Software Safety

(Static) Code Analysis

Prof. Dr.-Ing. Patrick Mäder, M.Sc. Martin Rabe

Contents

Overview

Motivation

- Software is hard to get right
  - Complex APIs
  - Difficult language features (concurrency, asynchronicity)
- Nobody is perfect 100% of the time
- Result: **bugs**
  - Wasted development time
  - Frustrated users
  - even worse for safety-critical software $\rightarrow$ harm

Use Tools

- Create programs, *bug checkers* or *linters*, to analyze code for potential errors (including stylistic errors)
- Running the bug checker produces a list of potential bugs in the code
- Goal: find bugs early
  - Before debugging and testing
  - Before program is distributed to users

Is it worth going through this kind of reports?

- Here's an example – in June of 1996, a certain mission ignored this static analysis report:
  - "The code at this point is multiplying acceleration with time..."
  - "...both are floating point variables..."
  - "...the result is stored in a 16-bit integer"
  - "...this could go out of bounds!"
- The warning was ignored...
  - ...lost amongst lots of similar ones
  - ...that were "harmless"
  - ...and a mission was lost.
- It's not just floating point vs. integer mismatches
  - Some static analyzers track the exact ranges of variables (e.g., -10 to 10), so the analyzer can "see" the exact range of, e.g., a multiplication's result – and warn if it won't always fit in the target variable!
  - Add data flow errors, potential divisions by zero, accesses of unassigned variables, variables assigned but never used, null pointer accesses...

Observation

- Many bugs share common characteristics
- *Bug Pattern*: a code idiom that is frequently an error
- Automatically detect instances of bug patterns

SpotBugs

Tool: SpotBugs

- Static analysis tool to find bugs in Java code
- Spiritual successor of *FindBugs*
- Checks for more than 400 bug patterns in several categories

Bug Categories

- **Bad practice:** Violations of recommended and essential coding practice (equals, clone, dropped exceptions)
- **Correctness:** mistakes, unintended by developers (∞ loops)
- **Performance:** correct, but inefficient code (boxing, string handling)
- **Security:** use of untrusted input (SQL injection, hard coded passwords)
- **Dodgy Code:** code that is confusing, anomalous, or written in a way that leads itself to errors (style, redundant checks)

Limitations

- *Static Analysis*: process of analyzing a program's code to find out how the program will behave at runtime
- Nontrivial properties of programs are undecidable (e.g., Halting Problem)
- All possible program behaviors are not determined

Consequences of Imprecision

- Can't predict all possible program behaviors
  - Try to infer *likely* program behavior
- False positives: reporting bugs that can't really happen
- False negatives: failing to report a bug that can happen

Hands-on

- download SpotBugs here: Link
- install it by *unpacking*, e.g., to `Downloads/` folder
- download examples from Moodle
- unpack somewhere
- open projects editor
- examine code

Refresher: Java Tooling

Compiling:

```
cmd> javac App.java
cmd> "C:\Programs\jdk<version>\bin\javac" App.java
```

Running:

```
cmd> java App
cmd> "C:\Programs\jdk<version>\bin\java" App
```

How it works

- Analyze bytecode using the Apache Byte Code Engineering Library (BCEL)
- Several approaches
  - Simple: Scanning
  - More complex: Scanning with control flow
  - Most complex: Data flow analysis
- Similar to techniques used in compilers

Bytecode Scanning

- Scan through bytecode instructions, driving a state machine
- Example: unconditional wait

```
synchronized (lock) {
    lock.wait();
    // perform task
}
```

```
synchronized (lock) {
    while (!someCondition) {
        lock.wait();
    }
    // perform task
}
```

Bytecode Scanning

- Scan through bytecode instructions, driving a state machine
- Example: unconditional wait

```
synchronized (lock) {
    lock.wait();
    // perform task
}
```

bad

```
synchronized (lock) {
    while (!someCondition) {
        lock.wait();
    }
    // perform task
}
```

good – proper handling of spurious wakeups

Control flow

- Scanning is good for bug patterns that don't involve control flow
- Control flow is important and provides information

```
if (value != null) {
    // ...value is not null here...
}
```

- representation: Control Flow Graph

Control Flow Graph

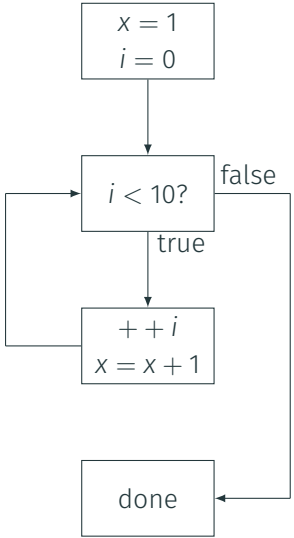
- Node: linear sequence of instructions with no control flow
- Edge: indicates control transfer from one block to another

```
x = 1;
for (i = 0; i < 10; ++i) {
    x = x + 1;
}
```

Control Flow Graph

- Node: linear sequence of instructions with no control flow
- Edge: indicates control transfer from one block to another

```
x = 1;
for (i = 0; i < 10; ++i) {
    x = x + 1;
}
```



Application of CFG

- Scanning approach can now take control flow into account
- For example: for an “ if” statement, continue scanning on both branches
- This can lead to **exponential cost**

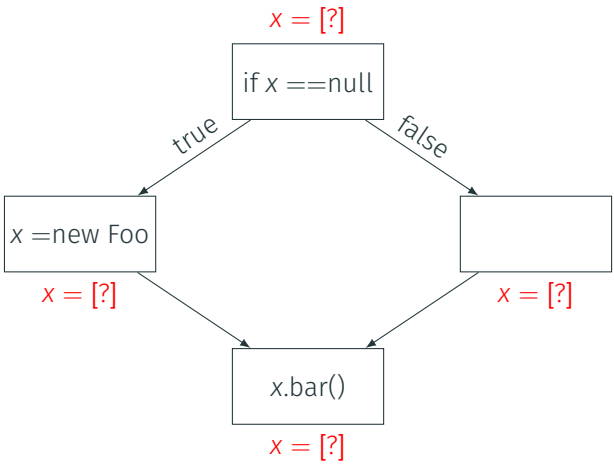
Dataflow Analysis

- Conservatively approximate facts about a program (e.g., Used extensively in compilers)
- For example: *"where might a null pointer be dereferenced"*
- Models the values of variables (locations on the operand stack), taking control flow into account

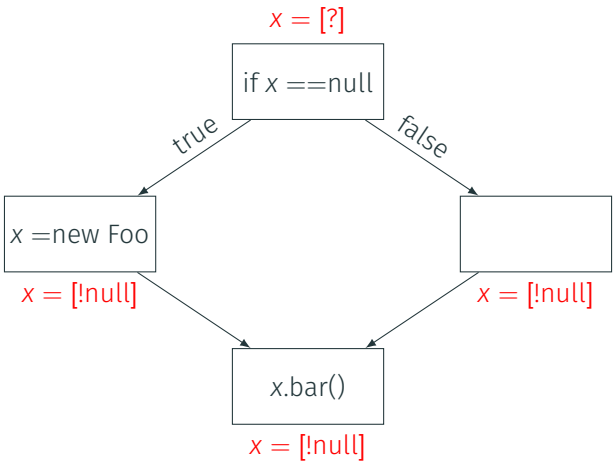
Dataflow Analysis II

- Dataflow value is an abstract representation of a runtime value
  - Foe example: *"value is null"*, *"value is not null"*, *"value could be either"*
- Transfer functions take dataflow values and model the effects of a basic block
- A merge function combines data flow values for when control paths merge

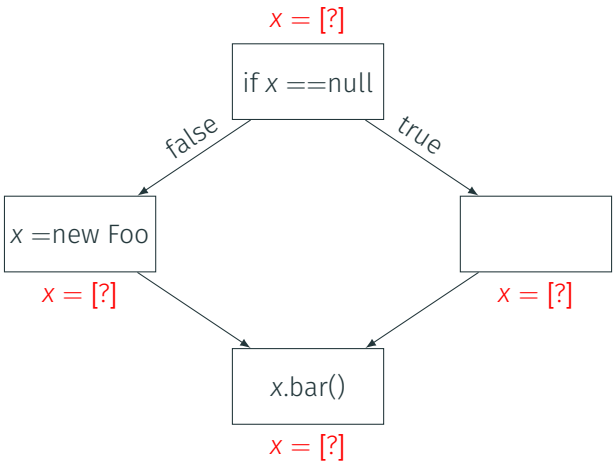
Dataflow Example



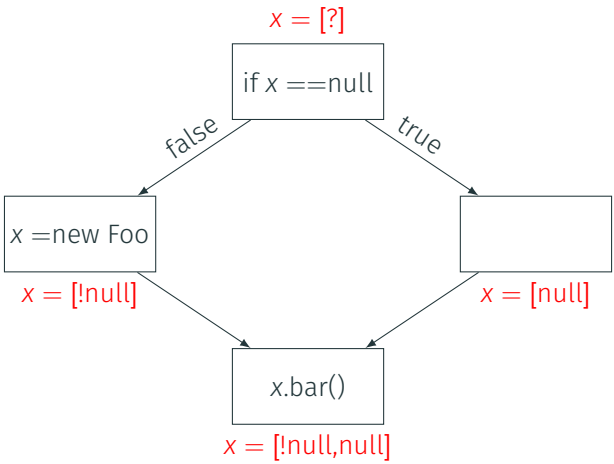
Dataflow Example



Dataflow Example II



Dataflow Example II



Hands on: JFreeChart

- Open source Java framework for chart calculation, creation and display.
- Download it here: Link
- Open with SpotBugs
- Analyse found issues

cppcheck

Cppcheck: an analysis tool for C/C++ code

- It provides unique code analysis to detect bugs and focuses on detecting undefined behaviour and dangerous coding constructs.
- It is designed to be able to analyze C/C++ code even if it has non-standard syntax (common in embedded projects).
- Both command line interface and graphical user interface are available.

Unique analysis

- Cppcheck uses unsound flow sensitive analysis.
- It will detect bugs that the other tools using path sensitive analysis based on abstract interpretation do not identify.
- In Cppcheck the data flow analysis is not only ”forward” but ”bi-directional”.

Most analyzers will diagnose this:

```
void foo(int x)
{
    int buf[10];
    if (x == 1000)
        buf[x] = 0; // <- ERROR
}
```

Most tools can determine that the array index will be 1000 and there will be overflow.

Cppcheck will also diagnose this:

```
void foo(int x)
{
    int buf[10];
    buf[x] = 0; // <- ERROR
    if (x == 1000) {}
}
```

Addons

- Addons are scripts that analyse Cppcheck dump files to check compatibility with secure coding standards and to locate issues.
- Cppcheck supports the following addons:
  - cert.py: checks for compliance with the safe programming standard SEI CERT.
  - misra.py: is used to verify compliance with MISRA C 2012
  - y2038.py: checks Linux systems for year 2038 problem safety.
  - threadsafety.py: analyses Cppcheck dump files to locate thread safety issues like static local objects used by multiple threads.

MISRA

- MISRA C is a C programming standard developed by the automotive industry
- stands for **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation
- first released in 1998
- third and current edition released in 2012
- goal is to facilitate code safety, security, portability and reliability in the context of embedded systems
- 143 rules grouped into 22 topics

Example: Rule 4.2 Trigraphs should not be used

- ?? followed by specific third character are replaced by another character
- e.g., ??- is replaced with ~
- Problem if for example used as separator:

```
//Reading the following text with ??- as separator from file
"First??-Second??-Third"
//Results in
"First~Second~Third"
//Instead of
("First", "Second", "Third")
```

Demo

```c
int fibonacci(int number)
{
    //Rule 17.2/15.5
    if (number > 0)
    {
        int result = 0;
        if ((number == 1) || (number == 2))
        {
            result = 1;
        }
        else
        {
            result = fibonacci(number - 1) + fibonacci(number - 2);
        }
        return result;
    }
    else
    {
        (void)printf("Incorrect number for the calculation of fibonacci numbers\n");
        return -1;
    }
}
```

Klee

KLEE

- symbolic execution engine
- works on bytecode in interpreter like fashion
- marked variables are used symbolically and not assigned a value
- restrictions accumulated during program execution (<,>, ...) are applied to the variables
- at function call KLEE checks if a possible value of the variable can induce an error
- runtime limitations require heuristical approach

Advantages

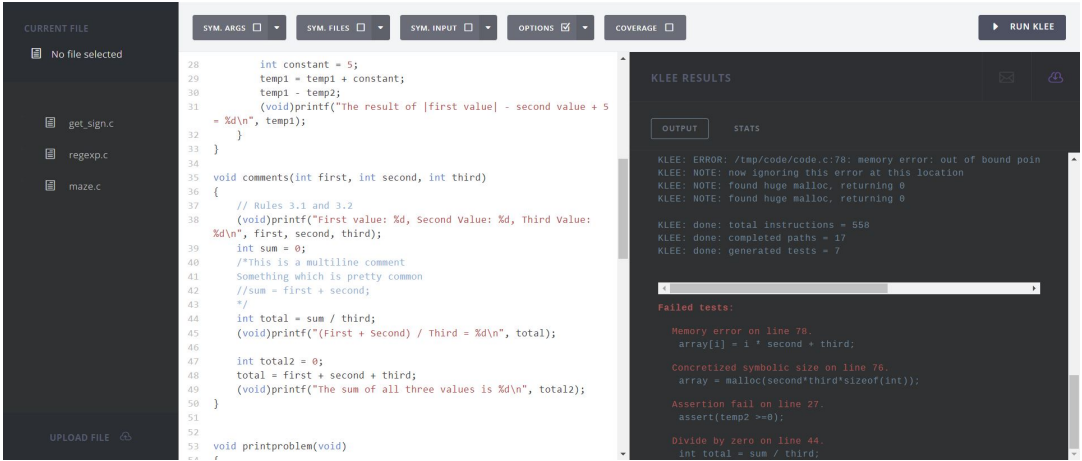
- extremely high degree of code coverage, often higher than the tests of highly tested programs
- shown to be able to find errors that had not been detected for years
- easy to implement, requiring no to little additions to the source code

	COREUTILS	
Coverage (w/o lib)	**KLEE tests**	**Devel. tests**
100%	16	1
90-100%	40	6
80-90%	21	20
70-80%	7	23
60-70%	5	15
50-60%	-	10
40-50%	-	6
30-40%	-	3
20-30%	-	1
10-20%	-	3
0-10%	-	1
Overall cov.	84.5%	67.7%
Med cov/App	94.7%	72.5%
Ave cov/App	90.9%	68.4%

Disadvantages

- floats require an external plugin to be able to be analysed
- only the first error on a path can be detected
- system is not emulated perfectly
- checks only syntactical correctness and program integrity, but not functional correctness
- higher runtime compared to debugging
- heuristical approach means that not all possible paths are evaluated
- web implementation specifically:
  - missing float plugin
  - only one option flag possible at a time

Demo



http://klee.doc.ic.ac.uk/

Coccinelle

Coccinelle

- tool for matching and transforming C code
- all instances of a pattern can be found and modified/replaced
- uses a semantic patch language (SmPL) to achieve this
- can run on single files or entire directories
- created to aid the development of the linux kernel, specifically for collateral evolution and bug finding and fixing

Advantages

- simple but powerful syntax
- can evaluate related code fragments regardless of intervening code
- it understands C syntax, specifically many isomorphisms
- usable on extremely large projects, e.g., Linux kernel
- uses linux patch-file system $\rightarrow$ easy integration in existing workflow and checking of created patches
- high degree of success: when applied to 5800 Linux files, success rate of 100% for 93% of files
- performance: average of 0.7s per file with average semantic patch size of 106 lines of code

Disadvantages

- can only find bugs who's syntax has been explicitly described
- limits Coccinelles to detecting bugs who's syntax structure is known
- it's degree of success is dependant on the quality of the patterns written by the user
- dealing with namespaces also depends on pattern quality
- features from C derivative languages such as C++ (classes, member functions, C++ style namespaces) are not supported

Semantic Patch Example

```
@@
expression lock, flags;
expression urb;
@@

 spin_lock_irqsave(lock, flags);
 <...
- usb_submit_urb(urb)
+ usb_submit_urb(urb, GFP_ATOMIC)
 ...>
 spin_unlock_irqrestore(lock, flags);

@@
expression urb;
@@

- usb_submit_urb(urb)
+ usb_submit_urb(urb, GFP_KERNEL)
```

Demo

```
@Rule1@
expression list E;
@@

- printf(E);
+ assert(printf(E));

@Rule2@
identifier func;
identifier message;
expression mode;
@@

void func(...){
...
- setTransmissionMode(mode);
...
- transmit(message);
+ transmit(message, mode);

}
```

Additional Tools

Additional Tools

- valgrind
  - tool for memory debugging, memory leak detection and profiling
- gdb
  - GNU Debugger for many programming languages, including ADA, C and C++