Bugs as Deviant Behavior: A General Approach to Inferring Bugs in Systems Code

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf

Computer Systems Laboratory

Stanford University

Outline

- Introduction
- Related work
- Methodology
- Internal consistency
- Statistical analysis
- Implementation
- Evaluation
- Conclusion

Introduction

- Problem: Hard to make sure programs are correct because it is difficult to know which rules the program must follow.
- Some rules are common and well-known such as "don't de-reference null pointer". Some rules are system specific such as "kernel should not de-reference user pointer".

Solution:

- Develop a tool or method which derives the rules for correctness from the code and statically checks adherence to the rule in the code.

Solution:

- Statically detect errors in code.
- MUST beliefs look for contradictions in code logic.
- MAY beliefs look for contradictions in programmer behavior.
- Demonstrated the technique on large real system such as Linux.

Solution:

- The methods used can take advantage of existing compiler optimization techniques such as constant propagation and dead code elimination.
- In some ways, it appears to be a maturing of the warnings that are displayed from gcc using –Wall flag.
- In some ways it is applying compiler optimizing technology to testing.

Related Work

- Most other testing techniques require intimate detailed knowledge of rules (formal specification) or a large well designed test set to exercise all the code (traditional).

Methodology

- Develop generic rule templates.
  - Example: <a> must be paired with <b>
- Code is statically analyzed and a set of beliefs is inferred for template matches.
- Contradictions are noted.

Internal Consistency and Statistical Analysis

- Internal consistency looks for contradictions in MUST beliefs and flags them as errors.
  - MUST beliefs are based on templates for rules that must always be followed, ex. Don't de-reference null pointer.
- Statistical analysis looks for contradictions in MAY beliefs and ranks them from most probable error to least.
  - MAY beliefs are all combinations that match a template for a rule that may apply to matching code.

Internal Consistency (MUST Beliefs)

- Propagate MUST beliefs throughout code.
- Every code path has it's own MUST belief sets.
- Example: Null pointer checks.

Example: Null Pointer Checker

- Rule: Don't de-reference null pointers
- and don't check for null when it can be shown to be not null at compile time.
- For pointer *p* propagate the set of belief values through the code.

Pointer Consistency

- Possible belief sets: {}, {*null*}, {*not null*}, {*null*, *not null*}
- Initially the belief set is empty set
- A de-reference of *p* sets the belief set to {*not null*}.
- Expression *p* == *null* sets the belief set to {*null*} along the true path and {*not null*} along the false path.
- An assignment to *p* will cause the belief set to be {*null*, *not null*}.

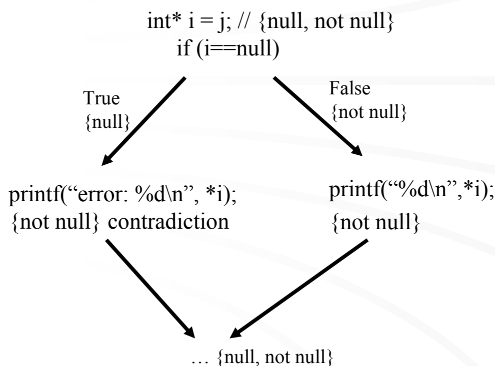
Pointer Consistency

- Contradicting MUST belief sets indicate errors or programmer misunderstanding.
- Example: if the belief set is {*null*} then it shouldn't be de-referenced because de-referencing has the belief set {*not null*}.

Example

1. …
2. int* i = j; // {null, not null}
3. if (i==null){ //{null}
4. printf("error: %d\n", *i); //{not null}
5. } else { // {not null}
6. printf("%d\n",*i); //{not null}
7. } // {null, not null}

- Line 4 has a logical inconsistency.
- Line 7 belief set is the union of the two path belief sets.



- Tracing possible values of expression through the paths of code can be quite complex.
- Loops, function calls, etc. must all be handled.
- Fortunately, compiler methods, algorithms, and tools can be utilized for this effort.

Another Example: User level Pointers in the Linux Kernel

- Pointers passed to the Linux kernel from user space are considered unsafe.
- No attempt to de-reference user pointers should be made.
  - Instead user pointers should be sent to a special copy function.
  - The copy function result can be safely de-referenced.
- Kernel pointers can be safely de-referenced
  - It is considered inappropriate to send kernel pointers to the copy function and indicates programmer confusion.

User Pointer Template

- Is <*p*> a dangerous user pointer?
  - If *p* is passed to copy function then it is dangerous. MUST belief = {*danger*}
  - If *p* is de-referenced it is safe. MUST belief = {*safe*}
- We don't need to know every detail of the Linux kernel code to check this rule for the entire system.

An error catch in the kernel

- The following code error was caught.
  - The programmer copied the user pointer but continued to use the dangerous user pointer.

```
//user is unsafe pointer, user pointer copied to safe.
copy_from_user(safe,user);
parse_quos(user); //continued using user pointer
```


Statistical Analysis (MAY Beliefs)

- Statistical analysis attempts to infer rules about system behavior based on programmer behavior.
- Look for changes in the way a particular code fragment is usually organized.

- The method is similar to internal consistency.
- Modifications include:
  - Assume all combinations matching the template are MUST beliefs.
  - Count the number of times a template slot was checked and how many inconsistencies were found.
  - Not all inconsistencies are errors so a ranking is made from most plausible to least plausible.

- Because statistical analysis is based on statistics a large sample size is needed for accurate results.
- Considering all the combinations that fit a template is usually impractical, preprocessing is needed to reduce the the number of cases.

Example

- Lock <*l*> protects variable <*v*>.
- A slot for this template will be filled for every combination of lock *l* and variable *v* where *v* is between the lock and unlock of *l*.

- Consider:

```
lock(l);
a = g+e;
unlock(l);
```

- The following slots will be created.
  - Lock *l* protects variable *a*.
  - Lock *l* protects variable *g*.
  - Lock *l* protects variable *e*.
- Lock *l* may only protect one of the variables and that should become apparent when all results are ranked.

- If lock *l* is used 1000 times and protects *a* 999 times then it is probable that *a* was the intended protected variable.
- This case will get a high ranking because the one case left is almost surely a bug.

Other Uses of Statistical Analysis

- Ensuring functions returns are properly evaluated.
- No $< a >$ after $< b >$.
  - Example: Don’t use de-allocated memory.
- $< b >$ must follow $< a >$.
  - Example: locks must be released.

Inferring Failure

- Some functions should always have their return value checked to ensure success or handle a failure.
- It is also important to make sure the programmer understands how to evaluate return values.
- Even Java can’t protect programmers from this because many get into the habit of throwing all exceptions back to the main function.

Implementation

- *xgcc* is an extended version of the gcc compiler.
- *xgcc* can be extended further with the state machine language *metal*.
- *Metal* is very similar to yacc in that it parses patterns, making it very handy for specifying templates.

System Dependencies

- Overall the checkers are mostly system independent.
- Some dependence exists in the names of functions used for some basic system operations.

Ranking

- Local errors then global errors.
- Short code span over long span
- Serious errors over minor errors.
- Templates with no matches will sometimes indicate an error.
- $Z(n,e) = (e/n - p_0) / (p_0 * (1 - p_0)/n)^{1/2}$
  - $p_0 = 0.9$
  - n = sample size
  - e = number of successes

Noise

- Statistical analysis is kind of fuzzy.
- Coincidences and imperfections can cause false positives or low ranked bugs.
- Counter noise with
  - Large samples
  - Ranking error messages
  - Human-level operations

Evaluation

- The Linux and OpenBSD kernels were evaluated for several error types
- A large number of errors were found and code patches were issued as a result of this work.

Internal Null Checker Results

- As mentioned before, internal consistency
- Linux 2.4.7
  - Check then use: 79 bugs found, 26 false positives.
  - Use then check: 102 bugs found, 4 false positives.
  - Redundant check: 24 bugs found, 10 false positives.

User Pointer Checker

- Internal consistency
- OpenBSD 2.8
  - 18 errors
  - 3 false positives
  - 1645 instances
- Linux 2.4.1
  - 12 errors
  - 16 false positives
  - 4905 instances
- Linux 2.3.99 had 5 errors

Derived Null Checker

- Linux 2.4.1
  - 154 errors
  - 16 false positives
- OpenBSD 2.8
  - 41 errors
  - 21 false positives

Conclusion

- The rules of the system don’t have to be known.
  - Logical inconsistencies are errors.
  - Programmer behavior inconsistencies usually indicate an error.
- The techniques shown have been applied to real systems and found many real bugs for less effort than would otherwise be required.

More Conclusion

- Internal consistency checks violations of MUST beliefs
  - MUST beliefs should have no contradictions.
  - Internal consistency will work on small programs.

More Conclusions

- Statistical analysis checks violations of MAY beliefs.
  - The results are ranked from most probably an error to least probably an error.
  - While the false positives are probably going to be more numerous, the templates can be more general.
  - Requires a large system to give many sample cases.

More Conclusion

- The effort required from these techniques is considerably smaller than other techniques.
  - The required knowledge of the system is small.
  - A set of test cases is not required.
  - Previous compiler techniques and tools can be utilized.

Critique

- The paper had problem, solution, implementation, evaluation.
- Despite questions about errors and false positives, I can't deny that it worked well on real systems.

More Critique

- (Depending on the intended audience), more could have been spent explaining the checkers relationship to compiler optimization.
- The paper layout was not good.
- The paper was too long.

Discussion

- All of the examples are low-level, can this work for high level checks?
- The checkers are integrated into the compilers. The authors didn't explain the details of this.
  - Should the checks be done before compiling, before optimization, with optimization, after optimization?
  - How much of this is already done in the compiler? (Warning: a may have not been initialized.)

More Discussion

- The purpose of functions is to write common code once, does it seem reasonable to expect large data sets from statistical analysis in a well designed system?
- What kind of errors can static checking find that run-time checking can't?
- What kind of errors can run-time checking find that static checking can't?

More Discussion

- If a programmer does something wrong every single time, will the statistic analysis checker ever rank the problem?
- …will the Internal consistency checker ever report contradiction?

Thank You