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

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***®

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
	C90 [Undefined 15, 19, 26, 30, 31, 32, 94] C99 [Undefined 15, 16, 33, 40, 43–45, 48, 49, 113]
Category	Required
Applies to	C90, C99

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.

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

Risks of Undefined Behavior

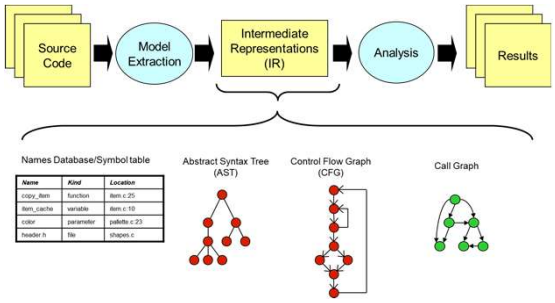
- The Achilles Heel of C programs
- => anything goes, including “Catch fire”!
- Not a rarely-encountered niche
- Source of most serious bugs
  - Buffer overruns
  - Invalid pointer indirection
  - Use after free
  - Double free
  - Data races
  - Division by zero
  - Use of uninitialized memory
  - Etc….

Overview

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

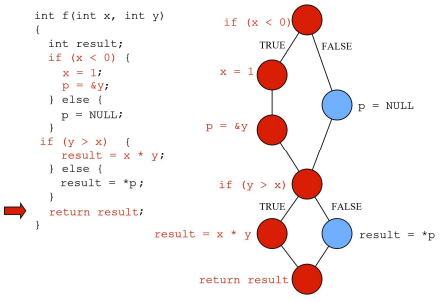
Advanced Static Analysis Tools

- 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



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”



Overview

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

Examples

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

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 }
```


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;
```

⚠ 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

```
⚠ 2194                 strcpy(newloc, s);
```

⚠ 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`[lexpgn.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

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 }
```

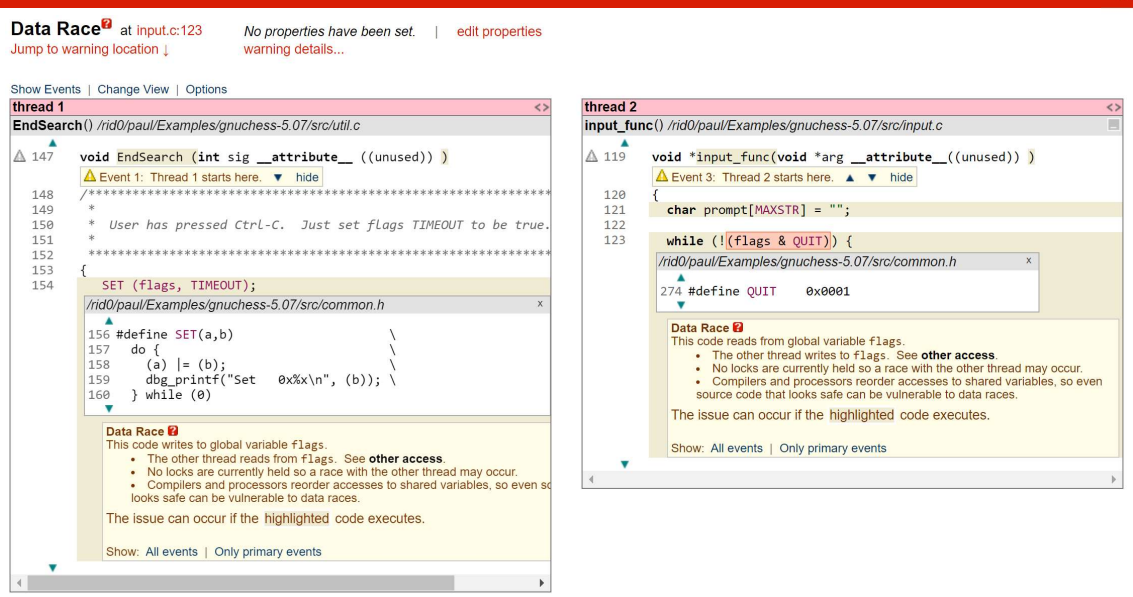
```
152 void
153 more_arrays ()
154 {
155   int indx;
156   int old_count;
157   bc_var_array **old_ary;
158   char **old_names;
159
160   /* Save the old values. */
161   old_count = a_count;
162   old_ary = arrays;
163   old_names = a_names;
164
165   /* Increment by a fixed amount and allocate. */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


Example: Copy-Paste Error

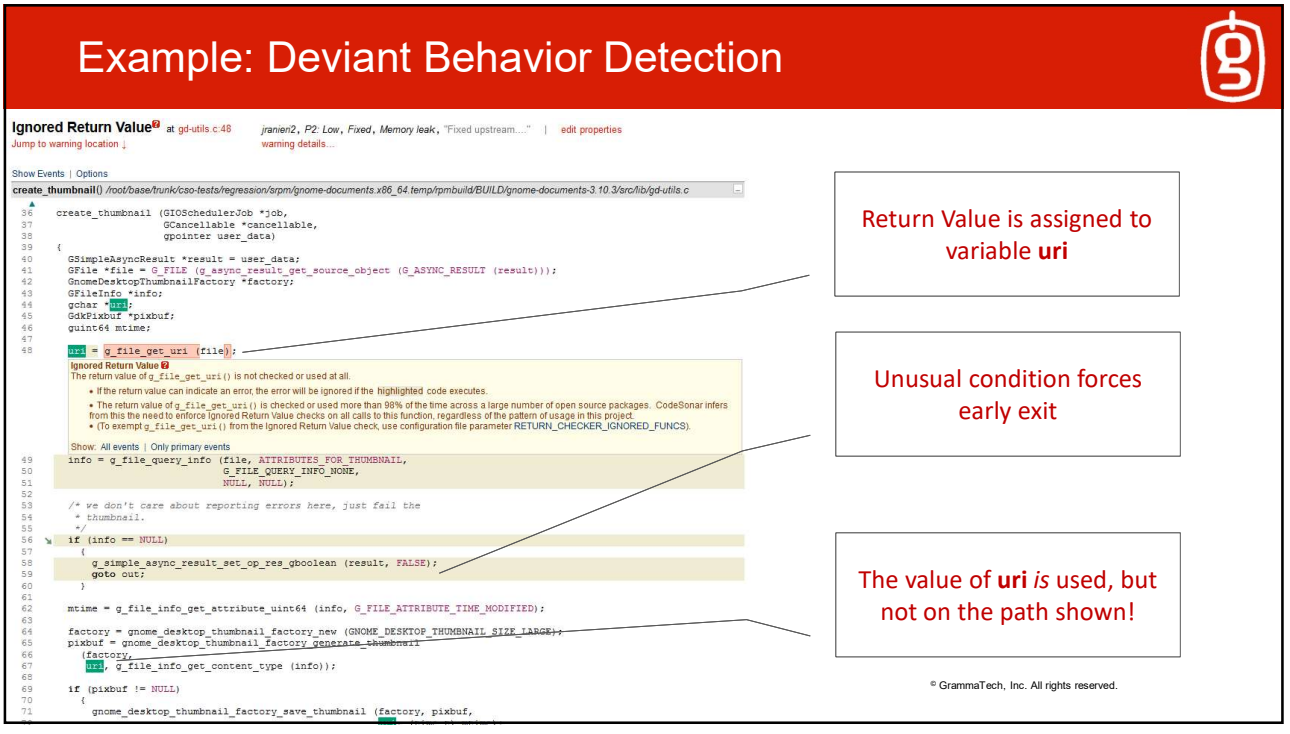

```
118 void
119 more_variables ()
120 {
121   int indx;
122   int old_count;
123   bc_var **old_var;
124   char **old_names;
125
126   /* Save the old values. */
127   old_count = v_count;
128   old_var = variables;
129   old_names = v_names;
130
131   /* Increment by a fixed amount and allocate. */
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
175   /* Initialize the new elements. */
176   for (; indx < v_count; indx++)
     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?
177     arrays[indx] = NULL;
178
179   /* Free the old elements. */
180   if (old_count != 0)
181     {
182       free (old_ary);
183       free (old_names);
184     }
185 }
```

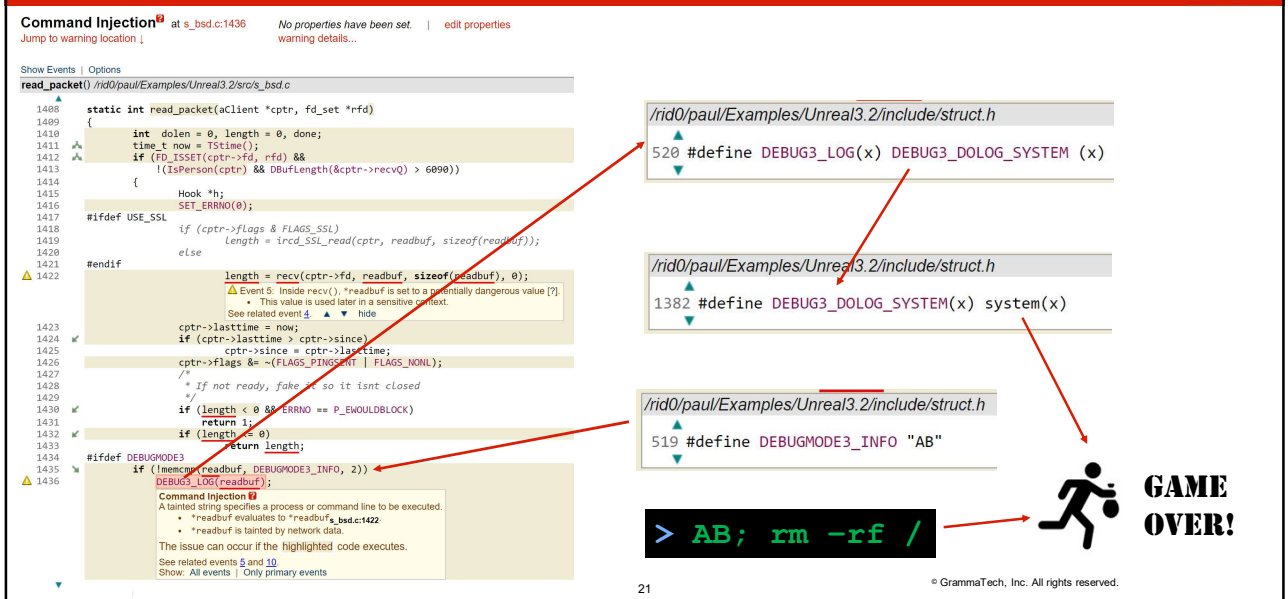
Example: Concurrency Analysis



Example: Deviant Behavior Detection



Example: Taint Analysis (aka Hazardous Information Flow)



Overview

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

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

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**

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

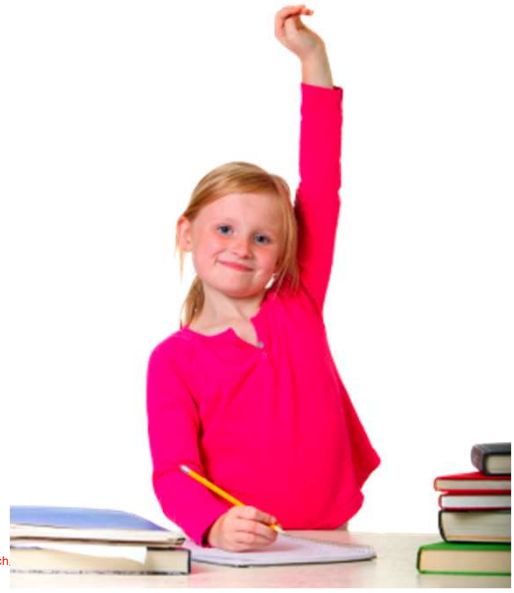
Real Functional Safety is More Important than Conformance

Do use a coding standard and check for violations,

but….

Don't assume that conformance guarantees safety

Questions?



- My contact info:
  - Paul Anderson
  - paul@grammatech.com
  - http://www.grammatech.com