

Static Analysis

17-313 Fall 2026

Foundations of Software Engineering

<https://cmu-17313q.github.io>

Eduardo Feo Flushing



Learning Goals

- Learn to get early feedback to reduce risk.
- Find ways to catch our technical errors.
- Gain an understanding of the relative strengths and weaknesses of static analysis.
- Examine several popular analysis tools and understand their use cases.
- Understand how analysis tools are used in large open source software.

Administrivia

- Past exams posted.
- Cheat sheet:
 - One double-sided A4.
 - You must submit it.
 - Handwritten = bonus points.
 - Printed cheat sheets permitted but not awarded points.
- Midterm: Next Tuesday.
 - We will use two rooms.
 - You will receive an email with instructions.
- Review session: Tomorrow during Recitation

Outline

- Why static analysis?
- What can we determine without running code?
- Formatting and pattern-based linters
- Type annotations and taint analysis
- Making analysis useful in practice
- Takeaways

Outline

- **Why static analysis?**
- What can we determine without running code?
- Formatting and pattern-based linters
- Type annotations and taint analysis
- Making analysis useful in practice
- Takeaways

Can you spot the problem?

```
1 static OSStatus SSLVerifySignedServerKeyExchange(
2     SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
3     uint8_t *signature, UInt16 signatureLen) {
4     OSStatus err;
5     ...
6     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
7         goto fail;
8     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
9         goto fail;
10    goto fail;
11    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
12        goto fail;
13    ...
14 fail:
15    SSLFreeBuffer(&signedHashes);
16    return err;
17 }
```

NOTES

A "goto fail;" here jumps to the cleanup/return label. Which statements are actually controlled by the second `if`? The duplicated jump always executes, making the final check unreachable: `err` stays 0 (success) even though the signature was never actually verified. What just happened?

- Bypassed Security: The device skipped checking if a website's security certificate was real.
- Man-in-the-Middle Attacks: Hackers on public Wi-Fi could trick devices into trusting fake connections and read or change encrypted data like emails and bank passwords.

goto fail;

The duplicated jump shipped in Apple's SSL/TLS stack (iOS, OS X) and went **unnoticed for nearly a year and a half. Millions of users were actively at risk;** there is zero data on how many malicious actors actually took advantage of the flaw before Apple fixed it.

Analysis

Apple's SSL iPhone vulnerability: how did it happen, and what next?

Charles Arthur

SSL vulnerability in iPhone, iPad and on Mac OS X appeared in September 2012 - but cause remains mysterious as former staffer calls lack of testing 'shameful'

goto fail; // [Apple SSL bug test site](#)

This site will help you determine whether your c

YOUR BROWSER IS VULNERABLE

We have examined your OS and browser version information and de our test image after seeing an invalid ServerKeyExchange message. **networks**) can freely **snoop on you**, for example when you log into them right away. **Other applications on your system** such as mail.

Apple's SSL vulnerability is still active on Safari on Mac OS X as shown at the gotofail.com site.

Photograph: Public domain Photograph: Public domain

tomorrow belongs to those who embrace it today

ZD NET

trending tech innovation business security advice buying guides

Home / Business / Companies / Apple

/ business

When will Apple get serious about security?

The tech community (and beyond) is an uproar over the recently revealed iOS and OS X SSL/TLS code flaw. Apple developers have questions about Apple's commitment to quality and the flaw itself.

Written by **David Morgenstern**, Contributor on Feb. 23, 2014

NOTES

Timeline: The bug was introduced around September 2012 (with iOS 6) and patched by Apple in February 2014 via iOS 7.0.6 and OS X 10.9.2

The point is how long an exploitable, structurally obvious bug can hide.

Can you spot the problem?

```
1  /* Linux 2.3.99 drivers/block/raid5.c */
2  static struct buffer_head *
3  get_free_buffer(struct stripe_head *sh,
4                  int b_size) {
5      struct buffer_head *bh;
6      unsigned long flags;
7      save_flags(flags);
8      cli(); // disable interrupts
9      if ((bh = sh->buffer_pool) == NULL)
10         return NULL;
11     sh->buffer_pool = bh->b_next;
12     bh->b_size = b_size;
13     restore_flags(flags); // restore
14     return bh;
15 }
```

ERROR: function returns
with interrupts disabled!

The return that skips cleanup

NOTES

After interrupts are disabled, which return path skips restoring the previous state? `restore\_flags` restores the prior flags, not "always enables interrupts". Checking every relevant exit path is useful for resource and API protocols.

Twitter’s week year bug

“ISO 8601: the first week of the year is the week containing the first Thursday. If January 1 is a Thursday, all days of that week (Mon Dec 29 – Sun Jan 4) belong to week 01 of the new year.”

```
DateTimeFormatter
  .ofPattern("dd MMM YYYY")
  .format(zonedDateTime)
```

YYYY gives the ISO week-based year, which can differ from the calendar year near January 1.

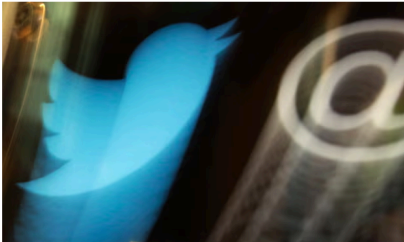
Solution: use yyyy instead.

Twitter’s Android app and mobile site went down for about five hours on December 29, 2014, reportedly because an incorrect date-format specifier caused the date to be interpreted as December 29, 2015.

December 2014							<	>
S	M	T	W	T	F	S		
30	1	2	3	4	5	6		
7	8	9	10	11	12	13		
14	15	16	17	18	19	20		
21	22	23	24	25	26	27		
28	29	30	31	1	2	3		
4	5	6	7	8	9	10		

Twitter kicks Android app users out for five hours due to 2015 date bug

The social network celebrated 2015 in style, by breaking its Android app and mobile website - and all, it seems, because of one misplaced letter



Crashy bird: Twitter was down for five hours overnight. Photograph: Richard Drew/AP
If you're worried about how your New Year's Eve will go, don't. It's not even 2015 yet, and Twitter's already had a worse one than you.
The service was down for many users over five and a half hours on Monday morning UK time, between midnight and 5am (7pm to midnight ET, and 4pm to 9pm PT), after a bug in a line of code caused the service to think that it was 29 December, 2015.

NOTES

For most dates, the calendar year and week-based year agree, so this mistake can easily survive testing. Under ISO week rules, December 29, 2014 belongs to the first week of 2015. Combining its day and month with the week-based year therefore produces “29 Dec 2015.” Tests around year boundaries could catch this. A static checker with a suitable rule could also flag the suspicious date format without executing the code.

Could you have found them?

- How often would those bugs trigger?
- Driver bug: what happens if you return from a driver with interrupts disabled? Consider: that's one function...
 - ...in a 2000 LOC file
 - ...in a module with 60,000 LOC
 - ...IN THE LINUX KERNEL

Some defects are very difficult to find via testing or inspection.

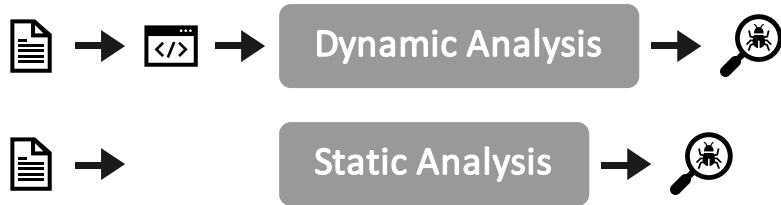
NOTES

These bugs are on uncommon or hard-to-force execution paths, exactly where testing is weakest.

Defects of interest...

- Are on uncommon or difficult-to-force execution paths (vs. testing).
- Executing (or interpreting/otherwise analyzing) all paths concretely to find such defects is infeasible.
- What we really want to do is check the entire possible **state space** of the program for particular properties.
- What we **can** do is check an **abstract state space** of the program for particular properties.

What are program analysis tools?



Dynamic analysis: examines a program by executing it with particular inputs. Examples: testing, coverage measurement, and performance profiling.

Static analysis: examines code without executing it to identify potential problems or check properties.

A screenshot of a Coveralls coverage report for a file named 'Auth.js'. The report shows line numbers 66 through 88. Lines 66-68, 70, 74-76, 78-80, 82-84, and 86-88 are highlighted in green, indicating 100% coverage. Lines 71-73 and 81 are highlighted in orange, indicating partial coverage. Line 77 is highlighted in red, indicating no coverage. The code is a JavaScript function for authentication, including comments for local logins, HTTP bearer authentication, and additional logins via SSO plugins.

A screenshot of a code editor showing a static analysis error in a file named 'src/controllers/accounts/posts.js'. The error message is: 'This function expects 3 arguments, but 4 were provided.' The error points to line 141, where the function 'getPostsFromUserSet' is called with four arguments: 'account/bookmarks', 'req', 'res', and 'next'. The function definition on line 140 shows it expects three arguments: 'req', 'res', and 'next'.

Activity: Analyze the Python program statically

```
def n2s(n: int, b: int):  
    if n <= 0: return '0'  
    r = ''  
    while n > 0:  
        u = n % b  
        if u >= 10:  
            u = chr(ord('A') + u-10)  
        n = n // b  
        r = str(u) + r  
    return r
```

1. What are the set of data types taken by variable `u` at any point?
2. Can `u` be a negative number?
3. Will this function always return, either normally or by raising an exception?
4. Can there ever be a division by zero?
5. Will the returned value ever contain a minus sign "-"?

Assume `n` and `b` are integers, with no further restriction. 90 seconds in pairs.

NOTES

Assuming `n` and `b` are integers, with no further restrictions: 1. `int` or `str`. `n % b` produces an integer. If `u >= 10`, `chr(...)` converts it to a string. 2. Yes, if `b` is negative. For example, `5 % -2` is `-1` in Python. 3. No. If `n > 0` and `b = 1`, then `n // b` leaves `n` unchanged, so the loop runs forever. For example, `n2s(5, 1)`. 4. Yes. If `n > 0` and `b = 0`, `n % b` raises `ZeroDivisionError` before reaching `n // b`. 5. Yes. `n2s(5, -2)` returns `"-1"`: `u` becomes `-1`, and `n` becomes `-3`, ending the loop.

What did we establish?

- `u` is initially an integer and may become a string.
- For `n > 0`, `b = 0` fails at the modulo operation.
- For `n > 0`, `b = 1` leaves `n` unchanged, so the loop does not terminate.

NOTES

Under that precondition: the divisor is nonzero, positive `n` strictly decreases until zero, and the character conversion stays in the intended digit range. This does not by itself prove the full conversion specification.

What is Static Analysis?

- **Systematic** examination of an **abstraction** of program **state space**.

Does not execute code! (like code review)

- **Abstraction:** produce a representation of a program that is simpler to analyze.

Results in fewer states to explore; makes difficult problems tractable.

- Check if a particular **property** holds over the entire state space:

Liveness: "something good eventually happens." Safety: "this bad thing can't ever happen."

Compliance with mechanical design rules.

For the type question, we tracked whether u is an integer or a string without tracking its exact value. That is an abstraction.

NOTES

Abstraction means keeping the information needed for the question we want to answer and leaving out other details. In our example, values such as 1, 5, and 20 can all be represented as “integer” when checking types. However, that abstraction cannot tell us whether division by zero is possible. For that question, we need to track whether a value could be zero. Grouping many concrete values together makes analysis manageable, but losing detail can also make the answer less precise.

What static analysis can and cannot do

- **Type-checking** is well established.
 - Set of data types taken by variables at any point.
 - Can be used to prevent type errors (e.g. Java) or warn about potential type errors (e.g. Python).
- Checking for **problematic patterns** in syntax is easy and fast.
 - Is there a comparison of two Java strings using `==` ?
 - Is there an array access `a[i]` without an enclosing bounds check for `i` ?

What static analysis can and cannot do

- Reasoning about **termination** is **impossible** in general.

Halting problem.

- Reasoning about **exact values** is **hard**, but conservative analysis via abstraction is possible.

- Is the bounds check before `a[i]` guaranteeing that `i` is within bounds?
- Can the divisor ever take on a zero value?
- Could the result of a function call be `42` ?
- Will this multi-threaded program give me a deterministic result?
- Be prepared for "**MAYBE**".

- Verifying some advanced properties is possible but expensive.

CI-based static analysis usually over-approximates conservatively.

Static analysis has fundamental limits

The Bad News: Rice's Theorem

No algorithm can always correctly decide a nontrivial property of program behavior for every possible program.

"Any nontrivial property about the language recognized by a Turing machine is undecidable."

Henry Gordon Rice, 1953

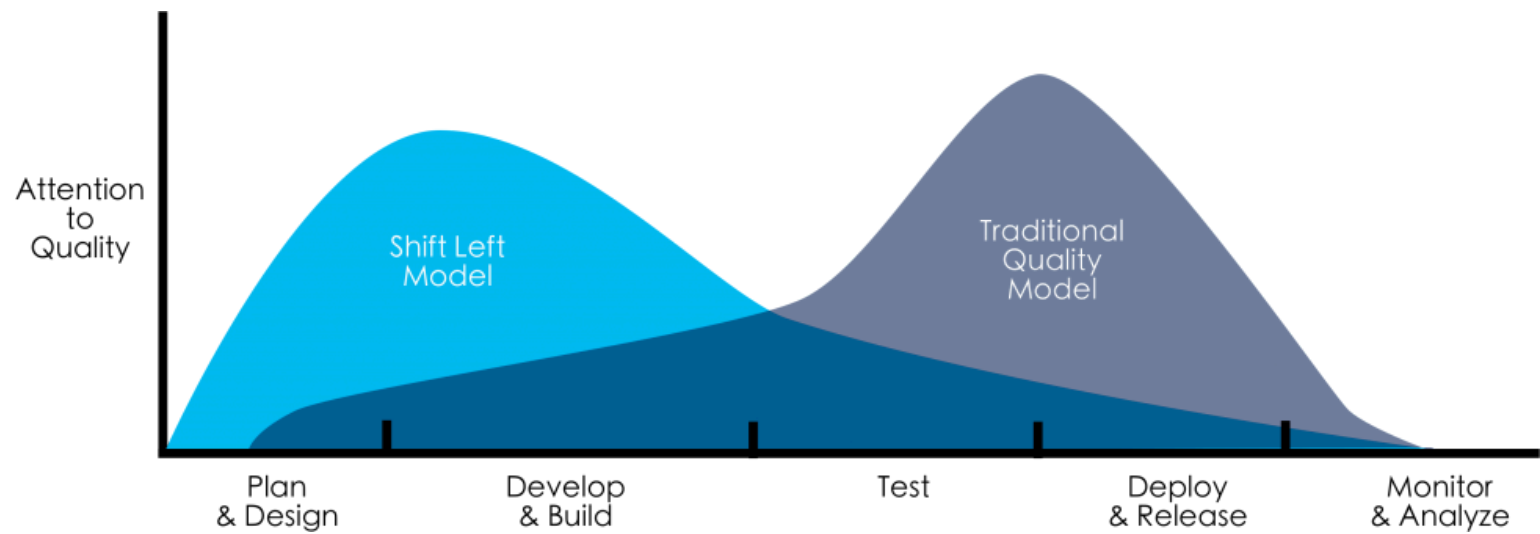
NOTES

What this means in practice: - An analyzer may report a possible bug that is not real. - It may miss some bugs or return "unknown." - Stronger guarantees are possible when we restrict the programs or properties being checked.

Static Analysis is well suited to detecting certain defects

- **Security:** buffer overruns, improperly validated input...
- **Memory safety:** null dereference, uninitialized data...
- **Resource leaks:** memory, OS resources...
- **API protocols:** device drivers; real-time libraries; GUI frameworks.
- **Exceptions:** arithmetic/library/user-defined.
- **Encapsulation:** accessing internal data, calling private functions...
- **Data races:** two threads access the same data without synchronization.

Static Analysis is Key to Shifting Left

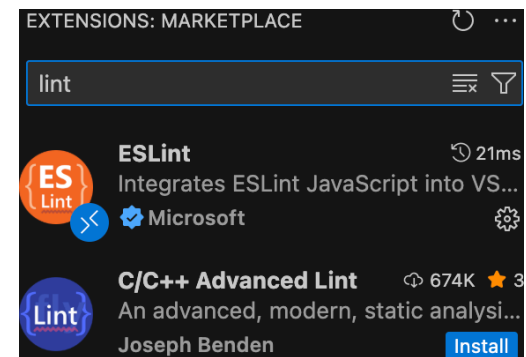
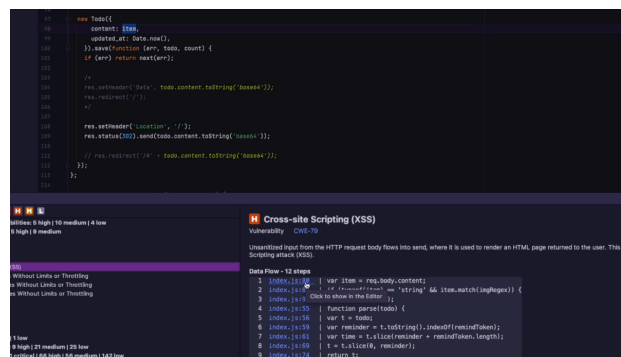
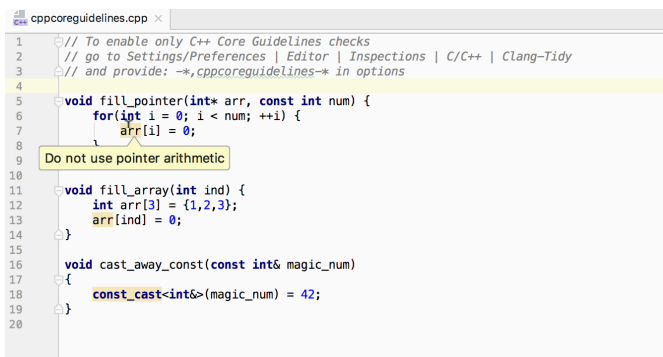


Issues are cheaper and faster to fix the earlier they are found. Find and prevent them as early as possible.

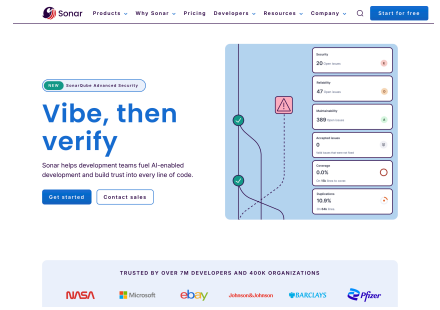
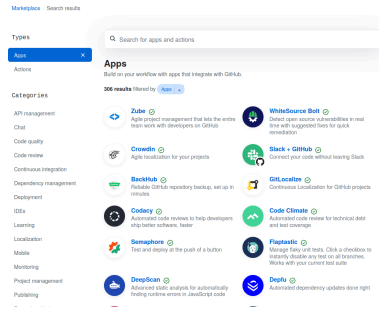
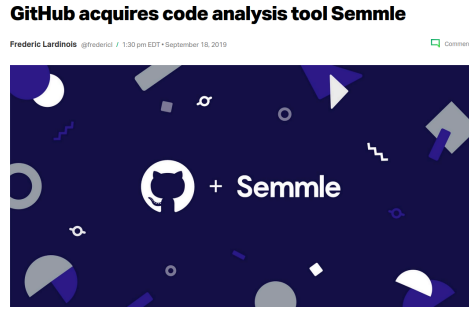
Static analysis is integrated in your IDE



Inline inspections flag issues as you type, e.g. a C++ Core Guidelines / Clang-Tidy warning on raw pointer arithmetic, or a severity-ranked security finding (XSS, SQL injection) in a scanning panel.

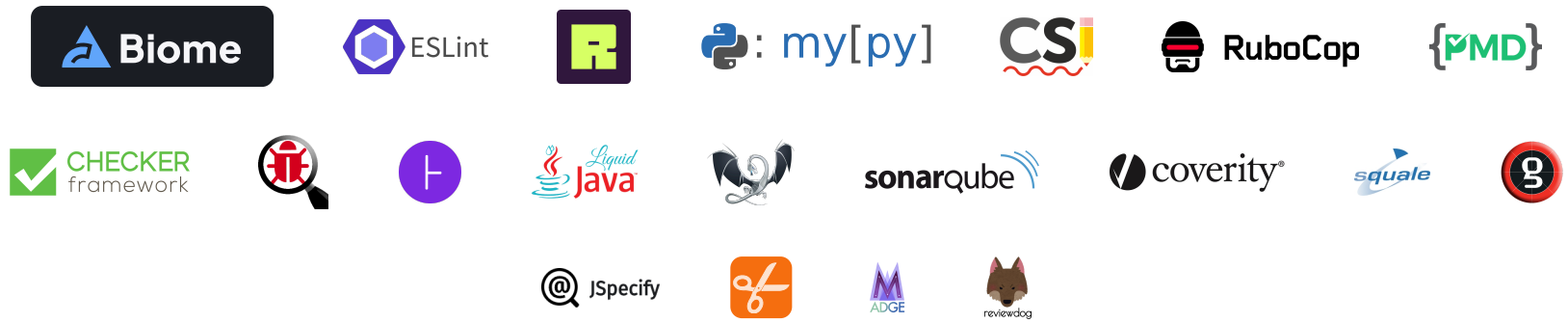


Static analysis used to be an academic amusement;
now it's heavily commercialized



GitHub acquires Semmle (2019). Snyk secures \$150M at a \$1B valuation (2020). Even AI coding assistants now market themselves on "vibe, then verify."

There are lots of static analysis tools!



Static analysis techniques we will explore

Formatting Linters

Shallow syntax analysis for enforcing code styles and formatting.

Pattern-Based Linters ("bug detectors")

Simple syntax- or API-based rules for identifying common programming mistakes or violations of best practice.

Type-Based Analysis

Checks conformance to user-defined types. Types can be complex (e.g. "Nullable").

Taint analysis

Tracks untrusted (tainted) data from a source to a sensitive sink, e.g. user input reaching a SQL query.

Outline

- Why static analysis?
- What can we determine without running code?
- **Formatting and pattern-based linters**
- Type annotations and taint analysis
- Making analysis useful in practice
- Takeaways

Formatting Linters

Cheap, fast, and lightweight static source analysis

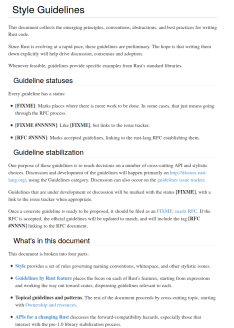
Use linters to improve maintainability

Why? We spend more time reading code than writing it.

- Developers spend most of their time maintaining code.
Various estimates of the exact %, some as high as 80%.
- Code ownership is usually shared.
- The original owner of some code may move on.
- Code conventions make it easier for other developers to quickly understand your code.

Use Style Guidelines to facilitate communication

- Indentation
- Comments
- Line length
- Naming
- Directory structure
- ...



Guidelines are inherently opinionated, but **consistency** is the important point. Agree to a set of conventions and stick to them.

Use linters to enforce style guidelines

Don't rely on manual inspection during code review!



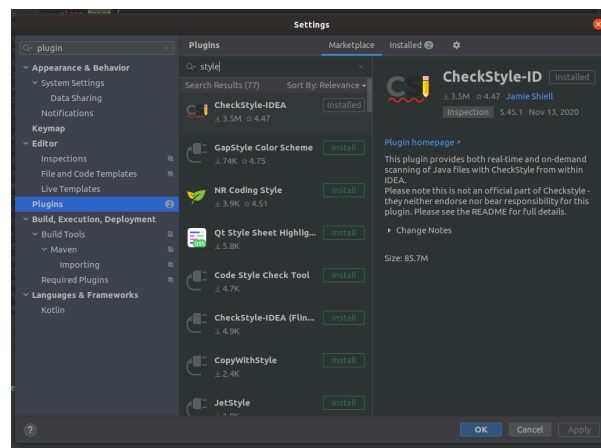
RuboCop



Java · Ruby · Python

Automatically reformat your existing code

Developer time is valuable!



Formatter plugins (e.g. CheckStyle-IDEA) and standalone tools (e.g. Black) can rewrite existing code to match a style guide mechanically, so you don't have to fix every file by hand.

Style is an easy way to improve readability

- Everyone has their own opinion (e.g., tabs vs. spaces).
- Agree to a convention and stick to it.
Use continuous integration to enforce it.
- Use automated tools to fix issues in existing code.

Pattern-Based Linters

Catching common mistakes automatically

Pattern-based linters group checks into categories

Bad Practice

Correctness

Performance

Internationalization

Malicious Code

Multithreaded Correctness

Security

Dodgy Code



FindBugs	FindBugs Bug Descriptions
	This document lists the standard bug patterns reported by FindBugs version 3.0.1.
Summary	Description
Issues and Info	Category
FindBugs 1.0	BC, Equals method should not assume anything about the type of its argument
FindBugs 1.0	B2, Class has static instance of superclass
FindBugs 1.0	CN, Class implements Comparable but does not define or use clone method
FindBugs 1.0	CS, clone method does not call super.clone()
FindBugs 1.0	CN, Class defines clone() but doesn't implement Comparable
FindBugs 1.0	C2, Bad use of throw statement
FindBugs 1.0	CN, Abstract class implements concrete, concrete's method is abstract
FindBugs 1.0	CS, Comparable compareTo incorrectly handles NaN, Double value
FindBugs 1.0	CS, Comparable compareTo incorrectly handles NaN, Float value
FindBugs 1.0	CN, Comparable compareTo() method defined
FindBugs 1.0	HE, Method might throw exception
FindBugs 1.0	CN, Method might throw exception
FindBugs 1.0	DM, Use of equals() method of an array class is dangerous due to reuse of array objects
FindBugs 1.0	DM, Random number created and used only once
FindBugs 1.0	DM, Random number is responsible for an array class's behavior
FindBugs 1.0	DM, Random invokes System.exit()
FindBugs 1.0	DM, Method not responsible for an array class's behavior
FindBugs 1.0	CS, Comparison of String parameter array not as in JLS
FindBugs 1.0	ES, Comparison of string objects using == or !=
FindBugs 1.0	ES, Abstract Class defines compareTo() but doesn't implement Comparable
FindBugs 1.0	HE, Equals checks for null, implements operator
FindBugs 1.0	HE, Equals checks for null, implements operator
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable
FindBugs 1.0	ES, Equals, method fails for Comparable

FindBugs / SpotBugs organize their rules into categories like these.

Pattern-Based Analysis evaluates program syntax against a set of rules

Matches syntactic patterns (via the abstract syntax tree) to identify likely mistakes and API misuses.

- Good at finding disallowed or deprecated APIs, dangerous language features, and obvious mistakes.
 - Provides fast, **best-effort** bug finding when used appropriately.
 - Saves time during code review by checking for common mistakes.
- Can only find issues for which there is a corresponding rule or pattern.
 - Some issues may incorrectly trigger in benign cases (false positives).

Example: Bad Practice

```
String x = new String("Foo");
String y = new String("Foo");

if (x == y) {
    System.out.println("x and y are the same!");
} else {
    System.out.println("x and y are different!");
}
```

ES_COMPARING_STRINGS_WITH_EQ : comparing strings with ==

Fix: compare with `x.equals(y)` instead.

NOTES

It prints "different" even though the contents are equal, because == compares object identity for `new String(...)`.

Example: Performance

```
public static String repeat(String string, int times) {
    String output = string;
    for (int i = 1; i < times; ++i) {
        output = output + string;
    }
    return output;
}
```

SBSC_USE_STRINGBUFFER_CONCATENATION: method concatenates strings using + in a loop

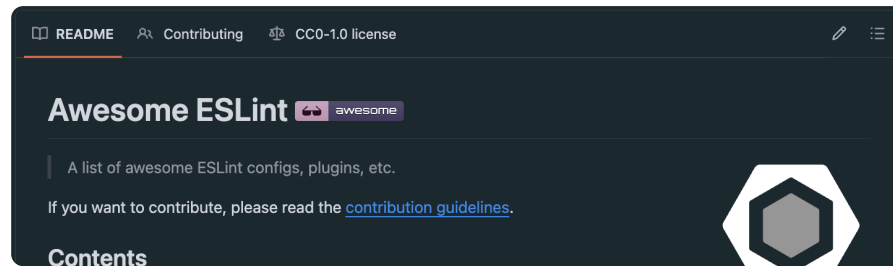
```
public static String repeat(String string, int times) {
    StringBuffer output = new StringBuffer(string);
    for (int i = 1; i < times; ++i) {
        output.append(string);
    }
    return output.toString();
}
```

NOTES

Each `+` on Strings allocates a new String; a StringBuffer grows an internal buffer instead.

ESLint can be extended with plugins

- To find more code quality issues: `eslint-plugin-depend`, SonarJS, Unicorn, ...
- To scan different languages: SQL, HTML, JSON, YAML, ...
- To identify issues with frameworks: React, Angular, Vue, ...
- To identify issues with libraries: JSDoc, jQuery, RequireJS, ...



The same plugin-ecosystem pattern shows up around most pattern-based linters, not just ESLint.

Outline

- Why static analysis?
- What can we determine without running code?
- Formatting and pattern-based linters
- **Type annotations and taint analysis**
- Making analysis useful in practice
- Takeaways

Type Annotations

Catching type errors before the code ever runs

Some type mistakes only appear at runtime

No editor warnings
in JavaScript files

This code crashes at
runtime!

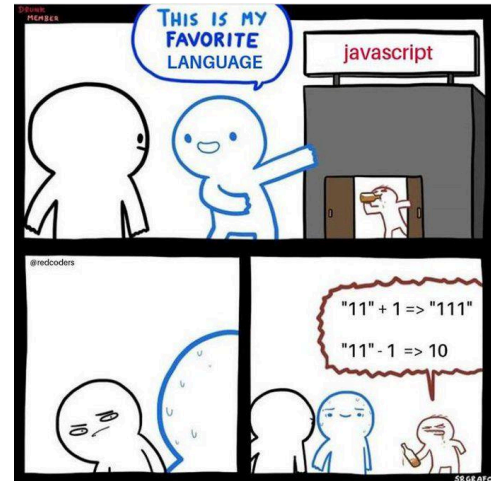
```
function compact(arr) {  
  if (orr.length > 10)  
    return arr.trim(0, 10)  
  return arr  
}
```

This code crashes at runtime: `orr` is never defined.

JavaScript can produce surprising results through type conversion.

- JavaScript allows a variable to hold values of different types.
- Some operations automatically convert values between types.
- These conversions can produce unexpected results without raising an error.

```
"11" + 1 // "111": string concatenation  
"11" - 1 // 10: numeric subtraction
```



TypeScript: JavaScript with Types

- TypeScript is a strongly typed language: errors are caught before run-time.
- TypeScript is converted (“transpiled”) to JavaScript.

 TypeScript adds natural syntax for providing types

```
function compact(arr: string[]) {  
    if (arr.length > 10)  
        return arr.slice(0, 10)  
    return arr  
}
```

Add Types to Existing Code via Annotations

- Add type annotations on top of the existing language.
- Allows you to mix and match typed and untyped code: easier to transition incrementally.

```
/**  
 * JSDoc  
 */  
// @ts-check  
/** @param {any[]} arr */  
function compact(arr) {  
  if (arr.length > 10)  
    return arr.trim(0, 10)  
  return arr  
}
```

Using JSDoc to give type information

Property 'trim' does not exist on type 'any[]'.

Now TS has found a bad call. Arrays have slice, not trim.

Use type annotations to detect common errors

- Uses a conservative analysis to prove the **absence** of certain defects:
Unsanitized input, null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings...
- Assuming that code is annotated and those annotations are correct.
- Use annotations to enhance the type system.

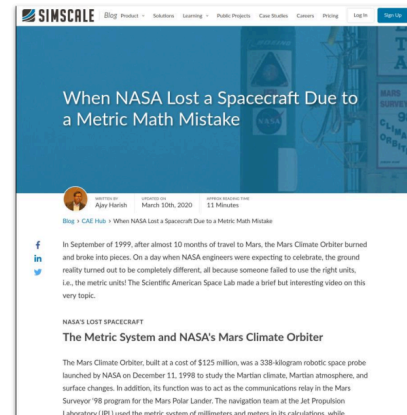
NOTES

We have seen checks for suspicious code patterns. Now we will track properties of values, such as their units or whether they came from user input.

Remember the Mars Climate Orbiter incident from 1999?



Remember the Mars Climate Orbiter incident from 1999?



NASA lost a \$125M spacecraft because one navigation team used pound-force seconds while the other assumed newton-seconds, a unit mismatch that an uncaught type error would have avoided.

Does this program compile? No

```
void demo() {  
    @m int x;  
    x = 5 * m;  
  
    @m int meters = 5 * m;  
    @s int seconds = 2 * s;  
  
    @mPERs int speed = meters / seconds;  
    @m int foo = meters + seconds;    // rejected  
    @s int bar = seconds - meters;    // rejected  
}
```

@m indicates that a value represents meters. To assign a unit, multiply by the appropriate unit constant.

Limitations of Type-Based Static Analysis

- Can only analyze code that is **annotated**.

Requires that dependent libraries are also annotated; can be tricky to retrofit annotations into existing codebases.

- Only considers the **signature and annotations** of methods.

Doesn't look at the implementation of methods that are being called.

- Can't handle **dynamically generated** code well.

Examples: Spring Framework, templates.

- Can produce **false positives!**

A byproduct of necessary approximations.

Taint Analysis

Tracking untrusted data from source to sink

Taint Checking using Annotations

```
void processRequest() {  
    @Tainted String input =  
        getUserInput();  
    executeQuery(input);  
}  
  
public void executeQuery(  
    @Untainted String input) { ... }  
  
@Untainted  
public String validate(  
    String userInput) { ... }
```



`@Tainted` marks data that came from outside the program. An `@Untainted` parameter must never receive tainted data; an `@Untainted` return type guarantees its result is safe.

"Malicious" user input, e.g. a value containing SQL, is exactly the kind of data taint analysis tracks from a source to a sensitive sink.

Activity: Does this code pass the taint checker?

```
void processRequest() {  
    @Tainted String input = getUserInput();  
    if (input.contains("drop tables")) {  
        input = validate(input);  
    }  
    executeQuery(input); // requires @Untainted  
}
```

Assume `validate` returns `@Untainted` data.

1. Should the checker accept this code? Why?
2. What is the smallest change that would make it pass?

2 minutes, in pairs.

Debrief: Does this code pass the taint checker?

```
void processRequest() {  
    @Tainted String input = getUserInput();  
    if (input.contains("drop tables")) {  
        input = validate(input);  
    }  
    executeQuery(input); // requires @Untainted  
}
```

No: the checker rejects this.

- `input` is only reassigned to the validated, `@Untainted` result inside the `if` branch.
- When the condition is false, the original `@Tainted` value flows straight to `executeQuery`.
- The substring check for `"drop tables"` isn't a real defense: it only catches that one literal phrase.

Smallest fix: call `validate` unconditionally before `executeQuery`, removing the `if` entirely.

Does this program compile? Yes

```
void processRequest() {  
    @Tainted String input = getUserInput();  
    input = validate(input);  
    executeQuery(input);  
}  
public void executeQuery(  
    @Untainted String input) { ... }
```

Calling `validate` unconditionally removes the unsafe path. Every path now reaches `executeQuery` only after validation.

The checker needs to establish that the input is safe on every path reaching the query.

Outline

- Why static analysis?
- What can we determine without running code?
- Formatting and pattern-based linters
- Type annotations and taint analysis
- **Making analysis useful in practice**
- Takeaways

What makes a good static analysis tool?

- Static analysis should be **fast**.

Don't hold up development velocity; this becomes more important as code scales.

- Static analysis should report **few false positives**.

Otherwise developers will start to ignore warnings and alerts, and quality will decline.

- Static analysis should be **continuous**.

Part of your CI pipeline; diff-based analysis is even better: don't analyze the entire codebase, just the changes.

- Static analysis should be **informative**.

Messages that help the developer quickly locate and address the issue; ideally, it should suggest or automatically apply fixes.

When an analyzer cannot determine whether code is safe, should it report a warning?

NOTES

An analyzer facing an undecidable question must choose between over-approximating (more false positives) and under-approximating (more false negatives). That tradeoff is exactly why "few false positives" is a real engineering requirement.

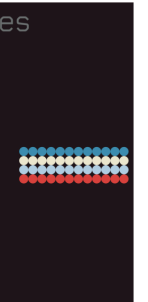
Lessons for Static Analysis Tools at Google

- Make it a compiler workflow; value compiler checks.
 - Reporting issues sooner is better.
 - Warn during code review.
 - Engineers building static analysis must demonstrate impact through hard data.
- Finding bugs is easy; most developers will not go out of their way to use a tool.
 - Developer happiness is key.
 - Do not just find bugs. Fix them.
 - Crowdsource analysis development.

contributed articles

DOI:10.1145/1380720
For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.
BY CAITLIN SADOWSKI, EDWARD AFTANDILIAN, ALEX EAGLE, LIAM MILLER-CUSHON, AND CHERA JASPER

Lessons from Building Static Analysis Tools at Google



Sadowski, Aftandilian, Eagle, Miller-Cushon, Jaspan. "Lessons from Building Static Analysis Tools at Google."

Reasons engineers do not always use static analysis tools or ignore their warnings

- **Not integrated.** The tool isn't in the developer's workflow, or takes too long to run.
- **Not actionable.** The error should include a suggested fix that can be applied mechanically.
- **Not trustworthy.** Users don't trust the results.
- **Not manifest in practice.** The bug is theoretically possible, but doesn't actually occur.
- **Too expensive to fix.** Fixing the detected bug is too expensive or risky.
- **Warnings not understood.**

Which of these would make your team ignore a warning?

Key Takeaways

How is this different to using AI tools?

- Static analysis is driven by a set of **deterministic rules**.

We can confidently apply them and obtain stronger assurances.

- LLMs are **probabilistic**.

We can't repeat results; some results will be catastrophically incorrect. But LLMs are potentially richer and more expressive, and patterns are implicitly captured in the latent space.

- It makes sense to use both in **different contexts**.

CI: static analysis. PRs: AI-provided suggestions and draft changes.

Which tool to use?

Depends on use case and available resources.

- **Formatters:** fast, cheap, easy to address issues or set ignore rules.
- **Pattern-based linters:** intuitive, but need to deal with false positives.
- **Type-annotation-based checkers:** more manual effort required; needs overall project commitment, but good payoff once adopted.
- **Deep analysis tools:** can find tricky issues, but can be costly; might need some awareness of the analysis to deal with false positives.

The best QA strategy involves multiple analysis, testing, and inspection techniques!

What you need to know

Static analysis checks code without executing it.

A warning may need investigation; no warnings does not necessarily mean no bugs.

Different checks detect different classes of problems.

Automate useful checks in the IDE and CI, with actionable feedback.