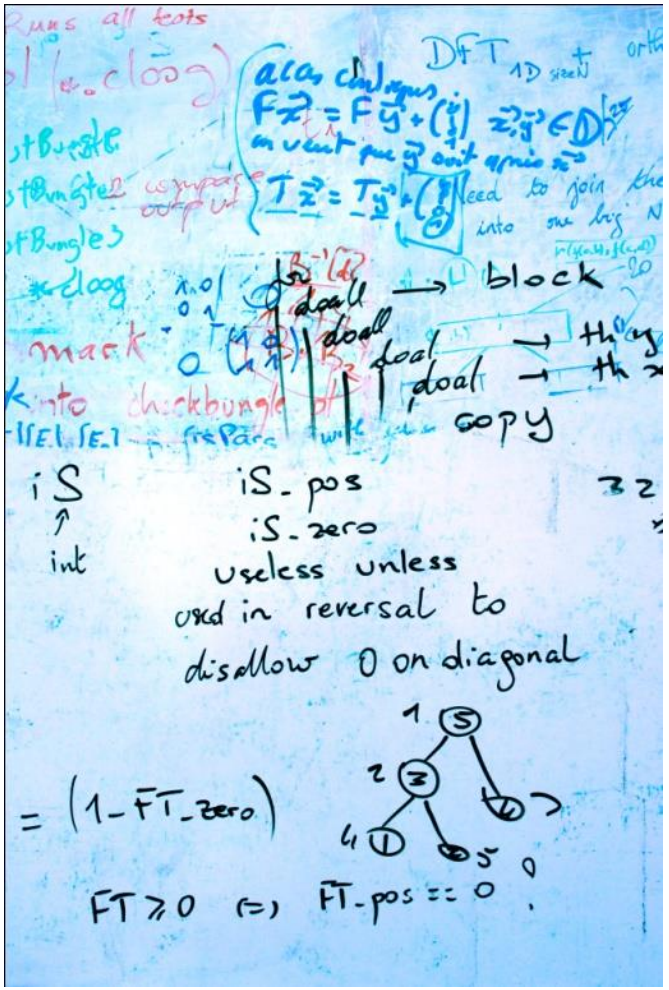
Blackspot: Using Tensor Decompositions to Guide Inspection of Source Code

David Bruns-Smith
James Ezick
Janice McMahon
Jonathan Springer

Reservoir Labs, Inc.
New York, NY

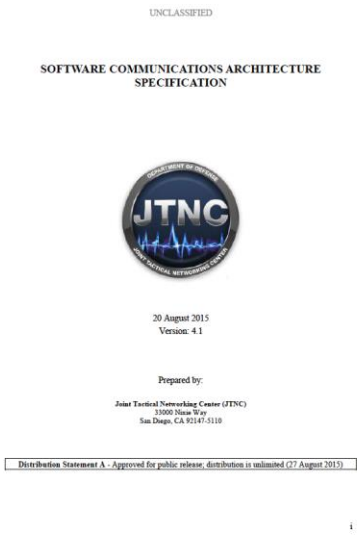


WInnComm '16
Wireless Innovation Forum
Conference on
Wireless Communications Technologies
and Software Defined Radio

16 March 2016

Holy Grail of Software Testing

The dream …


UNCLASSIFIED

SOFTWARE COMMUNICATIONS ARCHITECTURE SPECIFICATION

JTNC

20 August 2015
Version: 4.1

Prepared by:

Joint Tactical Networking Center (JTNC)
33000 Nixie Way
San Diego, CA 92147-5110

Distribution Statement A - Approved for public release; distribution is unlimited (27 August 2015)

Automatic Process

Plain Text Specification Document

Fast Automated Test
Pass: Safe! Compliant!
Fail: And here's why …

The reality …

Testing remains a challenging process that will continue to require and benefit from a combination of tools and processes

The Testing Gap

Static Analysis	Testing Gap	Dynamic Testing (Including Simulation)
Analyze the Code • Evaluate all possible execution paths • Formal methods can be difficult to scale • When precision of analysis is limited – *False Positives* – Hard to conclusively prove correctness	 Manual Source Code Inspection 	Execute the Software • Reassurance from "Test what you built" • Predetermined set of test cases or scenarios • When all execution paths cannot be covered – *False Negatives* – Hard to conclusively prove correctness

SCA 2.2.2 and 4.1 Compliance Requirements

SCA 2.2.2 and 4.1 Compliance Requirements

"The X operation shall Y if/when Z occurs."

SCA15: The initialize operation shall raise an InitializeError exception when an initialization error occurs.

SCA16: The releaseObject operation shall release all internal memory allocated by the component during the life of the component.

- Static Analysis
  - Can find *X*, can determine if *Y* and test for *Z* (sometimes) exists
  - Success depends on matching *Y* with *Z* – difficult (undecidable?)
- Dynamic Testing
  - Can run *X*, select a sample of *Z* scenarios, check for *Y*
  - Success depends on coverage of *Z* (usually assume $\neg Z \rightarrow \neg Y$)

C/C++ Code Weaknesses

MITRE CWE

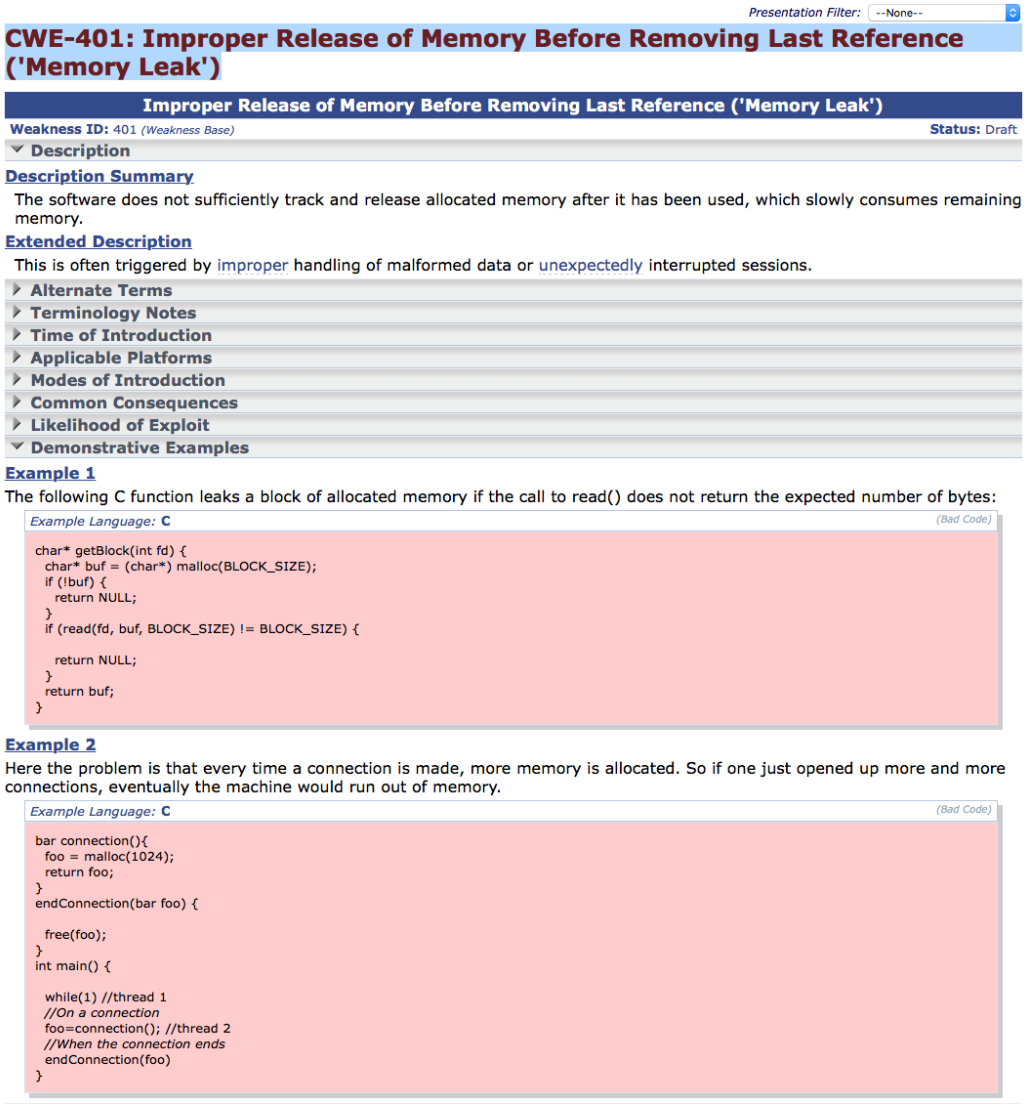
Common Weakness Enumeration

- Large Database
- Updated Frequently
- Includes Examples

Static Analysis

- Effort required to maintain specific tests
- Can be hard to capture cause in a single rule

- What if we could use latent factors derived from examples?



Presentation Filter: --None--

CWE-401: Improper Release of Memory Before Removing Last Reference ('Memory Leak')

Improper Release of Memory Before Removing Last Reference ('Memory Leak')

Weakness ID: 401 *(Weakness Base)* **Status:** Draft

Description

Description Summary

The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory.

Extended Description

This is often triggered by improper handling of malformed data or unexpectedly interrupted sessions.

Alternate Terms

Terminology Notes

Time of Introduction

Applicable Platforms

Modes of Introduction

Common Consequences

Likelihood of Exploit

Demonstrative Examples

Example 1

The following C function leaks a block of allocated memory if the call to read() does not return the expected number of bytes:

Example Language: **C** *(Bad Code)*

```
char* getBlock(int fd) {
  char* buf = (char*) malloc(BLOCK_SIZE);
  if (!buf) {
    return NULL;
  }
  if (read(fd, buf, BLOCK_SIZE) != BLOCK_SIZE) {

    return NULL;
  }
  return buf;
}
```

Example 2

Here the problem is that every time a connection is made, more memory is allocated. So if one just opened up more and more connections, eventually the machine would run out of memory.

Example Language: **C** *(Bad Code)*

```
bar connection(){
  foo = malloc(1024);
  return foo;
}
endConnection(bar foo) {

  free(foo);
}
int main() {

  while(1) //thread 1
  //On a connection
  foo=connection(); //thread 2
  //When the connection ends
  endConnection(foo)
}
```

https://cwe.mitre.org/data/definitions/401.html

Latent Semantic Analysis

Technical Memo Example

Titles:
c1: *Human* machine *interface* for Lab ABC *computer* applications
c2: A *survey* of *user* opinion of *computer system response time*
c3: The *EPS user interface* management *system*
c4: *System* and *human system* engineering testing of *EPS*
c5: Relation of *user*-perceived *response time* to error measurement

m1: The generation of random, binary, unordered *trees*
m2: The intersection *graph* of paths in *trees*
m3: *Graph minors* IV: Widths of *trees* and well-quasi-ordering
m4: *Graph minors*: A *survey*

Terms	Documents								
	c1	c2	c3	c4	c5	m1	m2	m3	m4
human	1	0	0	1	0	0	0	0	0
interface	1	0	1	0	0	0	0	0	0
computer	1	1	0	0	0	0	0	0	0
user	0	1	1	0	1	0	0	0	0
system	0	1	1	2	0	0	0	0	0
response	0	1	0	0	1	0	0	0	0
time	0	1	0	0	1	0	0	0	0
EPS	0	0	1	1	0	0	0	0	0
survey	0	1	0	0	0	0	0	0	1
trees	0	0	0	0	0	1	1	1	0
graph	0	0	0	0	0	0	1	1	1
minors	0	0	0	0	0	0	0	1	1

A sample dataset consisting of 9 technical memoranda. Terms are italicized. This dataset can be described by means of a term by document matrix.

S. Deerwester, S. T. Dumais and R. Harshman, "Indexing by Latent Semantic Analysis," Journal of the American Society for Information Science, Vol. 41, No. 6, 1990.

LSA is a from of Unsupervised Machine Learning

Unstructured Text

- Convert text into a word-occurrence matrix
- Compute SVD of the matrix
- Output allows clustering of terms and documents (topics)

Source Code

- Convert code into a function-structure/symbol occurrence matrix
- Compute SVD of the matrix
- Output allows clustering of functions and structures/symbols

Prior Work

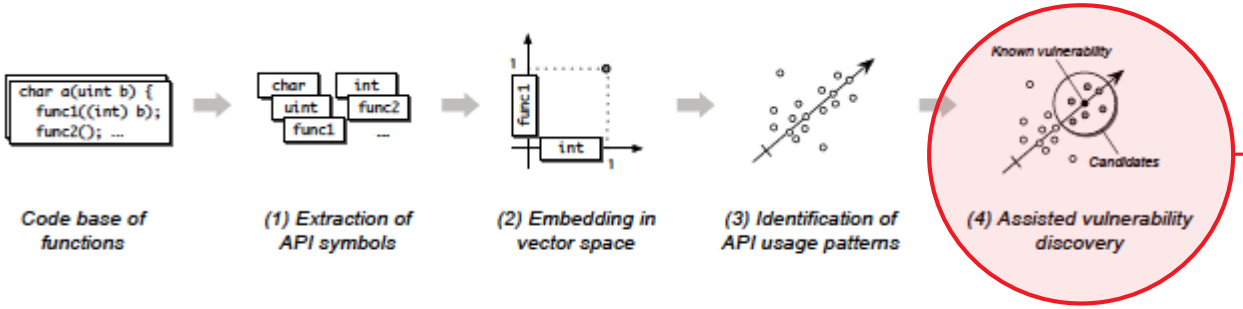


Figure 1: A schematic overview of our method for vulnerability extrapolation.

```
static int flic_decode_frame_8BPP(AVCodecContext *avctx,
                                  void *data, int *data_size,
                                  const uint8_t *buf, int buf_size)
{   [...] signed short line_packets; int y_ptr; [...]
    pixels = s->frame.data[0];
    pixel_limit = s->avctx->height * s->frame.linesize[0];
    frame_size = AV_RL32(&buf[stream_ptr]); [...]
    frame_size -= 16;
    /* iterate through the chunks */
    while ((frame_size > 0) && (num_chunks > 0)) { [...]
        chunk_type = AV_RL16(&buf[stream_ptr]);
        stream_ptr += 2;
        switch (chunk_type) { [...]
        case FLI_DELTA:
            y_ptr = 0;
            compressed_lines = AV_RL16(&buf[stream_ptr]);
            stream_ptr += 2;
            while (compressed_lines > 0) {
                line_packets = AV_RL16(&buf[stream_ptr]);
                stream_ptr += 2;
                if ((line_packets & 0xC000) == 0xC000) {
                    // line skip opcode
                    line_packets = -line_packets;
                    y_ptr += line_packets * s->frame.linesize[0];
                } else if ((line_packets & 0xC000) == 0x4000) {
                    [...]
                } else if ((line_packets & 0xC000) == 0x8000) {
                    // "last byte" opcode
                    pixels[y_ptr + s->frame.linesize[0] - 1] =
                        line_packets & 0xff;
                } else { [...]
                    y_ptr += s->frame.linesize[0];
                }
            }
            break; [...]
        } [...]
    } [...]
    return buf_size;
}
```

Figure 4: Original vulnerability (CVE-2010-3429).

Similarity	Function name
1.00	flic_decode_frame_8BPP
0.96	flic_decode_frame_15_16BPP
0.83	lz_unpack
0.80	decode_frame (lcldec.c)
0.80	raw_encode
0.76	vmdvideo_decode_init
0.72	vmd_decode
0.70	aasc_decode_frame
0.68	flic_decode_init
0.67	decode_format80
0.66	targa_decode_rle
0.66	adpcm_decode_init
0.66	decode_frame (zmbv.c)
0.66	decode_frame (8bps.c)
0.65	msrle_decode_8_16_24_32
0.65	wmavoice_decode_init
0.65	get_quant
0.64	MP3lame_encode_frame
0.64	mpegts_write_section
0.64	tgv_decode_frame

Table 1: Top 20 of 6,778 functions ranked by cosine similarity to flic_decode_frame_8BPP. Discovered vulnerabilities are indicated by a shaded background.

F. Yamaguchi, F. Lindner and K. Rieck, "Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning," WOOT'11, August 2011.

Later work extended this method to incorporate Abstract Syntax Tree (AST) fragments

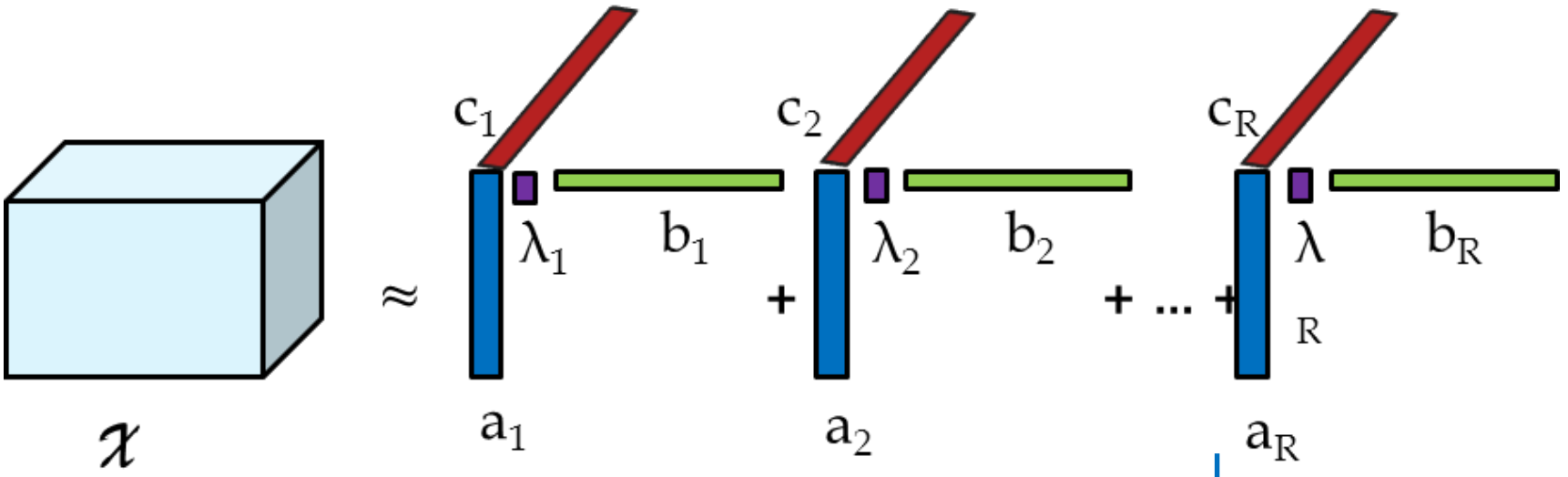
Method uses a form of Principal Component Analysis (PCA) –

Cannot separate structure from symbols

Any significant separation in any portion of the pattern would put the points "outside the circle" and would not be detected by similarity

What is needed is an approach that detects cases where different symbols are used within similar structure and vice versa

Tensor Decompositions

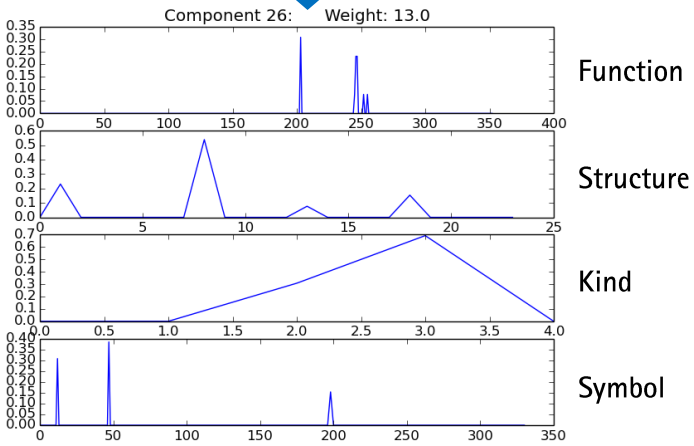


CP Tensor Decomposition

Tensor is decomposed into a non-unique weighted sum of a pre-defined number of rank-1 components

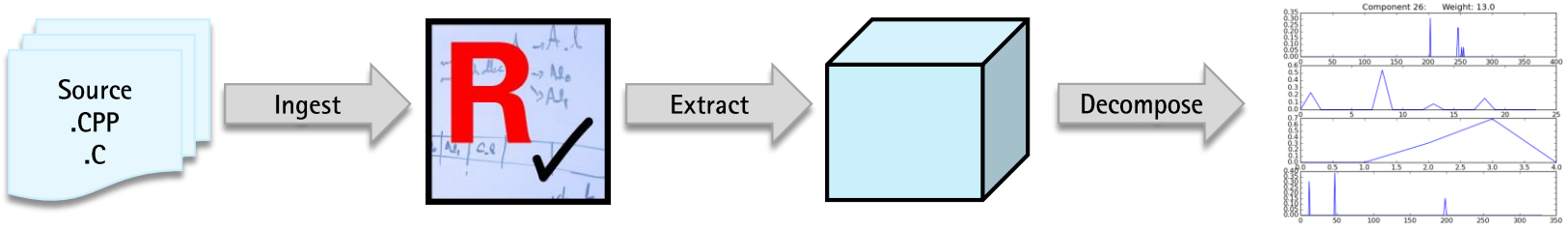
T. Kolda and B. Bader, "Tensor Decompositions and Applications," SIAM Review, Vol 51, No. 3, 2009.

Tensor Component
A tensor is a multidimensional array. Tensor decompositions generalize the SVD using techniques from multilinear algebra.



FM3TR Source Code
CP-APR, R = 100 components

Blackspot Workflow



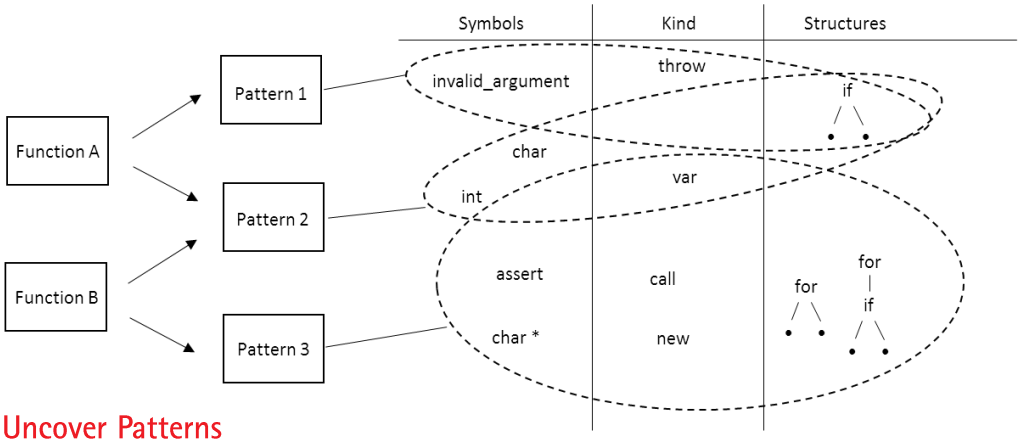
C/C++
Source Code for Analysis "as is"

R-Check SCA
Uses compiler-grade whole project parsing combined with Pitchfork rules to extract markers from code

Pitchfork
Metalanguage that combines syntactic patterns with actions – used to identify AST structures in code

Tensor
Function x
Structure x
Symbol x
Kind

Analyze



Uncover Patterns
Cluster functions based on similarity of pattern occurrence
Logically organize code for manual inspection
Identify functions that share patterns with known defects
Find "synonyms" of known defects for inspection

Blackspot Tensor Analysis Experiments

FM3TR Code

- ~100,000 lines of C++
- 353 functions,
  331 symbols,
  5 kinds,
  24 structures
- 1,515 non-zero entries

Reservoir ENSIGN

- Optimized, parallelized decompositions for sparse tensors
- ~1.1 seconds

```
CORBA::Object_ptr
AudioCapture::getPort(const
CORBA::Char* name)
  throw (CORBA::SystemException,
     CF::PortSupplier::UnknownPort)
{
  std::string temp(name);

  if (temp != "AudioCaptureOut") {
    throw
CF::PortSupplier::UnknownPort();
  }

  return m_pcmout->getPort();
}
```

AudioCapture::getPort

Pattern	Uniqueness Score
39	0.2222
57	0.0502
93	0.0172

Pattern 39

Structure	Score
IF()	1.0

Symbols	Score
CF::PortSupplier::UnknownPort	0.5
operator!=	0.5

Kind	Score
CALL	1.0

Pattern 57

Structure	Score
IF()	1.0

Symbols	Score
tk_error	1.0

Kind	Score
THROW	1.0

Pattern 93

Structure	Score
*	1.0

Symbols	Score
tk_class	0.98
tk_typeref tk_class	0.02

Kind	Score
VAR	1.0

These tables show the results of the tensor decomposition for the AudioCapture::getPort() function. The table below the code shows the breakdown of the function into patterns.

The Uniqueness Score is a value between 0 and 1 indicating how unique the pattern is to this particular function.

The tables to the right show three of these patterns and their composition in terms of structure, symbol, and symbol kind. The score indicates what fraction of the total symbols, structures, or kinds within the pattern that particular index represents. Note that "tk_error" and "tk_class" are Pitchfork type symbols that refer to the type of "UnknownPort" and "std::string," respectively.

Blackspot for Code Inspection

Pattern 10. Uniqueness = 0.05

Structure	Score
IF(IF())	0.79
IF()	0.21

Symbols	Score
tk_error	1.0

Kind	Score
THROW	0.9
NEW	0.1

Pattern 11. Uniqueness = 0.11

Structure	Score
IF()	0.88
IF(IF())	0.12

Symbols	Score
char *	1.0

Kind	Score
THROW	1.0

Pattern 37. Uniqueness = 1.0

Structure	Score
IF(LOOP()LOOP()IF())	1.0

Symbols	Score
tk_class	0.4
char	0.2
tk_class *	0.2
int	0.2

Kind	Score
VAR	0.8
THROW	0.2

Example: [SCA 2.2.2] AP0090: The getPort operation shall raise an UnknownPort exception if the port name is invalid.

Uniqueness Score captures the number of functions in which a particular code pattern occurs (# functions = 1/uniqueness score)

From the breakdown of patterns –
Code inspection tasks can be organized in a logical way –
Sort relevant functions by pattern – or – sort by pattern across requirements

Blackspot for Weakness Discovery

MskDemodulator::SymbolSynch

Pattern	Uniqueness Score
6	0.584
87	0.487
84	0.286
74	0.222 Kind: NEW
67	0.021
16	0.016

Has similar memory leak to SymbolSynch

MskDemodulator::SymbolSynch

Function	Cosine Similarity	Shared Patterns
MskDemodulator::Run	0.482	6, 67
FileOutput_MAC_LLC::RewritePacket	0.321	84
ShortInPort_impl::ShortInPort_impl	0.263	74
MskModulator::Modulate	0.263	87, 67, 16
CVSDDecoder::CVSDDecoder	0.217	74
RsBlockDecoder::RsBlockDecoder	0.216	74, 67
FileInput_MAC_LLC::Init	0.129	74
FileOutput_MAC_LLC::Run	0.121	84
MskDemodulator::Init	0.116	74, 67

SymbolSynch function in which Cppcheck discovered a memory leak

RHS: Shows the patterns that make up SymbolSynch
LHS: Shows those functions with the top cosine similarities to SymbolSynch

By focusing on functions with shared patterns, less inspection in necessary to identify similar weaknesses

In this case, Pattern 74 is the only pattern with NEW kind –
Only two functions with Pattern 74 were not constructors –

One had a memory leak that was not found by Cppcheck or SVD approach!

Future Work

Technique is applicable to general C/C++ code

- With ENSIGN, scales to millions of lines of code
- Cyber-physical systems

Expand the base of markers

- Tensors not limited to 4 dimensions
- Can add derivative factors
  (E.g., cyclomatic complexity)

Validate

- Does this accelerate code inspection?
- Automation?
- Provides a new way of thinking about compliance testing workflow

Thank You

R-Check SCA development supported by SPAWAR under
Navy Phase II.5 SBIR Contract
"Static Analysis Tool for Interface Compliance Verification and Program Comprehension"

For more information on R-Check SCA

- https://www.reservoir.com/rchecksca

Contact us by email

- rcheck-support@reservoir.com