ENGR-E 399/599: Embedded systems reverse engineering

Lecture 02: Standard library function identification

Austin Roach
ahroach@iu.edu

January 20, 2022

Today's plan

- Introduction to Assignment 1
- Manual identification of standard library functions
- Automated techniques for function identification

Introduction to Assignment 1

Why do we want to identify standard library functions?

Standard library functions are called from many of the parts of the code

- memcpy()
- strcpy()
- strstr()
- memset()
- printf()
- sprintf()
- ...

In embedded applications there is often no metadata to provide function names

Identifying standard library functions can be key to understanding application-specific functionality

Identifying functionality

Four key questions:

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

How many arguments does the function take?

- Calling convention specifies that arguments are passed in registers (r0-r3), then onthe stack
- Look for registers or memory that are accessed before they are modified by the function

```
                        undefined FUN_00000000()
      undefined            r0:1              <RETURN>
                        FUN_00000000
ram:00000000 01 30 40 e2     sub        r3,r0,#0x1
ram:00000004 02 20 81 e0     add        r2,r1,r2

                        LAB_00000008                            XREF[1]:     00000018(j)
ram:00000008 02 00 51 e1     cmp        r1,r2
ram:0000000c 1e ff 2f 01     bxeq       lr
ram:00000010 01 c0 d1 e4     ldrb       r12,[r1],#0x1
ram:00000014 01 c0 e3 e5     strb       r12,[r3,#0x1]!
ram:00000018 fa ff ff ea     b          LAB_00000008
```

What are the types of the arguments?

- Is the argument used as an *address* for a memory access?
  - Argument is a pointer
- What is the width of the memory access?
  - Byte, half-word, word? Or are there hints that it's some other object type?
- Is the value added to a pointer or used for some other arithmetic?
  - Integer value; look for signed/unsigned comparisons if need to determine signedness.

```
                        undefined FUN_00000000()
     undefined             r0:1           <RETURN>
                        FUN_00000000
ram:00000000 01 30 40 e2     sub        r3,r0,#0x1
ram:00000004 02 20 81 e0     add        r2,r1,r2

                        LAB_00000008                               XREF[1]:     00000018(j)
ram:00000008 02 00 51 e1     cmp        r1,r2
ram:0000000c 1e ff 2f 01     bxeq       lr
ram:00000010 01 c0 d1 e4     ldrb       r12,[r1],#0x1
ram:00000014 01 c0 e3 e5     strb       r12,[r3,#0x1]!
ram:00000018 fa ff ff ea     b          LAB_00000008
```

What does the function do?

- What is the logic of the function?
  - Evaluate loop structure/stopping conditions
  - Can help to provide some example value for the arguments, work through how the function behaves

```
                         undefined FUN_00000000()
     undefined             r0:1          <RETURN>
                         FUN_00000000
ram:00000000 01 30 40 e2      sub         r3,r0,#0x1
ram:00000004 02 20 81 e0      add         r2,r1,r2

                         LAB_00000008                                 XREF[1]:     00000018(j)
ram:00000008 02 00 51 e1      cmp         r1,r2
ram:0000000c 1e ff 2f 01      bxeq        lr
ram:00000010 01 c0 d1 e4      ldrb        r12,[r1],#0x1
ram:00000014 01 c0 e3 e5      strb        r12,[r3,#0x1]!
ram:00000018 fa ff ff ea      b           LAB_00000008
```

What does the function return?

- Calling convention specifies that return value is passed in r0
- What is the type of the return value?
- How does it relate to the input arguments?

```
                        undefined FUN_00000000()
        undefined          r0:1            <RETURN>
                        FUN_00000000
ram:00000000 01 30 40 e2     sub        r3,r0,#0x1
ram:00000004 02 20 81 e0     add        r2,r1,r2

                        LAB_00000008                           XREF[1]:     00000018(j)
ram:00000008 02 00 51 e1     cmp        r1,r2
ram:0000000c 1e ff 2f 01     bxeq       lr
ram:00000010 01 c0 d1 e4     ldrb       r12,[r1],#0x1
ram:00000014 01 c0 e3 e5     strb       r12,[r3,#0x1]!
ram:00000018 fa ff ff ea     b          LAB_00000008
```

Putting it together

```
                          undefined FUN_00000000()
        undefined             r0:1            <RETURN>
                          FUN_00000000
ram:00000000 01 30 40 e2     sub        r3,r0,#0x1
ram:00000004 02 20 81 e0     add        r2,r1,r2

                          LAB_00000008                          XREF[1]:     00000018(j)
ram:00000008 02 00 51 e1     cmp        r1,r2
ram:0000000c 1e ff 2f 01     bxeq       lr
ram:00000010 01 c0 d1 e4     ldrb       r12,[r1],#0x1
ram:00000014 01 c0 e3 e5     strb       r12,[r3,#0x1]!
ram:00000018 fa ff ff ea     b          LAB_00000008
```

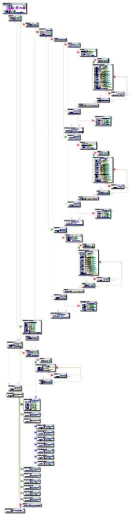
```
void *memcpy(void *dest, const void *src, size_t n);
```

Description

The `memcpy()` function copies `n` bytes from memory area `src` to memory area `dest`. The memory areas must not overlap. The `memcpy()` function returns a pointer to `dest`.

Automating function identification

Mystery function



But we already saw this!

```
                      undefined FUN_00000000()
     undefined           r0:1          <RETURN>
                      FUN_00000000
ram:00000000 01 30 40 e2     sub        r3,r0,#0x1
ram:00000004 02 20 81 e0     add        r2,r1,r2

                      LAB_00000008                              XREF[1]:     00000018(j)
ram:00000008 02 00 51 e1     cmp        r1,r2
ram:0000000c 1e ff 2f 01     bxeq       lr
ram:00000010 01 c0 d1 e4     ldrb       r12,[r1],#0x1
ram:00000014 01 c0 e3 e5     strb       r12,[r3,#0x1]!
ram:00000018 fa ff ff ea     b          LAB_00000008
```

What is going on?!

- The version that we looked at looked at before is optimized for *size*
  - uclibc
- This new version is optimized for *performance*
  - glibc

Automated function identification goals

Automatic identification of functions that implement known functionality, including:

- Standard library functions
- Functions encountered in previous analysis of similar systems
- Functions derived from source code that we know
- Functions shared between multiple code bases
  - Perhaps a system with better debug access than the system we are examining
- Functions that implement known algorithms

Many similarities to the problem of ‘clone detection’

Signature-based approaches

Approach

For each function:

- Compute a signature based on the instructions of the function
- Ignore bytes that may be variable
  - Fixed data addresses, or some constants for example
- Match to stored signatures in a database

Used by Ghidra's Function ID, IDA Pro's FLIRT, and radare2's zignatures

Example function

```
int power(int base, unsigned int exponent)
{
        int result = base;

        if (exponent == 0) return 1;

        for (int i = 1; i < exponent; i++) {
                result *= base;
        }
        return result;
}
```

Function disassembly

```
0x00000570    000051e3    cmp r1, 0
0x00000574    0120a003    moveq r2, 1
0x00000578    0700000a    beq 0x59c
0x0000057c    010051e3    cmp r1, 1
0x00000580    0700009a    bls 0x5a4
0x00000584    0020a0e1    mov r2, r0
0x00000588    0130a0e3    mov r3, 1
0x0000058c    900202e0    mul r2, r0, r2
0x00000590    013083e2    add r3, r3, 1
0x00000594    010053e1    cmp r3, r1
0x00000598    fbffff1a    bne 0x58c
0x0000059c    0200a0e1    mov r0, r2
0x000005a0    1eff2fe1    bx lr
0x000005a4    0020a0e1    mov r2, r0
0x000005a8    fbffffea    b 0x59c
```

Example zignature

```
sym.power:
  bytes: 000051e30120a0030700000a010051e30700009a0020a0e10130a0
         e3900202e0013083e2010053e1fbffff1a0200a0e11eff2fe10020a0e1fbffffea
  mask: fffffffffffffffffffffffffffffffffffffffffffffffffffffff
         fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  graph: cc=4 nbbs=6 edges=8 ebbs=1 bbsum=60
  addr: 0x00000570
  vars: r1, r0
  bbhash: 6b321b8192a577c1fb7870f10dcbd95ff645cf9faa364d8a504d4a5d480fcee1
```

Applicability

- Statically linked library functions
  - For widely distributed libraries
- (Nearly) exact copies of functions encountered in some previous analysis

Limitations

- Algorithm is unaware of functionality
  - Different implementations of same algorithm produce different signatures
- Signatures not portable across architectures
- Fragile with respect to compiler optimizations, etc.

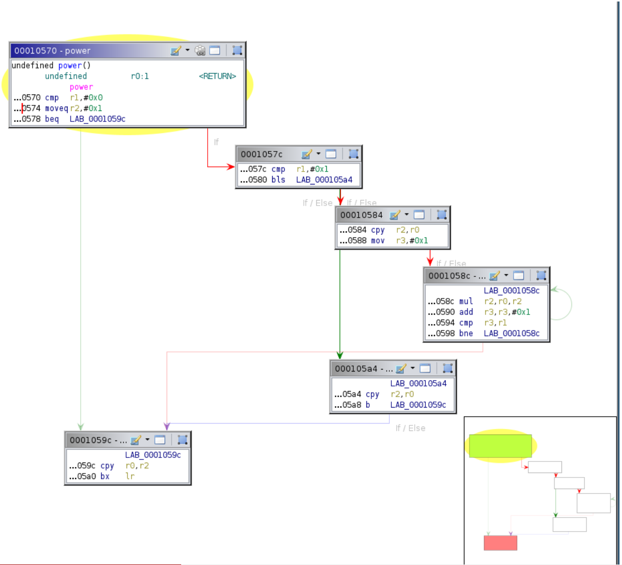
Control-flow graph analysis

Approach

- Serialize control-flow graph topology in some way that allows comparison
- Possibly supplement with some information about basic blocks
  - Hashes
  - Standard library calls
  - Instruction mnemonics
- Compare control-flow graphs with adjustable threshold

Used in radare2's zignatures, Zynamics Bindiff, and other software diffing approaches

Function CFG



Example zignature

```
sym.power:
  bytes: 000051e30120a0030700000a010051e30700009a0020a0e10130a0
         e3900202e0013083e2010053e1fbffff1a0200a0e11eff2fe10020a0e1fbffffea
  mask: fffffffffffffffffffffffffffffffffffffffffffffffffffffff
         fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  graph: cc=4 nbbs=6 edges=8 ebbs=1 bbsum=60
  addr: 0x00000570
  vars: r1, r0
  bbhash: 6b321b8192a577c1fb7870f10dcbd95ff645cf9faa364d8a504d4a5d480fcee1
```

Applicability

- Nearly the same as byte signatures
- Compilation with different compilers, if CFG structure determined by implementation in source code and not compiler optimizations
- Possibility of cross-architecture comparison
  - As long as architectural variations don't manifest too strongly in CFG

Limitations

- Compiler optimizations can heavily modify CFG
- Different implementations of similar algorithms may also produce different CFGs
- Cross-architecture support can be fragile

Layers of heuristics

Approach

- Domain of specialization differencing or version-tracking tools
- Attempt to increase matching opportunities through a host of heuristics
- Examples:
  - Zynamics Bindiff: https://www.zynamics.com/bindiff/manual/#chapUnderstanding
  - Ghidra Version Tracking tool

FunctionSimSearch

- SimHash - Locality sensitive hashing
- Inputs:
  - Subgraphs of control-flow graph
  - n-grams of mnemonics of disassembled instructions
- Machine learning to determine weights
- Weighted signature matching

Description: https://github.com/googleprojectzero/functionsimsearch/blob/master/doc/01-motivation-and-overview.md

Cryptographic constant search

Approach

- Cryptographic algorithms often have standardized defined constants
- Search program for constants matching cryptographic algorithms
- Exmaple: FindCrypt https://github.com/polymorf/findcrypt-yara
- FindCrypt-Ghidra https://github.com/d3v1l401/FindCrypt-Ghidra

Emulation

Approach

- Identify functions by *behavior* rather than instructions or structure
- Example: Sibyl (`https://github.com/cea-sec/Sibyl`)
- Assumes a calling convention, and provides specific tests to identify standard library functions by behavior
- Defined tests: (`https://github.com/cea-sec/Sibyl/tree/master/sibyl/test`)

Advantages

- Evaluating the *semantics* of the function
- Tolerant of implementation differences, compiler optimizations, and obfuscation

Limitations

- Assumes a calling convention
- Need the ability to emulate the target architecture
- Test cases are manually generated
- Without carefully defined test cases, can confuse functions that are similar in functionality

Demo

```
$ docker pull commial/sibyl
$ docker run --rm -it \
  -v /path/to/lectures/lecture-02/demo:/work \
  commial/sibyl /bin/bash
$ sibyl find -a arml /work/memcpy_simple 0x0
$ sibyl find -a arml /work/memcpy_complex 0x0
```

Data flow analysis

Approach

P. Lestringant, F. Guihéry, and P.-A. Fouque, “Automated identification of cryptographic primitives in binary code with data flow graph isomorphism”, Proc. 10th ACM Symp. Inf. Comp. and Comm. Sec. (ASIA CCS ’15), 2015, pp.203-214.

- Identification of algorithms through operations performed on data
- Simplication (‘normalization’) of data flow graphs to allow comparison regardless of implementation variation and compiler optimizations

https://www.amossys.fr/upload/asiaccs15_Automated_Identification_Of_Cryptographic_Primitives_In_Binary_Code_With_Data_Flow_Graph_Isomorphism_paper.pdf

Advantages

- No need to be able to execute or instrument the code under analysis
- No assumption of calling convention
- Resilient to differences in implementation and compilation
- Possibility of cross-architecture comparisons (if equivalent operations on data)

Limitations

- Most appropriate for computationally complex algorithms (like cryptography)
- Manual signature generation
- Normalization approach may not be robust to deliberate obfuscation

Synthesizable adapters

Approach

V. Sharma, K. Hietala, and S. McCamant, "Finding substitutable binary code by synthesizing adapters", IEEE 11th Conf. on Soft. Test., Ver., and Valid. (ICST), 2018.

- Test various adapters applied to functions in an attempt to demonstrate equivalence
- Use symbolic execution to identify counterexamples that show that adapted functions are not equivalent
- Use symbolic execution to generate new adapters to satisfy all previously generated test cases

https://arxiv.org/pdf/1707.01536.pdf

Advantages

- No assumption of calling convention
- Tolerant of implementation, compiler, and optimization differences
- Resistant to obfuscation

Limitations

- Computationally demanding

Key takeaways

- Multiple approaches to automatically identifying functions
  - Range from straightforward (version tracking based on simple heuristics) to exotic (dedicated tools using theorem provers)
- In many emedded projects you can identify common functionality faster than you can run an automated tool
- For projects with complex functionality or where you wish to translate prevoius analysis, automated tools can be a huge timesaver.