Emulators, Disassembly, and Anti-Debug tricks

赖纳文

December 17 2020

Abstract

This paper, as the title suggests, introduces a brief overview on several topics, including but not limited to:-

- Emulators[1]
- Disassemblers
- Some simple Anti-Debug tricks

For the purposes of demonstration, I will define a toy CPU architecture to emulate and to disassemble programs for that specific CPU architecture.

1 Introduction

I should start off with setting expectations. This is mostly just enough information to complete the challenge, and some extra information sprinkled in for your additional knowledge. It is not extensive however, and it is an extremely large topic that I could not possibly hope to explain, even just *only* all the interesting bits, in this one paper. I should also mention that I am in no way responsible for any false information in this paper, although I do not believe I have made any extraneous errors, I do not take any responsibility for any that may have slipped through. Let me first start off by defining what the terms in the title of the paper means.

1.1 Emulator

Let's start off with the *Emulator*. You, the reader, might be familiar with the type of Emulator used to play video games originally meant to be played on consoles, on some other device such as your laptop. Perhaps you have run DesMuME to play Nintendo DS games, or ePSXE to play Playstation games, bsnes for SNES games, Retroarch, PCSX2, DosBOX, and the list goes on. [2]. There

[1] There is some discussion to be had on the difference between *Emulators*, *Simulators* and *Interpreters* (and perhaps some other things like Virtual Machines), which will be discussed later on.

[2] The legality of these are up for debate, and which I would rather not get into here. The argument is that these are important for historical archival purposes, as in the present for example, obtaining a working NES device can be a difficult process, and eventually if no emulators exist, the NES games would be lost and can never be played again. As for modern games

are other applications however, such as in multi-platform development (e.g: developing for embedded devices), where for example a developer is in an x86 environment and needs to develop for a ARM environment in the case of developing for an embedded device. Or more often, an Android developer using Android Studio would use the Android Emulator to emulate an Android device.[3]. Although emulating the device may not always be necessary (they could always run or debug it remotely on a separate device), it can be useful, as it may just save the hassle of using another device, or if you don't happen to have a device of the targeted architecture.

I had promised an explanation of the difference between Simulators, Emulators and Interpreters. I found this stack-overflow answer most useful.[4]. The gist of it is that Emulators would attempt to act as a *replacement* for the device it is trying to emulate. A Simulator aims to *replicate down to the internal details* of the device it is trying to simulate. The term interpreter is separate from both Emulators and Simulators, and simply refers to the token-by-token parsing, decoding and running cycle. If you're wondering why Video Game Emulators don't use the word Simulators, it's because they don't aim to accurately simulate down to the silicon level the hardware implementation of the devices[5], because that would negatively impact on the performance of the emulator. Instead, they use a bunch of short cuts and tricks to make it run faster (e.g: frame skipping).

1.2 Disassembler

To understand what a disassembler is, you must first understand what an Assembler is. An assembler translates assembly code into machine code. For example, for an ARM CPU, the following is valid ARM code (Note that I use '0x' as a hexadecimal prefix for the rest of the document):-

```
label:
MOV r0, #0xc0
B label
```

Nothin very interesting, even without knowledge of the instructions, one can infer it is an infinite loop always settings register r0 to '0xc0' . The point is, the

being emulated, the argument is that once a consumer buys a video game, they do not simply buy a license to play the game, but have the right to have a 'backup' of it, and used however they see fit. Downloading copies of video games that you do not own however, has no such justification.

[3]This is a bit of a different comparison, Android Studio usually doesn't emulate an Android device running on an ARM chip, but rather an Android device running on an x86 chip, since that has the benefit of being faster since the host device is an x86 device, so no translation layer is necessary.

[4]Yes I am aware this isn't as reputable source as a peer-reviewed journal. This isn't a real paper, don't take it as such.

[5]There are actually some projects that *do* do this however, that is, aim to be as accurate to the original hardware as possible. But the name "emulator" sticks. Either because of the historical reasons of the word "Emulator" being used so often that is carries over, or because it is very difficult to accurately "simulate" everything and there are still compromises that the developer does for the sake of performance.

assembler will look through it line-by-line and assembly each instruction based on the opcode and the arguments used (There are a few possibly types of arguments, and there are different addressing modes, and it is possible to have no arguments at all. Some examples of addressing modes are Immediate Addressing, Indexed Addressing, Indirect Addressing, etc., you can search up 'Addressing Modes' for more information). [6]. So basically this is what the assembler does (in very simplified terms):-

- Look through line by line for non-empty lines
- If it is a label, keep track of where in memory it currently is in
- If it is an instruction, figure out what the opcode for that instruction is
- check if it has arguments, and put that alongside the opcode as fit
- Continue until done

For example, the assembler might see the opcode 'MOV" and while keeping in mind this is not the real ARM opcode, but rather, something I am making up for the purpose of demonstration, the opcode may be '0x01', and the corresponding argument for it to be register 'r0' could be '0x02', and finally the last byte can be simply '0xc0'. All-in-all, the assembler could assemble that instruction to '0x0102c0' [7]. Note that there exist CPUs with variable length opcodes as well, which would mean that perhaps a specific opcode would indicate that it is an 'extended' opcode with two words instead of just one. In my example, I also assumed that the instruction, the first argument, and the second argument, are each 1 byte large (2 hexadecimal digits). Although easier to see, in practice this may not be the case, and for example, perhaps only 4 bits is used to determine the instruction, etc.

From the knowledge gleaned from the Assembler, given the bytes of a program running on that CPU, one could create a Disassembler. It is simply the reverse process of the Assembler. Retrieve the bytes, analyse it, and spit out the mnemonic that is used for the assembly language. There is a few things that may be useful to you to know.

1.2.1 AND masking

If you recall, an AND operation returns true if and only if both inputs are true. We can apply the bitwise operator of AND to two numbers, which will convert the numbers to binary and compute the AND for each bit, and then store the result in a resulting number. For example, 10110101_2 AND 00001111_2 would result

[6] The label 'label' is not an instruction, and is instead kept track of and is used to help with determining where in memory to jump to when using branching instructions. Besides labels, some assembly languages also have directives used which tell the assembler some information.

[7] The positioning of what goes where is arbitrary and varies from different ISAs, also note this is an imaginary machine code I made. A real once would probably have another space for specifying the addressing mode, among other things.

in 00000101_2. An observant reader may notice that this is merely the lowest 4 bits of the former number. Indeed, the latter number seems to indicate which bits are the 'important' bits to keep, as the ones with 1s indicate that it will be kept 1 (or 0 if the number at that digit is 0), and if the latter is a 0, it would be 0 regardless of the other bit. This can be seen as the concept of 'masking', where the latter number indicates the 'important' bits to be kept. Note that this isn't exclusive to the lower 4 bits of a number. You can do this with any position of the number or of any length. For example, 11110000_2 AND 10101111_2 results in 10100000_2, and the lower 4 bits are changed to 0, while the upper 4 bits remain the same. [8] If you're wondering why this is useful for disassemblers, recall that assemblers would assemble an instruction and combine the instruction opcode with its arguments. At the initial stage, we do not care about what the arguments are, and simply want to know what the instruction. And so if the pattern is 0xAABBCC where AA is the instruction opcode, BB is the first argument, and CC is the second argument, we can simply do AND '0xFF0000' to get just '0xAA0000' since we do not care about BB and CC yet. Similarly, once we *do* care about BB and CC, we can do a similar trick to get BB and CC by themselves.

[8] Notice how I swapped orders, this works because AND is commutative

1.2.2 Bit Shifting

You probably realised something in the previous section. Once we obtain BB, it would be in the format '0x00BB00', when we simply want '0xBB'! Using simple maths, you may realise for example, a number like 600 can be changed to just 6 by dividing it by 100. This can be decomposed into a general solution as "dividing it by 10 to the power of < number of digits you want to shift right >", assuming that the rightmost digits are 0 (or you use integer division). The same applies to numbers in base 2, you *can* simply "divide by 2 to the power of < number of digits you want to shift right >", but most languages provide a quicker way to do this with a bitwise shift ($\gg$and $\ll$). As so you can simply shift '0x00BB00' right by one byte (8 binary digits- 8 bits) and you should obtain he result you are looking for.

You may be wondering about the case of labels. In this case, wouldn't disassembling a label simply give the address of the label, and not the name of the label itself? Alas, label names are usually not preserved in the assembly process, so it cannot be retrieved by the disassembler. The best we can do is for the disassembler to keep track of where it jumps to and assign a temporary autogenerated label name for it.

2 Emulator

Assume we have a program meant to be executed in that architecture, and assuming we don't have a physical ver-

sion of a machine running this toy architecture, one could simply use *static analysis* using the disassembler to find out what it does, by manually tracing the code in her/his mind. However, We can go one step further; we can utilise *dynamic analysis* by emulating/simulating the program. You already know how the disassembler works, and that is very useful for writing an Emulator. The emulator would likely have a area to simulate the memory of the machine. It will also have to have variables to represent the registers of the machine. It would then, like the real machine, simulate a "Fetch", "Decode" and "Execute" cycle until the program ends. The 'Fetch' and 'Decode' process remains largely the same as in the Disassembly process. The Execute part is a bit more interesting. Usually it would use a giant switch case or a lookup table of function pointers to execute it. The code in it would then do what they're supposed to do, i.e: modify registers, store things to RAM, display things on the screen, etc. Most crucially, it would increment the program counter[9] so that in the next iteration of the Fetch-Decode-Execute Cycle, it would fetch the next instruction (usually it is accessed by RAM[pc])

[9]In some ISAs, this is called the Instruction pointer, not to be confused with the Instruction Register in some ISAs, which hold the contents of the instruction itself

3 Anti-Debugging Tricks

Completely different from the other sections, I just added this section because I implemented some simple anti-debug tricks in my challenge.

Debugging the process formally defined as when a developer bashes their against the wall repeatedly in frustration at 2:12am in the morning, talking to their trusty rubber duck[10] and praying to whatever god they believe in, before eventually heading to stackoverflow or an obscure forum thread where they find someone else had the same problem, but the question was either closed as being a Duplicate of a completely separate issue, or the OP had 'figured it out, nvm' without posting what their solution was, and then realising 3 hours later the problem was a missing semi-colon.

Jokes aside, a debugger is an invaluable tool allowing a developer to step through their code line-by-line to find the symptom of an error they might be encountering. Reverse-engineers usually do this with even assembly code, as there is no way to hide that[11].

Since this tool is so invaluable, a malicious party who does not want their code to be debugged can take several steps to make it harder for it to be done so. So how would one detect a debugging tool

[10]This is a real thing called rubber duck debugging

[11]There is a variety of ways to *obfuscate* it however. Insertion random unnecessary instructions, adding a compressor or decryption to generate instructions dynamically to avoid static analysis, etc.

being used?

““Magic always leaves traces,” said Dumbledore, as the boat hit the bank with a gentle bump, “sometimes very distinctive traces…””

If you would forgive my quote, do note that the point I am trying to make is that using the debugger would leave traces. One such example that I will not go into detail in is the environment variables. Usually using a debugger such as gdb would set up some additionally environment variables for the debugger program itself to use, but this environment variables can be accessed by the child process as well (it is a child process because the debugger as the parent is the process starts the child process- otherwise it would not have access to be able to debug it!), and so the program can simply check for those environment variables.

Sometimes, there is even an API to detect if the debugger is running. In Windows, there is an IsDebuggerPresent function that returns true if the debugger is running. Another thing one could do is check if the process has a parent process. Or, since a debugger is being used, it would be possible to do some sort of timing attack to figure out if debugger is used since it would have slightly different timing and wouldn't be as fast with a debugger on, though this is risky as it could simply be the case of a separate computer just being slower.

The next question is what to do once you detect a debugger is present. The obvious answer would be to exit the program to not allow the user to do anything else. But there is a more evil thing to do. It can break itself in subtle ways, and 'pretend' to work. This would waste the person's time as they grow more and more frustrated trying to figure out what is wrong. This method has been used by game developers in the past as a anti-piracy tactic, with my favourite example being in Earthbound where it makes the game much harder, and if you do end up getting near the end, right before the climax it would freeze up and delete your save file! Another excellent example is a video game called Game Dev Tycoon where players create video games and sell it, and if the game detecst it is a pirated version, it would tell the players their games are being pirated and sales are plummeting!

4 Further Reading

See the References section. The "Ultimate Game boy talk" also has various other "Ultimate X talks" in CCC conference talks. The MGBA blog is also good.

References

[] Mgba io (gameboy emulator blog). URL `https://mgba.io/tag/emulation`.

[] Asami. https://book.rvemu.app/index.html.

[] CTurt. Cinoop gameboy emulator. URL `https://cturt.github.io/cinoop.html`.

[] V. M. del Barrio. *Study of the techniques for emulation programming*. URL http://www.xsim.com/papers/Bario.2001.emubook.pdf.

[] L. M. (M.Sc.). How to write an emulator (chip-8 interpreter). URL http://www.multigesture.net/articles/how-to-write-an-emulator-chip-8-interpreter/.

[] I. Nazar. Gameboy emulation in javascript. URL http://imrannazar.com/GameBoy-Emulation-in-JavaScript:-The-CPU.

[] M. Steil. Ultimate gameboy talk. URL https://www.youtube.com/watch?v=HyzD8pNlpwI.