Decompilation, Shmecompilation

An Introduction to Matching and Non-matching Decompilation Practices

Charles Averill

Computer Security Group
The University of Texas at Dallas

October 19, 2022

What is Decomp?

- A subset of Reverse Engineering (**reversing**)
- Reversing generally deals with understanding the operation and structure of **binaries** (machine code)
- **Decomp** goes a step further and aims to reconstruct a set of source files that replicate the behavior of the binary when re-compiled
- **Matching decompilation** goes another step further and aims to reconstruct source files such that when they are compiled the binary is identical to the original binary

ARM Recap

- Decompilation relies heavily on a good understanding of assembly code, such as x86, ARM, or MIPS
- Assembly instructions work by moving chunks of data around on the CPU between storage containers called **registers**
- `int r3 = r1 + r2;`

  ```
  add r3, r1, r2
  ```
- `int r0 = 5; while(r0 != 0) { /*do something*/ }`

  ```
  mov r0, #0x5
  my_loop_label:
  // do something
  sub r0, r0, -1
  cmp r0, 0
  bne my_loop_label
  ```

Let's Try It Out

- `https://decomp.me/scratch/dojn4`
- Utilizes function calling, multiple integer widths, pointers, conditions, return statements
- More ARM stuff
    - `push { argument+ }` - Instruction to push arguments onto stack memory
    - `lr` - "Link Register", holds return program counter (current step in program)
    - `lsl` - "Logical shift left"
    - `ldr` - "Load", loads a value given an adress and a byte offset
    - `bl` - "Branch and Link", sets `lr` to the adress of the next instruction, and branches to a label

Solution

```c
extern u8 gUnknown_09AF3790[];

extern void* sub_0800289C(u8*, s32);

#define NULL 0

u8* sub_08039B24(u16 param_1) {
    u16* t1 = (u16*)sub_0800289C(gUnknown_09AF3790, 0x24);
    if (t1) {
        u8* t2 = (u8*)sub_0800289C(gUnknown_09AF3790, 0x25);
        return t2 + t1[param_1];
    } else {
        return 0;
    }
}
```

So What?

- The previous code snippet was from the Game Boy Advance game "Mother 3", I picked this example because it was easy
- You can decompile better games (and other things), like Zelda (`https://github.com/CharlesAverill/oot_le`)
- The goal behind decompilation projects is threefold: modding (straightforward), preservation (straightforward), and understanding the development of the binary (not straightforward)
- Understanding the structure of the binary well enough to perform a matching decompilation of it gives you insight into the growth of the codebase over time
- If the binary was developed by an organization, decompilation almost always gives insight into how other binaries distributed by the organization are built

So What?

- Is decomp a red team or blue team activity? Sort of both!
- Red team decomp looks like trying to figure out how a program works to see if you can break it or similar software
- Blue team decomp looks like reassembling viruses or other malicious payloads to determine what vulnerabilities they target so that defensive patches can be written

That was the Easy Part!

- Non-matching decompilation can be heavily automated for the most part with software like IDA and Ghidra
- That's all good and well, but how useful is this? Not very!

  Ocarina of Time Player Controller Code
- Documentation takes longer than decompilation for large projects
- If you don't know what an individual function or even an individual variable is for, it's very hard to analyze large program structures
- OOT decomp (matching) took almost 2 years to hand-decompile (finished in Novemeber 2021). Documentation was taking place during decompilation, now it's just documentation. Game will likely not be fully documented for a few years, even with more attention now

Documentation

- Some aspects of compilers and certain program features make parts of documentation easier
- Filenames can be embedded in user-defined asserts/errors

  OOT Player Code
- OOT decomp pulled the entire project directory structure using data embedded in the binary by IDO, Nintendo's in-house C Compiler from the 1990s

Legal Issues

- Nintendo probably doesn't like that we can rebuild their games and modify them as we please and port them to modern platforms easily
- Decomp is in a weird gray area. You sold me the binary, I disassembled it, why shouldn't I be able to look at the assembly? If I can look at the assembly, why shouldn't I be able to write C code that compiles to something similar/identical? I wrote the code!
- Then again, the binary includes non-code assets, like 3D models, textures, and music that are very clearly Nintendo's IP
- Answer? Decomp projects make **you** supply a ROM file to extract assets from, and they only distribute decompiled code.
- Does this solve any legal issues? Not really, but it has kept the Nintendo ninjas away for now

Controversies

- Sometimes, original source code gets leaked (see the Gigaleak: `https://en.wikipedia.org/wiki/Nintendo_data_leak`)
- This includes the code Nintendo wrote, so it should very clearly be off-limits
- Suprise, some people disagree! Major breaches of this policy have been encountered, where people secretly reference gigaleak code when decompiling / documenting games
- Poses a huge issue when discovered: do we uproot the git tree and revert thousands of hours of progress, because we built on top of leak data for months? Do we just modify the current branch and tell people not to look in the history? Do nothing and allow leaks? Each option has huge cons, and nobody agrees on what to do about it