Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions

Dawson Engler Benjamin Chelf Andy Chou Seth Hallem

[1]Computer Systems Laboratory
Stanford University

Presenter: Yoon-Kah Leow

Outline of Topics

- Motivation for using Meta-level Compilation
- Meta-level Compilation
- Metal and xg++ Extensions
- Coverity Lessons Learnt
- Discussion and Conclusion

Checking for System-Rules Violations Using Formal Verification

1. Advantages
   1.1 Able to locate hard-to-find errors. (i.e., when defined appropriately)
2. Disadvantages
   2.1 Hard to create a comprehensive specification.
   2.2 Specifications are only an abstraction of actual code.

Can we propose something much **easier** than this?

Checking for System-Rules Violations Using Testing

1. Advantages
   - 1.1 Testing is easier than verification to implement.
   - 1.2 Operates on the actual code.
2. Disadvantages
   - 2.1 Execution paths grows exponentially with increasing code size.
   - 2.2 Difficult to track intermittent bugs.
   - 2.3 Hard to interpret test results.

Can we propose something that **scales** better than this?

Checking for System-Rules Violations Using Manual Inspection

1. Advantages
   - 1.1 Take into consideration of various semantic levels.
   - 1.2 Flexible
2. Disadvantages
   - 2.1 Difficult to scale to large code sizes.
   - 2.2 Inconsistent results.

Can we propose something that is more **consistent**?

Checking for System-Rules Violations Using Compilers

YES! Compilers are a good alternative but what is missing?

- Meta-semantics of the underlying code.

Checking for System-Rules Violations Using Compilers

YES! Compilers are a good alternative but what is missing?

- Meta-semantics of the underlying code.
- An easy yet scalable way for developers to extend the compiler, **xg++**.

Checking for System-Rules Violations Using Compilers

YES! Compilers are a good alternative but what is missing?

- Meta-semantics of the underlying code.
- An easy yet scalable way for developers to extend the compiler, **xg++**.
- Solution is **Meta-level Compilation (MC)**.

What is Meta-level Compilation?

1. Extend compilers with checkers defined as high-level state machines.
2. State machines are defined using a language called *Metal*.
3. Checkers are dynamically-linked into the compiler.

An Example *Metal* State Machine

Pattens defining state transitions

1. sti() | restore_flags() $\rightarrow$ enable
2. cli() $\rightarrow$ disable

Available States

1. is_enabled: disable $\rightarrow$ is_disabled | enable $\rightarrow$ error(double enable)
2. is_disabled: enable $\rightarrow$ is_enabled | disable $\rightarrow$ error(double disable) | end_of_path $\rightarrow$ error(exiting with interrupt request disabled!)

Potential Caveats

1. Extensions do not ensure 100% bugs-free.
2. Unable to check for proprietary systems features.
   - 2.1 Send error-logs to system designers for verification.
   - 2.2 Disregard items that is hard to reason.
3. False positives caused by local analyses.
   - 3.1 Add global analysis or system-specific information.
   - 3.2 Provide extra API calls to suppress warnings.
4. Compiler compatibility issues with other languages.
   - 4.1 Remove illegal GNU 'C' constructs.
   - 4.2 Relax type-checking in g++ compiler front-end.
   - 4.3 Port *Metal* language based checkers to support other languages.

Expected Behavior of Assertions

MC detects code deviation in assertion constructs based on the following 2 behavior definitions.

1. *assert* is used typically during development.
2. *assert* should never fail.

Assertion Checking Extension State-Machine

Patterns defining state transitions

1. Any statement resembling the form *assert*(*expr*).
2. Variables of any type in *expr*.
3. Any function call with any arguments in *expr*.

Available States

1. start: *assert*(*expr*) $\rightarrow$ mgk_expr_recurse(expr, in_assert)
2. in_assert: any_fcall(args) $\rightarrow$ error(function call) | x=y $\rightarrow$ error(assignment) | z++ $\rightarrow$ error(post-increment) | z++ $\rightarrow$ error(post-decrement)

Assertion Checking with Static Analysis

1. Track scalar variable values using dataflow analysis.
2. Evaluate the set of values accumulated for the variable in an assert statement.
3. Set of values is the union of all previous constant assignments.

Temporal Orderings

MC checks for code execution order in the following scenarios.

1. System calls should always validate application pointers before using them.
2. Execution order during memory allocation and deallocation.
   - 2.1 Check returned pointer handler after *malloc* invocation before using it.
   - 2.2 Ensure deallocated memory is not used.
   - 2.3 Paths that exits with an error needs to deallocate memory.
   - 2.4 Allocated memory size cannot be lesser than the size of the object.

Tracking Pointer Copying Between Kernel and User Space State-Machine

1. Track all pointer variables during each system call.
2. Mark each pointer variable as *tainted*.
3. Eliminate/kill each pointer variable if it is re-assigned or passed on to a *tainted* function.

Extensions Dealing with Global Contexts

Expected global Linux rules.

1. Check for kernel code invoking a blocking function during interrupt disabled or while holding a spin-lock.
2. Check for kernel code invoking a blocking function during kernel module loading.

Global analysis to detect blocking routine

1. Invoke local pass using a *Metal* extension.
2. Mark every blocking kernel routine.
3. Generate a global call-graph.
4. Invoke global pass and mark all routines that invokes a blocking routine.

Check for deadlocks in the Linux Kernel

Pattens defining state transitions

1. sti() | restore_flags() $\rightarrow$ enable
2. cli() $\rightarrow$ disable

Available States

1. is_enabled: disable $\rightarrow$ check_blocking_functions(is_disabled)
2. is_disabled: enable $\rightarrow$ is_enabled

Optimization Extensions

Created extensions to optimize FLASH cache coherence protocol in the following areas.

1. Buffer optimization to remove redundant buffer allocation for control messages.
2. Detects redundant default message's buffer length.
3. Applying XOR operation to consecutive messages that have headers the same as the previous message.

Coverity Lessons Learnt

After commercialization of their product under the brand name, Coverity, the authors experience the harsh reality from the commercial world.

1. Associating coding practices with compiler.
2. Managing unconventional compiler extensions in embedded design development.
3. Acquiring out-dated compiler designs.
4. Customers are not concern with bugs.
5. Programmers have a weak grasp in compilers and disregard hard-to-understand bugs.
6. Customers have low trust in the product.
7. A good error is an error which can be diagnosed easily.

Managing Unconventional 'C' Code with a 'C' Compiler

Utilise a third party software to manage variations in 'C' language.

1. Utilize Edison Design Group (EDG) front-end. Design targets feature compatibility and version-specific bug compatibility.
2. Develop code 'transformers' to parse code into reasonable intermediate representations to be interpreted by EDG.
3. Hack EDG code to ensure compatibility!

Discussion and Conclusion

1. A good bug is easy to diagnose yet hard to dispute.
2. Users emphasizes on a tool which is **scalable**, **easy** to use, and maintains high standard of **accuracy** with low number of false positives.
3. Validation engineers might not have sufficient knowledge of the tested code to write the necessary extensions.