ENGR-E 399/599: Embedded systems reverse engineering

Lecture 11: STM32F4, real-time operating systems, and object-oriented code

Austin Roach
ahroach@iu.edu

March 31, 2022

Mystery function

```
0x00000000    0030a0e3    mov r3, 0
0x00000004    ff1001e2    and r1, r1, 0xff
0x00000008    00c0a0e1    mov ip, r0
0x0000000c    0120d0e4    ldrb r2, [r0], 1
0x00000010    010052e1    cmp r2, r1
0x00000014    0c30a001    moveq r3, ip
0x00000018    000052e3    cmp r2, 0
0x0000001c    f9ffff1a    bne 8
0x00000020    0300a0e1    mov r0, r3
0x00000024    1eff2fe1    bx lr
```

- How many arguments does the function take?
- What are the types of the arguments?
- What does the function do?
- What does the function return?

Mystery function revealed

```
char *strrchr(const char *s, int c)
```

Description

The `strrchr()` function returns a pointer to the *last* occurrence of the character `c` in the string `s`.

Return value

The `strrchr()` function returns a pointer to the matched character or NULL if the character is not found. The terminating null byte is considered part of the string, so that if `c` is specified as '\0', this function returns a pointer to the terminator.

Today's plan

- STM32F4
  - Memory map and I/O
- Real-time operating systems (RTOS)
- Analyzing object-oriented (C++) code
  - Object initialization
  - Method calls
  - Inheritance
  - Virtual method tables (indirect function calls)

STM32F429

- 32-bit ARM Cortex-M4 processor
  - Cortex-M = Microcontroller variant
  - No memory management unit
    - Memory addresses are physical rather than virtual addresses
    - No support for compartmentalizing processes in their own address spaces
    - Memory protection unit for some access controls
  - Operates only in Thumb mode (supports Thumb-1 and Thumb-2 instructions)
- More sophisticated peripherals than AVR microcontroller
  - DAC
  - Ethernet
- For assignment 5, runs a simple real-time operating system

STM32F429 memory map

See STM32F427xx STM32F429xx datasheet (stm32f427vg-056239.pdf)

- Memory map overview: Figure 19 (Page 86)
- MMIO boundary addresses: Table 13 (Page 87)

STM32F429 I/O

Generic descriptions of I/O blocks in datasheet Section 3 (Functional Overview)

Detailed specifications and register descriptions in RM0090 reference manual

Information used in Ghidra script (st_f429zi_ghidra.py)

Real-time operating systems

Real-time operating system goals

- Provide operating-system-like abstractions
  - Resource sharing between tasks
  - (Sometimes) Filesystem, network stack, etc.
- With real-time constraints:
  - Hard real-time system: Failure to respect timing constraint is a system failure
    - Industrial robot
  - Soft real-time system: Failure to respect timing constraints disrupts operation
    - Gaming console

Some RTOSes

- Unix-like operating systems
  - QNX Neutrino
- Less process separation than Unix systems, but allows use of high-performance features like SMP
  - VxWorks
- Lightweight, task-oriented
  - FreeRTOS
  - MbedOS

Task-oriented real-time operating system architecture

- Extremely simple kernel
  - Start, schedule, and stop threads
  - Mutexes, semaphores, message queues
- No shell
  - But you could write a task to provide a command-line interface
- No filesystem by default
  - Might optionally provide a very simple filesystem
- No networking by default
  - Optional tasks can provide this

Challenges for reverse-engineering

- Fairly complex
  - 10s of thousands of instructions
- The functionality that you care about may only be a few percent of these instructions
- No clear division of the code
  - System start-up will start tasks individually
  - But all the code is smushed into the flash memory together
  - No division into separate files as on embedded Linux system

How to find the code that you care about

- String references
  - Sometimes `assert()` statements that state line, file, function, or condition
  - Can find references through static analysis (Ghidra cross-references) or dynamic analysis (watchpoints)
- Search nonvolatile memory for data that you recognize, look for references
  - Can find references through static analysis (Ghidra cross-references) or dynamic analysis (watchpoints)
- Search volatile memory for user-supplied data, look for references
  - Can find references through static analysis (Ghidra cross-references) or dynamic analysis (watchpoints)
  - This is something you'll do in assignment 5, using watchpoints

Object-oriented code

Why do we care?

In general:

- Lots of software written using object-oriented programming languages (C++)
- Use leads to a high amount of indirection
  - Objects passed by reference – indirect data accesses
  - Virtual method tables – indirect function calls
  - Inheritance and polymorphism layer on complexity
- This indirection tends to confuse reverse-engineering tools
  - Insufficient information in local program context to resolve references
  - You get to fix these things by hand!
- (But intact mangled symbols can give helpful type information...)

You will encounter effects of object-oriented code in Assignment 5

- Methods have an implicit first argument (passed in `R0`): pointer to object (`this`)
- Object is simply a structure

Offset	**Length**	**Description**
0x0	0x4	num_legs (int)
0x4	0x1	has_tail (bool)
0x5	0x3	Padding

Virtual methods

- Virtual methods support derived classes
- Function pointers arranged in a virtual methods table ('vtable' or 'vftable')
- Pointer to vtable is first entry in object structure
- Can be multiple vtable entries in cases of multiple inheritance

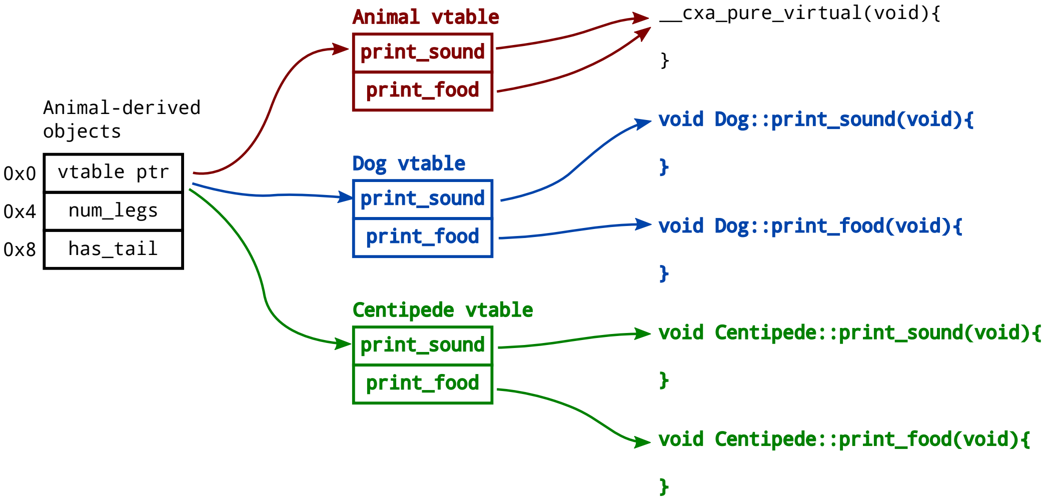
Object structure:

Offset	**Length**	**Description**
0x0	0x4	pointer to vtable
0x4	0x4	num_legs (int)
0x8	0x1	has_tail (bool)
0x9	0x3	Padding

Vtable:

Offset	**Length**	**Description**
0x0	0x4	pointer to print_sound()
0x4	0x4	pointer to print_food()

Object structure



Other reasons for indirect calls

- Tables of function pointers
  - Not created automatically to support C++ virtual methods
- Registered callback functions
- Dynamic linking
  - PLT essentially accesses a table of function pointers
- Plug-ins
- Self-mutating code

You might be able to find the call targets through static analysis, but sometimes it's easiest to just set a breakpoint.

References

- P.V. Sanabal and M.V. Yason: "Reversing C++", https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf
- docs/GhidraClass/Advanced/improvingDisassemblyAndDecompilation.pdf in your Ghidra installation directory
- I. Skochinsky, "Practical C++ Decompilation", RECon 2011, https://www.youtube.com/watch?v=efkLG8-G3J0, with slides here: http://www.hexblog.com/wp-content/uploads/2011/08/Recon-2011-Skochinsky.pdf
- E.J. Schwartz, "Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables", CCS 2018, https://dl.acm.org/doi/10.1145/3243734.3243793.
- R. Rolles, "Automation Techniques in C++ Reverse Engineering": https://www.msreverseengineering.com/blog/2019/8/5/automation-techniques-in-c-reverse-engineering
- OOanalyzer: https://github.com/cmu-sei/pharos/blob/master/tools/ooanalyzer/ooanalyzer.pod

- *Write your own examples and analyze them!*