GRAMMATECH

Capabilities Labeling

HCSS 2023 – May 10, 2023

Greg Nelson (gnelson@grammatech.com)
Coauthor Denis Gopan (gopan@grammatech.com)

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Defining the Problem

Have you ever needed to understand someone else's code?

```
/* Calculate a checksum. I don't know the name
of the one we're using but this seems to work. */
uint16_t crc16(const uint8_t* buffer, size_t size) {
  uint16_t tmp, crc = 0xffff;

  for (size_t i=0; i < size; i++) {
    tmp = (crc >> 8) ^ buffer[i];
    crc = (crc << 8) ^ crc_table[tmp];
  }

  return crc;
}
```

Defining the Problem

What if you didn’t even have source code?

```
00000000 <proc_800213c>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd01c        sub     sp, sp, #28
   c:   e50b0018        str     r0, [fp, #-24]  ; 0xffffffe8
  10:   e50b101c        str     r1, [fp, #-28]  ; 0xffffffe4
  14:   e3e03000        mvn     r3, #0
  18:   e14b30b6        strh    r3, [fp, #-6]
  1c:   e3a03000        mov     r3, #0
  20:   e50b300c        str     r3, [fp, #-12]
  24:   ea00001d        b       a0 <proc_800213c+0xa0>
  28:   e15b30b6        ldrh    r3, [fp, #-6]
  2c:   e1a03423        lsr     r3, r3, #8
  30:   e1a03803        lsl     r3, r3, #16
  34:   e1a02823        lsr     r2, r3, #16
  38:   e51b300c        ldr     r3, [fp, #-12]
  3c:   e51b1018        ldr     r1, [fp, #-24]  ; 0xffffffe8
  40:   e0813003        add     r3, r1, r3
  44:   e5d33000        ldrb    r3, [r3]
  48:   e1a03803        lsl     r3, r3, #16
  4c:   e1a03823        lsr     r3, r3, #16
  50:   e0233002        eor     r3, r3, r2
  54:   e14b30be        strh    r3, [fp, #-14]
  58:   e15b30b6        ldrh    r3, [fp, #-6]
  5c:   e1a03403        lsl     r3, r3, #8
  60:   e1a03803        lsl     r3, r3, #16
  64:   e1a02843        asr     r2, r3, #16
  68:   e15b30be        ldrh    r3, [fp, #-14]
  6c:   e59f1050        ldr     r1, [pc, #80]   ; c4 <proc_800213c+0xc4>
  70:   e1a03083        lsl     r3, r3, #1
  74:   e0813003        add     r3, r1, r3
  78:   e1d330b0        ldrh    r3, [r3]
  7c:   e1a03803        lsl     r3, r3, #16
  80:   e1a03843        asr     r3, r3, #16
  84:   e0233002        eor     r3, r3, r2
  88:   e1a03803        lsl     r3, r3, #16
  8c:   e1a03843        asr     r3, r3, #16
  90:   e14b30b6        strh    r3, [fp, #-6]
  94:   e51b300c        ldr     r3, [fp, #-12]
  98:   e2833001        add     r3, r3, #1
  9c:   e50b300c        str     r3, [fp, #-12]
  a0:   e51b200c        ldr     r2, [fp, #-12]
  a4:   e51b301c        ldr     r3, [fp, #-28]  ; 0xffffffe4
  a8:   e1520003        cmp     r2, r3
  ac:   bafffffd        blt     28 <proc_800213c+0x28>
  b0:   e15b30b6        ldrh    r3, [fp, #-6]
  b4:   e1a00003        mov     r0, r3
  b8:   e28bd000        add     sp, fp, #0
  bc:   e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
  c0:   e12fff1e        bx      lr
```

Defining the Problem

Maybe a decompiler will help? Maybe not a lot…


```
ushort proc_800213c(int param_1,int param_2)
{
  int local_10;
  ushort local_a;

  local_a = 0xffff;
  for (local_10 = 0; local_10 < param_2; local_10 =
local_10 + 1) {
    local_a = *(ushort *)
              (&table +
              (uint)(ushort)((ushort)*(byte
*)(param_1 + local_10) ^ local_a >> 8) * 2) ^
             (ushort)(((uint)local_a << 0x18) >>
0x10);
  }
  return local_a;
}
```

Towards a Solution…

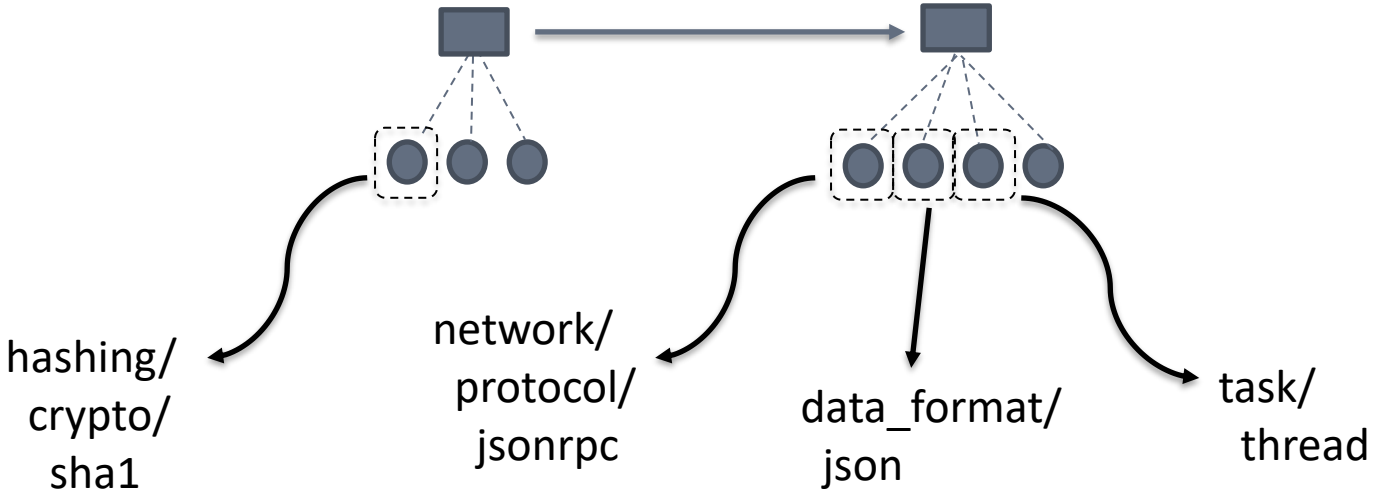
What if you had a tool that helped you understand what code does?

Capabilities Labeling

- Static analysis technique
- For binaries – no source code needed
- Works for “bare” binaries – no symbols
- Helps understand the purpose of code:
  - What code is part of the embedded web server?
  - Which functions access the encryption hardware?
  - Where can I inject simulated serial-port data?

What are Capability Labels?

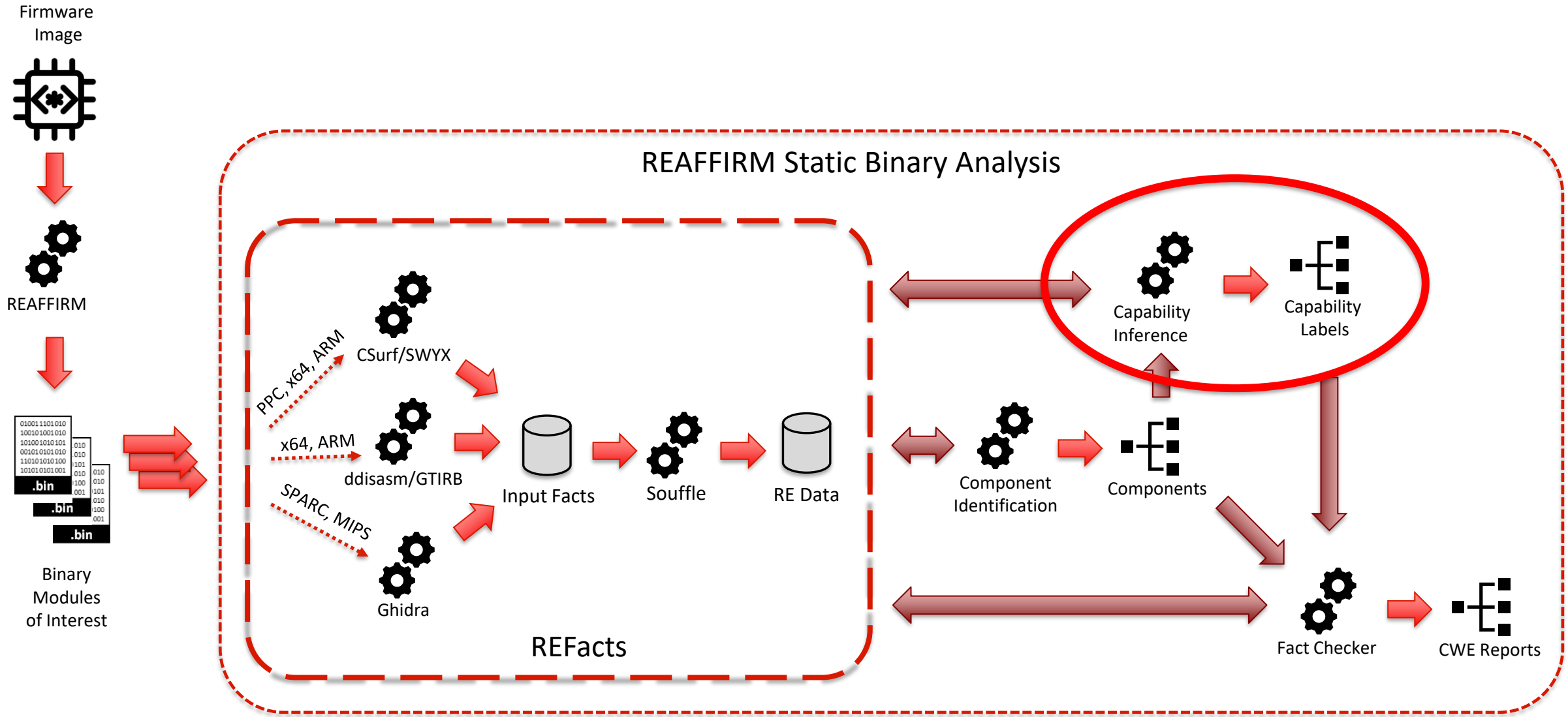
- Annotations that describe “purpose” of functions
  - Pre-defined collection of human-readable labels
  - Hierarchically organized (specific types within categories)
- May be combined with binary component analysis (e.g., compilation units)
  - Labels can then be aggregated over collections of related functions *(not shown below)*



REAFFIRM Intro

- REAFFIRM: Reverse Engineer, Analyze, and Fuzz FIRMware
  - Unpacks wide variety of firmware formats
  - Extracts, rehosts, and harnesses firmware functionality
  - Supports testing / fuzzing of firmware on commodity hardware
  - Also works on “easier” desktop/server binaries
- REAFFIRM is a toolbox
- Capabilities Labeling is one recently-added tool

REAFFIRM Toolchain Overview



Capabilities Hierarchy

- Represented as forest; most specific at the leaves
- Currently 187 types in 37 categories
- Examples:

```
audio/format:                Encodes or decodes audio formats
authentication/credentials:  Manages credentials for authentication
buffer/copy:                 Copies data between buffers/arrays
buffer/transform/base64:     Encodes/decodes data as base64
bus/serial:                  Generic serial data interface, e.g. UART
data_format/json:            Parses or emits data in JSON format
data_structure/ring_buffer:  Uses a ring buffer (circular queue)
encryption/aes:              Encrypts or decrypts data via the AES cipher
file/system/directory:       Accesses directories on the filesystem
hashing/checksum/adler32:    Computes Adler32 checksums
image/png:                   Encodes or decodes PNG images
network/protocol/http:       Communicates over HTTP
task/process/create:         Creates processes
```

Capability Labeling on ArduCopter

- For our examples, we are using “ArduCopter”
- Drone autopilot, open-source, C and C++
- Monolithic, typical of baremetal or real-time OS firmware
- Binary size: 3.6MB, 10760 functions
- 36 capability types in 16 categories identified in 14.8m
  - Static analysis: 9.0m (serves all later analyses)
  - Capabilities: 0.5m
  - Other analyses ~5.3m

Applications for Capabilities

- Helps focus analysis effort where it counts
  - Manual analysis:
    - *Highlight most relevant code*
  - Automated analysis
    - *Detect some weaknesses automatically*
    - *Improve scalability*

Capabilities Focus in Manual Analysis

- Expected functionality helps find targets for deeper dive
  - Locate buffer overrun risks
    - CWE-120 “buffer copy without checking size of input” (**`buffer/copy`**)
    - CWE-242 “inherently dangerous function” (**`bad_func`**)
  - Analyze code in a downed UAV
    - Check for GPS time spoofing weaknesses (**`time/lowlevel`**)
- Unexpected functionality may also warrant investigation
  - Understand why there is undocumented GPIO access (**`bus/gpio`**) inside the SD-Card driver code

Capabilities Focus in Automated Analysis

- Automatically identify weaknesses
  - E.g., CWE-327 “broken or risky cryptographic algorithm”
- Direct analysis to key security-related areas
  - Identify the (215) functions that relate to Arducopter’s “lua” parser
  - Fuzz to see if parsing failures can crash SW/device
- Level of effort can be scaled relative to importance
  - Allocate fuzzing resources (# of cores, # of days)
  - Safety critical systems ≫ user features
  - Exposed interfaces ≫ protected interfaces

Capabilities Heuristics

- How do we find capabilities?
- How much does this depend on symbols?
  - A little bit, but not as much as you might think
- Does this work on any processor type?
  - Very little that is ISA-dependent
  - Multiple frontends (including Ghidra) for flexibility

Multiple Sources of Signal for Labeling

- Library specifications (needs symbols)
  - Use of a library implies capability
    - e.g., libxml2 --> **`data_format/xml`**
  - Use of an API implies capability
    - e.g., execv() --> **`task/process/create`**

- Mathematical constants/tables (works on bare binaries)
  - Many cryptographic and checksum algorithms
  - Can usually distinguish between variants (**`encryption/des3`** vs **`encryption/aes`**)

- Hardware accesses (bare OK)
  - Map hardware addresses to capabilities
  - E.g., 0x40013800 is STM32F family UART status register (**`bus/serial`**)

Multiple Sources of Signal for Labeling

- Semantic patterns in instructions (bare OK)
  - Heuristics based on semantics or control/data flow
  - E.g., **`hashy`** functions have high percentage of XOR and shifts in loop(s)

- String literals/tokens (bare OK)
  - Look for file format or protocol specific strings/terms
  - Based on data mining from GitHub source code – machine learning

- Message matching (bare OK)
  - User and log messages *retained in binary* and contain valuable information
  - Based on human expertise – “expert system” like capability
  - E.g., "cannot open file '%s' (%s)“ -> **`file/system, error/file`**

Accuracy Assessment

- Validation by comparison to expert human analyst
  - But: analyst worked from *source code*; tool worked from *binary*
  - And: tool has improved since analyst did this work
- Human analyst reviewed 1046 functions:
    - 649 functions for which (older) tool identified at least one capability
    - 397 functions tool did not identify (random sample)
    - Represents only ~10% of functions in binary
  - Human review time: 985 min (pos), 865 min (neg) = 30.8h
- Automated analysis >1200x faster (on per-function basis)
- Analyst used more labels (76 vs 36), but less consistently

Accuracy Assessment

Statistic	Analyst Result	REAFFIRM Result
Functions analyzed	1046	10760
Processing time (minutes)	1850	14.8
Speed (functions/minute)	0.57	728
Functions initially labelled	518	649 (initial), 906 (latest)
Functions analyst could not find	125	n/a
# Unique capabilities	78	36
Most common labels	error, parsing, task, bus	error, parsing, time, processor

Accuracy Assessment

- Comparison of labels identified 67 human errors and omissions – cases where analyst initially failed
  - Some source code compile-time generated
  - Some functions too complex to readily understand
  - Compile-time optimizations couldn't be guessed
  - Or, "mental fatigue": analyst simply missed the obvious

Accuracy Assessment

- After analyst corrections:
  - Where both analyst and REAFFIRM applied a label, 71.6% category match
  - Approaches typical human-to-human inter-rater reliability of 75-80%
  - Where REAFFIRM applied labels and analyst did not, only 5 labels (<0.05%) marked “wrong” by analyst

Concluding Thoughts on Accuracy

- Difficult to establish absolute “ground truth”
  - Humans make mistakes and disagree on classifications
- Significant errors (false positives) from tool are rare and obscure
  - For example, 32-bit value 2.0f == 0x40000000; this happens to correspond to a target processor MMIO address for a hardware counter/timer.
- False negatives are more frequent, but less problematic
  - They won’t lead you astray; they will just fail to offer guidance
  - REAFFIRM is cautious, but FN rate (~22%) is being improved

Future Directions

- Several new areas of ongoing research
  - Structural information (wrappers, thunks)
  - Different ways to aggregate information above the function (“is-a” versus “has-a”)
  - Bounded symbolic execution for identification
    - Already demonstrated on math primitives: REMath
    - Could identify core library functions, e.g., strcpy

Questions