instruction (*minsn*) representing the data elements being implemented. Sometimes, optimization is applied to microde generation with huge, complex sets of assembly instructions which dre decomposed into sequences of simpler microps.

- **CFG Construction** - When people think of CFG's, I feel like there is a small sum of people that actually try to envision how such concepts are represented programmatically. Just think, before you see the CFG displayed by IDA, the program needs to know the points to plot and what information to store where and how to relate it. To satisfy such tasks, it uses a large array of algorithms. Why tho? Well, think about it. Why exactly would you need an algorithm for this?