



## Finding the Serious Bugs that Matter with Advanced Static Analysis

Static Analysis Days @ Verifysoft, May 2021

Paul Anderson, VP of Engineering, GrammaTech, Inc.

1

### Key Takeaway



## ***Real Functional Safety is More Important than Conformance***

***Do use a coding standard and check for violations,***

***but....***

***Don't assume that conformance guarantees safety***

2

## Overview



- Static Analysis – superficial vs. deep
- The risks of C and C++
- Techniques needed for advanced static analysis
- Examples
- Customization

3

## Introduction to Static Analysis



- Infers information about software behavior based on an abstract model of the code
  - In contrast to dynamic analysis, such as profiling, debugging, testing
- Analyzes code instead of executing it
  - So no test cases are needed
- Is usually a two-phase process
  - Extract semantic information from source code
  - Use information to discover defects or other properties of interest
- This talk is mostly about **Advanced** static analysis for **Bug Finding**
  - As exemplified by **CodeSonar**®

4

## Superficial vs. Deep



- Syntactic rules

- Mostly surface-level properties
- Most are decidable
- Many are about improving readability
- Violations generally easy to find
- *Low correlation with serious defects*
  - i.e., unlikely to cause crashing bugs

Checkable by early-generation tools such as lint, or by some modern compilers

Many discrete Misra rules cover many instances

Only 9/72 Misra rules were observed to be better than random at predicting defects

C. Boogerd and L. Moonen. Assessing the Value of Coding Standards: An Empirical Study. In Proceedings of the 24th International Conference on Software Maintenance (ICSM), pages 277–286. IEEE Computer Society Press, 2008.

- Semantic rules

- Mostly about run-time properties
- Usually undecidable
- Powerful analyses are necessary for detection
- *High correlation with serious defects*
  - E.g, leaks, buffer overruns, null pointer exceptions, use after free, uninitialized variables, etc.

Covered indirectly by a small number of Misra rules. E.g., Misra C 2012 Rule 1.3: There shall be no occurrence of undefined or critical unspecified behavior

5

## Overview



- Static Analysis – superficial vs. deep
- **The risks of C and C++**
- Techniques needed for advanced static analysis
- Examples
- Customization

6

## The Two Most Important Rules in Misra C



**Rule 1.3** There shall be no occurrence of undefined or critical unspecified behaviour

**Category** Required  
**Analysis** Undecidable, System  
**Applies to** C90, C99

### Amplification

Some undefined and unspecified behaviours are dealt with by specific rules. This rule prevents all other undefined and critical unspecified behaviours. Appendix H lists the undefined behaviours and those unspecified behaviours that are considered critical.

**Dir 4.1** Run-time failures shall be minimized

**Category** Required  
**Applies to** C90, C99

C90 [Undefined 15, 19, 26, 30, 31, 32, 94]  
 C99 [Undefined 15, 16, 33, 40, 43-45, 48, 49, 113]

### Rationale

The C language was designed to provide very limited built-in run-time checking. While this approach allows generation of compact and fast executable code, it places the burden of run-time checking on the programmer. In order to achieve the desired level of robustness, it is therefore important that programmers carefully consider adding dynamic checks wherever there is potential for run-time errors to occur.

7

## Undefined and Critical Unspecified Behavior



- **Undefined Behavior**
  - E.g.: “The program attempts to modify a string literal.”
  - 230 instances in C90/99
  - 65 not covered by any other MISRA Rule
- **Critical Unspecified Behavior**
  - What does `malloc(0)` return?
  - 51 instances
  - 17 not covered by any other MISRA Rule
- **C99 standard:**
  - 2½ pages of Unspecified behavior
  - 13 pages of Undefined behavior
  - 6½ pages of Implementation-defined behavior

8

## Risks of Undefined Behavior



- The Achilles Heel of C programs
- => anything goes, including “Catch fire”!
- Not a rarely-encountered niche
- Source of most serious bugs
  - Buffer overruns
  - Invalid pointer indirection
  - Use after free
  - Double free
  - Data races
  - Division by zero
  - Use of uninitialized memory
  - Etc....



9

## Overview



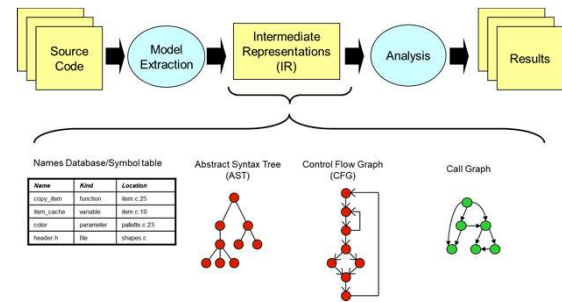
- Static Analysis – superficial vs. deep
- The risks of C and C++
- **Techniques needed for advanced static analysis**
- Examples
- Customization

10

## Advanced Static Analysis Tools



- Tools whose **primary** purpose is to find serious bugs
  - Mostly undefined behavior
- Understand **semantics**, not just syntax
- Based on **abstract interpretation**
  - Using techniques pioneered in high-assurance hardware design
- **API aware**
  - With knowledge of how library functions respond in anomalous circumstances

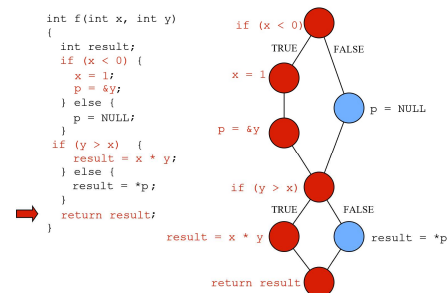


11

## Properties of Advanced Static Analysis Tools



- Precise whole program model
  - Derived from parsing the code just as the compiler would
- Flow-sensitive
  - Analysis understands order of execution
- Interprocedural
  - For tracking data and control flow between procedures
- Context-sensitive
  - Analysis understands that different call sites have different properties
- Whole-program analysis
  - To analyze effects of data and control flow across file boundaries
- Path-sensitive
  - Analysis can compute properties of distinct paths
  - Infeasible paths are eliminated
  - Results shown in terms of paths through the code
- Concurrency sensitive
  - Aware of threads and locking
- Learning/statistical analysis
  - To find deviations from “normal”



12

## Overview



- Static Analysis – superficial vs. deep
- The risks of C and C++
- Techniques needed for advanced static analysis
- **Examples**
- Customization

13

## Examples



- All were found in production code
- All are unlikely to be flagged as violations of coding standards

14

## Example: Abstract Interpretation



```

2186 char *return_append_str(char *dest, const char *s) {
2187     /* Append text s to dest, and return new result. */
2188     char *newloc;
2189     size_t newlen;
2190     /* This doesn't have buffer overflow vulnerabilities, because
2191        we always allocate for enough space before appending. */
2192     if (!dest) {
2193         newloc = (char *) malloc(strlen(s)+1);
2194         strcpy(newloc, s);
2195         return newloc;
2196     }
2197     newlen = strlen(dest) + strlen(s) + 1;
2198     newloc = (char *) malloc(newlen);
2199     strcpy(newloc, dest);
2200     if (!newloc) return dest; /* Can't do it, throw away the data */
2201     strcat(newloc, s);
2202     return newloc;
2203 }

```

15

## Example: Abstract Interpretation



```

2186 char *return_append_str(char *dest, const char *s) {
2187     /* Append text s to dest, and return new result. */
2188     char *newloc;
2189     size_t newlen;
2190     /* This doesn't have buffer overflow vulnerabilities, because
2191        we always allocate for enough space before appending. */
2192     if (!dest) {
2193         newloc = (char *) malloc(strlen(s)+1);
2194         strcpy(newloc, s);

```

**Event 2:** s is passed to strlen().

- This determines the capacity of the buffer that will be overrun later.

hide

**Event 5:** malloc() returns the address of a new object.

- This points to the buffer that will be overrun later.

hide

**Event 8:** s is passed to strcpy() as the second argument.

- This determines the position accessed during the buffer overrun later.

hide

**Buffer Overrun**

This code writes past the end of the buffer pointed to by newloc.

- newloc evaluates to malloc(strlen(s)) + 1 `!expn.c:2193`.
- strcpy() writes to the byte at an offset that is the length of the string pointed to by s, plus 1 from the beginning of the buffer pointed to by newloc.
  - The offset exceeds the capacity.
  - The length of the string pointed to by s, plus 1 is no less than 1. See related event [8](#).
  - The capacity of the buffer pointed to by newloc, in bytes, is the length of the string pointed to by s, which is bounded below by 0. See related events [6](#) and [9](#).
- The overrun occurs in heap memory.

The issue can occur if the `highlighted` code executes.

See related events [6](#), [8](#), and [9](#).

Show: All events | Only primary events

16



## Example: Copy-Paste Error



```

118 void
119 more_variables ()
120 {
121     int indx;
122     int old_count;
123     bc_var **old_var;
124     char **old_names;
125
126     /* Save the old values. */
127     old_count = v_count;
128     old_var = variables;
129     old_names = v_names;
130
131     /* Increment by a fixed amount and allocate. */
132     v_count += STORE_INCR;
133     variables = (bc_var **) bc_malloc (v_count*sizeof(bc_var *));
134     v_names = (char **) bc_malloc (v_count*sizeof(char *));
135
136     /* Copy the old variables. */
137     for (indx = 3; indx < old_count; indx++)
138         variables[indx] = old_var[indx];
139
140     /* Initialize the new elements. */
141     for (; indx < v_count; indx++)
142         variables[indx] = NULL;
143
144     /* Free the old elements. */
145     if (old_count != 0)
146     {
147         free (old_var);
148         free (old_names);
149     }
150 }

```

```

152 void
153 more_arrays ()
154 {
155     int indx;
156     int old_count;
157     bc_var_array **old_ary;
158     char **old_names;
159
160     /* Save the old values. */
161     old_count = a_count;
162     old_ary = arrays;
163     old_names = a_names;
164
165     /* Increment by a fixed amount and allocate. */
166     a_count += STORE_INCR;
167     arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var_array *));
168     a_names = (char **) bc_malloc (a_count*sizeof(char *));
169
170     /* Copy the old arrays. */
171     for (indx = 1; indx < old_count; indx++)
172         arrays[indx] = old_ary[indx];
173
174     /* Initialize the new elements. */
175     for (; indx < v_count; indx++)
176         arrays[indx] = NULL;
177
178     /* Free the old elements. */
179     if (old_count != 0)
180     {
181         free (old_ary);
182         free (old_names);
183     }
184 }
185
186

```

www.grammatech.com

17

© GrammaTech, Inc. All rights reserved.

17

## Example: Copy-Paste Error



```

118 void
119 more_variables ()
120 {
121     int indx;
122     int old_count;
123     bc_var **old_var;
124     char **old_names;
125
126     /* Save the old values. */
127     old_count = v_count;
128     old_var = variables;
129     old_names = v_names;
130
131     /* Increment by a fixed amount and allocate. */
132     v_count += STORE_INCR;
133     variables = (bc_var **) bc_malloc (v_count*sizeof(bc_var *));
134     v_names = (char **) bc_malloc (v_count*sizeof(char *));
135
136     /* Copy the old variables. */
137     for (indx = 3; indx < old_count; indx++)
138         variables[indx] = old_var[indx];
139
140     /* Initialize the new elements. */
141     for (; indx < v_count; indx++)
142         variables[indx] = NULL;
143
144     /* Free the old elements. */
145     if (old_count != 0)
146     {
147         free (old_var);
148         free (old_names);
149     }
150 }
151

```

```

152 void
153 more_arrays ()
154 {
155     int indx;
156     int old_count;
157     bc_var_array **old_ary;
158     char **old_names;
159
160     /* Save the old values. */
161     old_count = a_count;
162     old_ary = arrays;
163     old_names = a_names;
164
165     /* Increment by a fixed amount and allocate. */
166     a_count += STORE_INCR;
167     arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var_array *));
168     a_names = (char **) bc_malloc (a_count*sizeof(char *));
169
170     /* Copy the old arrays. */
171     for (indx = 1; indx < old_count; indx++)
172         arrays[indx] = old_ary[indx];
173
174     /* Initialize the new elements. */
175     for (; indx < v_count; indx++)
176         arrays[indx] = NULL;
177
178     /* Free the old elements. */
179     if (old_count != 0)
180     {
181         free (old_ary);
182         free (old_names);
183     }
184 }
185

```

**Copy-Paste Error**

This block of text appears to be a modified copy of the highlighted text. Did you intend to consistently change `v_count` to `a_count`, including here?

www.grammatech.com

18

18

# Example: Concurrency Analysis



Data Race at input.c:123 No properties have been set. Jump to warning location ↓ edit properties warning details...

```

thread 1
EndSearch() /rid0/paul/Examples/gnuchess-5.07/src/util.c
147 void EndSearch (int sig __attribute__((unused)) )
148
149
150 * User has pressed Ctrl-C. Just set flags TIMEOUT to be true.
151 *
152 *
153 *
154 {
155     SET (flags, TIMEOUT);
156     #define SET(a,b) \
157     do { \
158     (a) |= (b); \
159     dbg_printf("Set 0x%x\n", (b)); \
160     } while (0)

```

**Data Race**  
This code writes to global variable flags.

- The other thread reads from flags. See other access.
- No locks are currently held so a race with the other thread may occur.
- Compilers and processors reorder accesses to shared variables, so even source code that looks safe can be vulnerable to data races.

The issue can occur if the highlighted code executes.

Show: All events | Only primary events

```

thread 2
input_func() /rid0/paul/Examples/gnuchess-5.07/src/input.c
119 void *input_func(void *arg __attribute__((unused)) )
120 {
121     char prompt[MAXSTR] = "";
122
123     while (!(flags & QUIT)) {
124         #define QUIT 0x0001

```

**Data Race**  
This code reads from global variable flags.

- The other thread writes to flags. See other access.
- No locks are currently held so a race with the other thread may occur.
- Compilers and processors reorder accesses to shared variables, so even source code that looks safe can be vulnerable to data races.

The issue can occur if the highlighted code executes.

Show: All events | Only primary events

19

# Example: Deviant Behavior Detection



Ignored Return Value at gd-utils.c:48 jranien2, P2: Low, Fixed, Memory leak, "Fixed upstream..." edit properties warning details...

```

create_thumbnail() /root/base/trunk/cso-tests/regression/srpm/gnome-documents-x86_64/temp/fpmbuild/BUILD/gnome-documents-3.10.3/src/lib/gd-utils.c
36 create_thumbnail (GIOSchedulerJob *job,
37                  Gancellable *cancellable,
38                  gpointer user_data)
39 {
40     GSimpleAsyncResult *result = user_data;
41     GFile *file = G_FILE (g_async_result_get_source_object (G_ASYNC_RESULT (result)));
42     GnomeDesktopThumbnailFactory *factory;
43     GFileInfo *info;
44     gchar *uri;
45     GdkPixbuf *pixbuf;
46     guint64 mtime;
47
48     uri = g_file_get_uri (file);

```

**Ignored Return Value**  
The return value of g\_file\_get\_uri() is not checked or used at all.

- If the return value can indicate an error, the error will be ignored if the highlighted code executes.
- The return value of g\_file\_get\_uri() is checked or used more than 99% of the time across a large number of open source packages. CodeSonar infers from this the need to enforce Ignored Return Value checks on all calls to this function, regardless of the pattern of usage in this project.
- (To exempt g\_file\_get\_uri() from the Ignored Return Value check, use configuration file parameter RETURN\_CHECKER\_IGNORED\_FUNCS).

Show: All events | Only primary events

```

49     info = g_file_query_info (file, ATTRIBUTES_FOR_THUMBNAIL,
50                             G_FILE_QUERY_INFO_NONE,
51                             NULL, NULL);
52
53     /* We don't care about reporting errors here, just fail the
54     * thumbnail.
55     */
56     if (info == NULL)
57     {
58         g_simple_async_result_set_op_res_gboolean (result, FALSE);
59         goto out;
60     }
61
62     mtime = g_file_info_get_attribute_uint64 (info, G_FILE_ATTRIBUTE_TIME_MODIFIED);
63
64     factory = gnome_desktop_thumbnail_factory_new (GNOME_DESKTOP_THUMBNAIL_SIZE_LARGE);
65     pixbuf = gnome_desktop_thumbnail_factory_generate_thumbnail (factory,
66                                                                uri, g_file_info_get_content_type (info));
67
68     if (pixbuf != NULL)
69     {
70         gnome_desktop_thumbnail_factory_save_thumbnail (factory, pixbuf,

```

Return Value is assigned to variable uri

Unusual condition forces early exit

The value of uri is used, but not on the path shown!

© GammaTech, Inc. All rights reserved.

20

## Example: Taint Analysis (aka Hazardous Information Flow)



Command Injection at s\_bsd.c:1436 No properties have been set. | edit properties  
Jump to warning location | warning details...

Show Events | Options

```
read_packet() /rid0/paul/Examples/Unreal3.2/src/s_bsd.c
1408 static int read_packet(aClient *cptr, fd_set *rfd)
1409 {
1410     int dolen = 0, length = 0, done;
1411     time_t now = TStime();
1412     if (FD_ISSET(cptr->fd, rfd) &&
1413         !(IsPerson(cptr) && DBufLength(&cptr->recvQ) > 6090))
1414     {
1415         Hook *h;
1416         SET_ERRNO(0);
1417     #ifdef USE_SSL
1418         if (cptr->flags & FLAGS_SSL)
1419             length = ircd_SSL_read(cptr, readbuf, sizeof(readbuf));
1420         else
1421             length = recv(cptr->fd, readbuf, sizeof(readbuf), 0);
1422     #endif
1423     #ifdef USE_SSL
1424     #endif
1425     #endif
1426     cptr->lasttime = now;
1427     if (cptr->lasttime > cptr->since)
1428         cptr->since = cptr->lasttime;
1429     cptr->flags &= ~(FLAGS_PINGPMT | FLAGS_NONL);
1430     /*
1431      * If not ready, fake it so it isnt closed
1432      */
1433     if (length < 0 && errno == EWOULDBLOCK)
1434         return 1;
1435     if (length == 0)
1436         return length;
1437     #ifdef DEBUGMODE3
1438     if (!memcmp(readbuf, DEBUGMODE3_INFO, 2))
1439         DEBUG3_LOG(readbuf);
1440     #endif
1441     Command Injection
1442     A tainted string specifies a process or command line to be executed.
1443     * readbuf evaluates to *readbuf_s_bsd.c:1422
1444     * readbuf is tainted by network data.
1445     The issue can occur if the highlighted code executes.
1446     See related events 5 and 10
1447     Show: All events | Only primary events
```

```
/rid0/paul/Examples/Unreal3.2/include/struct.h
520 #define DEBUG3_LOG(x) DEBUG3_DOLOG_SYSTEM (x)

/rid0/paul/Examples/Unreal3.2/include/struct.h
1382 #define DEBUG3_DOLOG_SYSTEM(x) system(x)

/rid0/paul/Examples/Unreal3.2/include/struct.h
519 #define DEBUGMODE3_INFO "AB"

> AB; rm -rf /
```

**GAME OVER!**

21 © GammaTech, Inc. All rights reserved.

21

## Overview



- Static Analysis – superficial vs. deep
- The risks of C and C++
- Techniques needed for advanced static analysis
- Examples
- Customization**

22

## Why Customize?



- Custom APIs
  - Adapt built-in functionality for your own purposes
- Corporate Coding Standards
  - Naming conventions
  - Forbidden constructs
- Domain-specific Rules
  - Temporal Properties
  - Program Semantics

23

## Customization Mechanisms



- Configuration changes
  - Best for extending scope of existing checkers.
  - E.g., **extending leak checking to domain-specific resources**
- API Modeling
  - Write code to educate the analysis about key properties and constraints of the API
  - Best for finding violations of rules for using APIs
  - E.g., **find where preconditions are not satisfied**
- Program Model
  - Access to internal structures such as Abstract Syntax Trees, Control-flow Graphs, Call Graph, Symbol Tables
  - Best for surface-level properties
  - E.g., **violation of naming conventions**
- Analysis Visitors
  - Callbacks invoked at key points during the core analysis
  - Best for semantics-sensitive properties
  - E.g., **find where values of variables are in an inappropriate range**

24

## Conclusions



- Narrow focus on conformance with coding standards may blind you to what is really important
- Use Advanced Static Analysis to help find the most serious software defects

Key Takeaway

*Real Functional Safety is More Important than Conformance*

*Do use a coding standard and check for violations,*

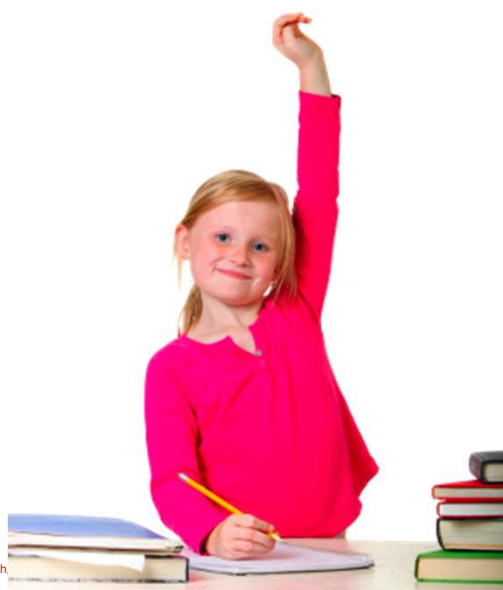
*but....*

*Don't assume that conformance guarantees safety*

www.grammatech.com 2 © GrammaTech, Inc. All rights reserved.

25

## Questions?



- My contact info:
  - Paul Anderson
  - [paul@grammatech.com](mailto:paul@grammatech.com)
  - <http://www.grammatech.com>

26