Shifting Left with Static Analysis

17-313: Foundations of Software Engineering

https://cmu-313.github.io

Michael Hilton and **Josh Sunshine**

Spring 2026

Smoking Section

- Last full row

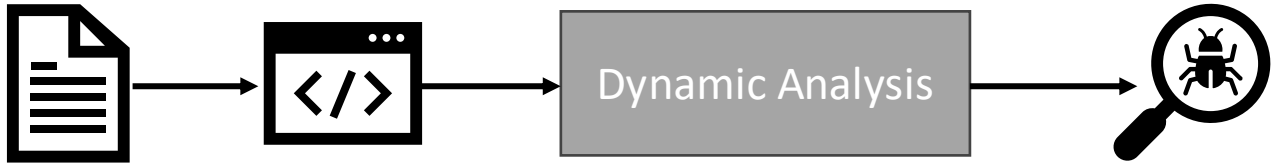
Course Announcements

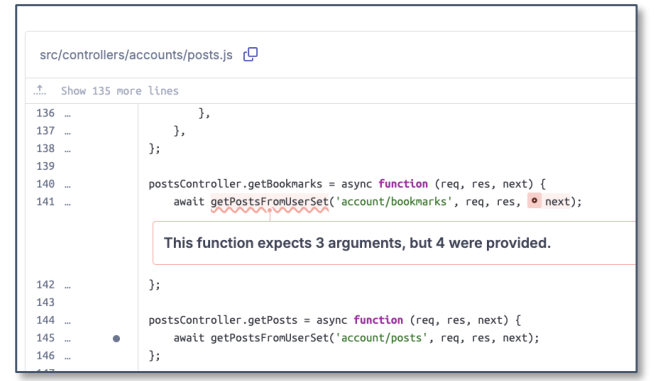
- Midterm review session
  - Monday, February 23 @ 7pm
  - GHC 4401 (Rashid)
- Midterm
  - Thursday, February 26
  - If you need a disability accommodation, please schedule with ODR immediately.
- Project 2C due ~~Thursday, February 26~~ Friday, February 27

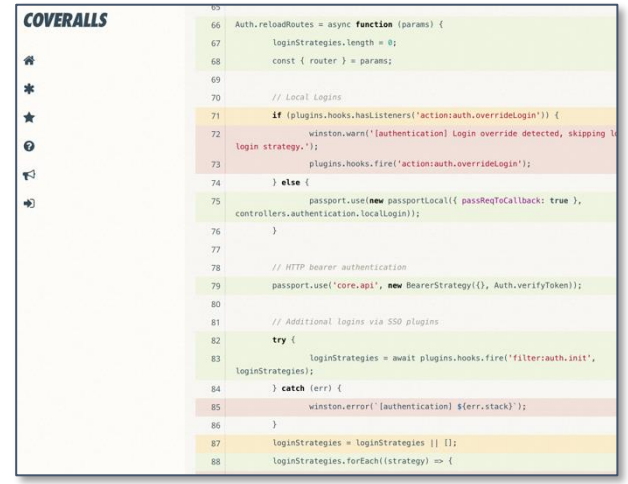
Learning Goals

- Understand static vs. dynamic analysis
- Recognize the strengths and limitations of static analysis
- Understand how static analysis is used to shift left and build systems faster and more confidently
- Explore techniques from linters to deep analyzers

What are Program Analysis Tools?







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`?
- 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? be prepared for “MAYBE”
- Verifying **advanced properties** is possible but expensive
  - CI-based static analysis usually over-approximates conservatively

The Bad News: Rice's Theorem
Every static analysis is necessarily incomplete, unsound, undecidable, or a combination thereof

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

Henry Gordon Rice, 1953

Static Analysis is well suited to detecting certain kinds of defect

- **Security:** Buffer overruns, improperly validated input …
- **Memory safety:** Null dereference, uninitialized data …
- **Resource leaks:** Memory, OS resources …

- These often rely on specific conditions and take place over long horizons (e.g., leaks). Difficult to find using traditional testing!

Dynamic analysis reasons about executions

- Tells you properties of the program that were **definitely observed**
  - Code coverage
  - Performance profiling
  - Type profiling
  - Testing

- In practice, implemented by program instrumentation
  - Think “Automated logging”
  - Slows down execution speed by a small amount

Static analysis has many applications

- Find bugs
- Refactor code
- Keep your code stylish!
- Identify code smells
- Measure quality
- Find usability and accessibility issues
- Identify bottlenecks and improve performance

Static Analysis vs. Dynamic Analysis

- Requires only source code
- Conservatively reasons about all possible inputs and program paths
- Reported warnings may contain false positives
- Can report all warnings of a particular class of problems
- Advanced techniques like formal verification can prove certain complex properties, but rarely run in CI due to cost

- Requires successful build + test inputs
- Observes individual executions
- Reported problems are real, as observed by a witness input
- Can only report problems that are seen. Highly dependent on test inputs. Subject to false negatives
- Advanced techniques like symbolic execution can prove certain complex properties, but rarely run in CI due to cost

Activity: Analyze the Python program dynamically

```
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

print(n2s(12, 10))
```

1. What is the type of variable `u` during program execution?
2. Did the variable `u` ever contain a negative number?
3. For how many iterations did the while loop execute?
4. Was there ever be a division by zero?
5. Did the returned value ever contain a minus sign ‘-’?

Activity: Analyze the Python program statically

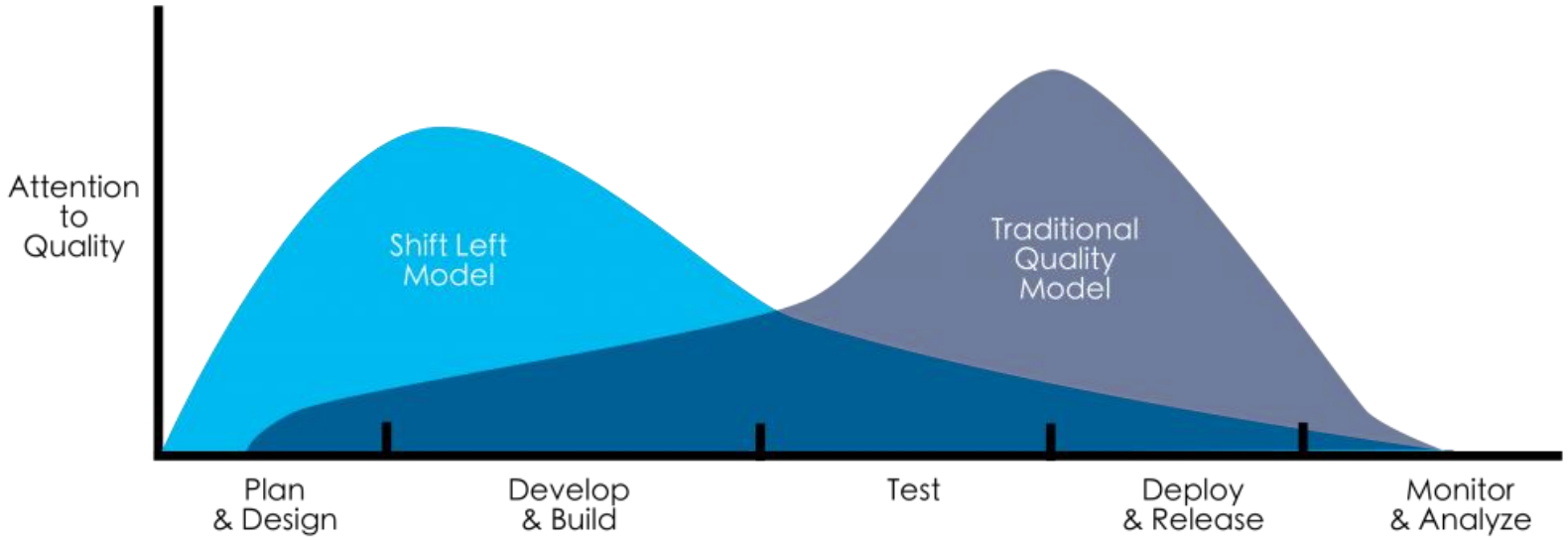
```
def n2s(n: int, b: int):
 if n <= 0: return '0'
 r = ''
 while n > 0:
  u = n % b
  if u >= 10:
   u = chr(ord('A') + u-10)
  n = n // b
  r = str(u) + r
 return r
```

1. What is the type of variable `u`?
2. Will the variable `u` be a negative number?
3. Will this function always return a value?
4. Will the program divide by zero?
5. Will the returned value ever contain a minus sign ‘-’?

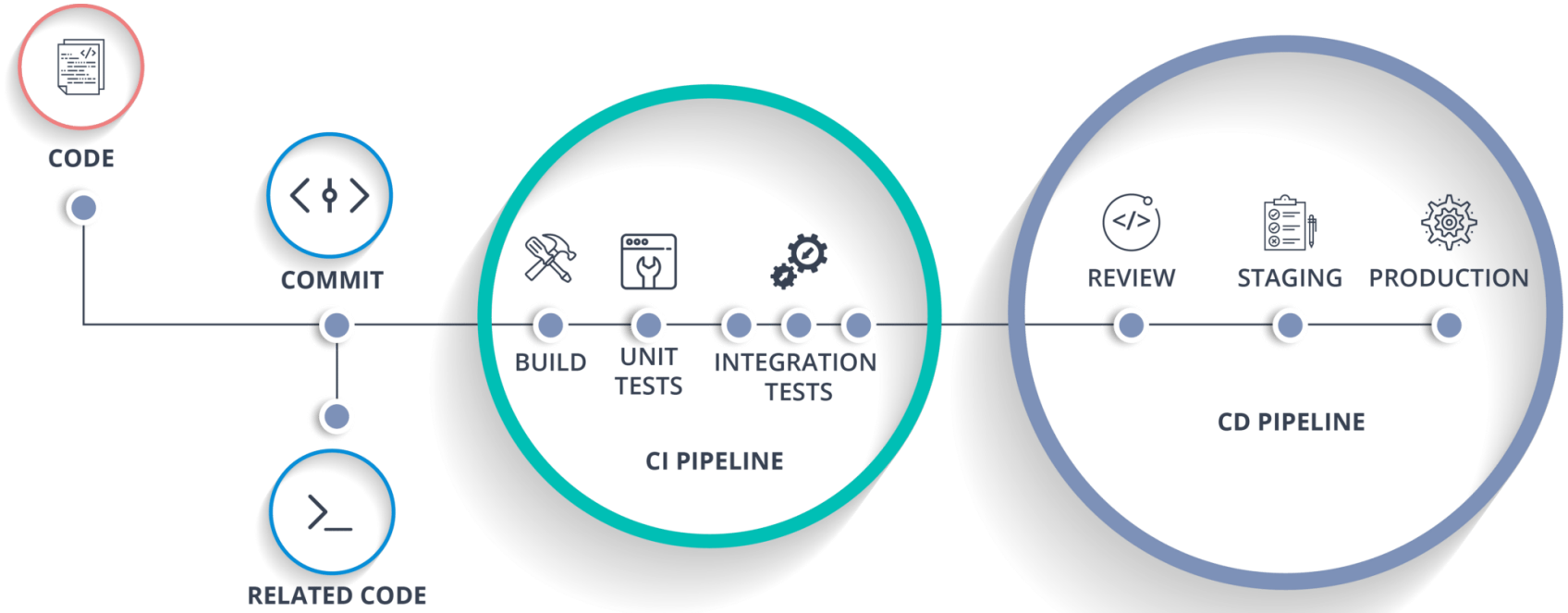
Static Analysis

Static Analysis is Key to Shifting Left

- Issues are cheaper and faster to rectify when discovered early
- → **Find and prevent issues as early as possible**

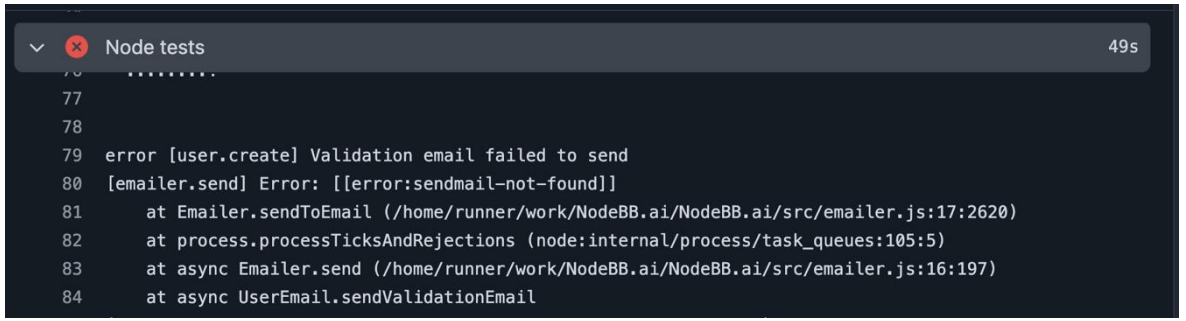


Static Analysis is a key part of Continuous Integration

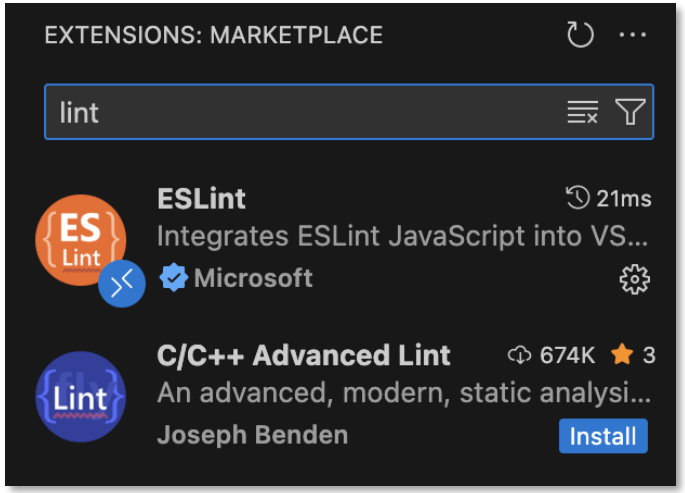


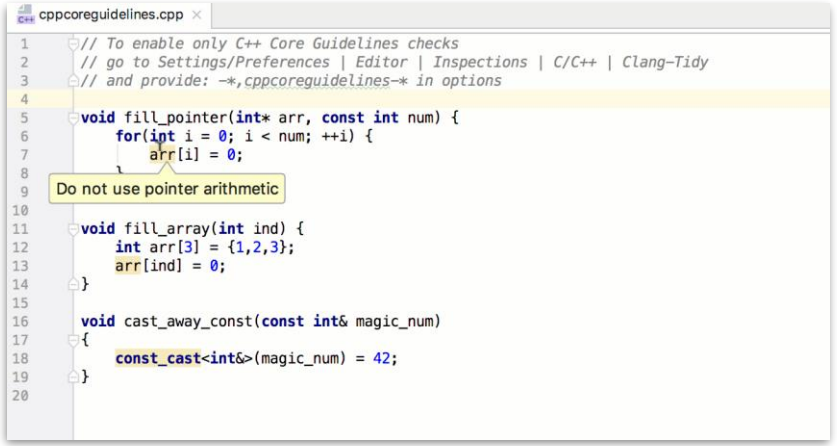
Reflecting on NodeBB

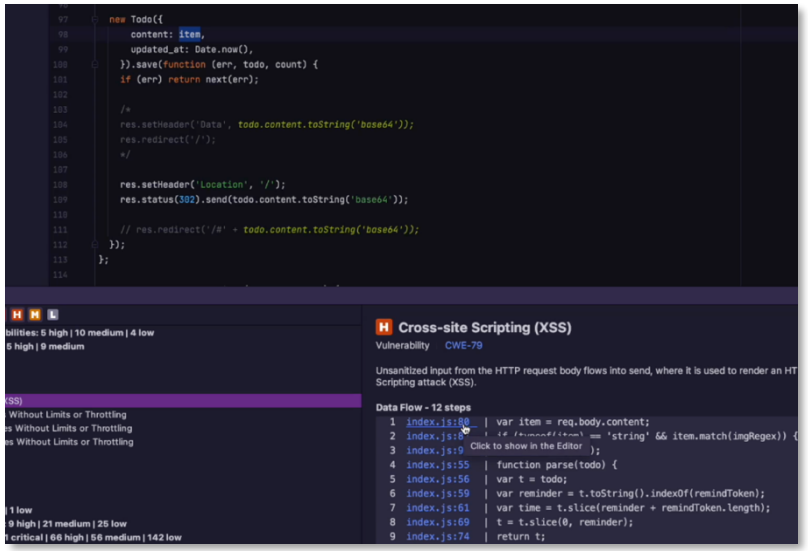
- Did your team **accidentally merge breaking changes?**
  - e.g., missing semi-colons, incorrect variable names, …
- How did it sneak past review?
  - you probably weren't expecting to look for small mistakes!
  - you didn't get much support from the CI setup
  - you might have been distracted by the flaky tests!



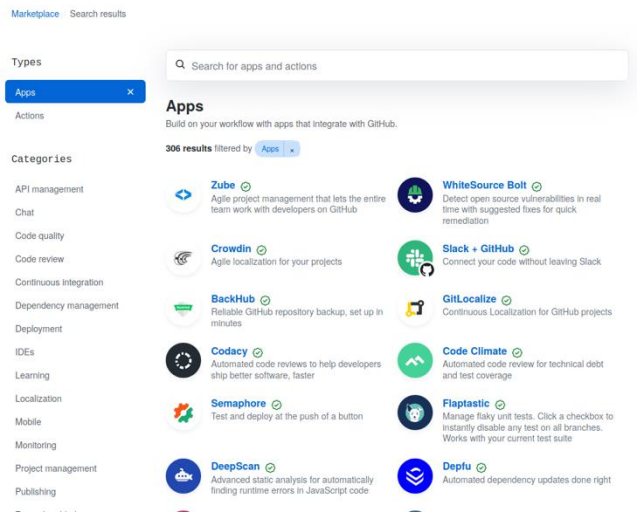
Static analysis is integrated in your IDE

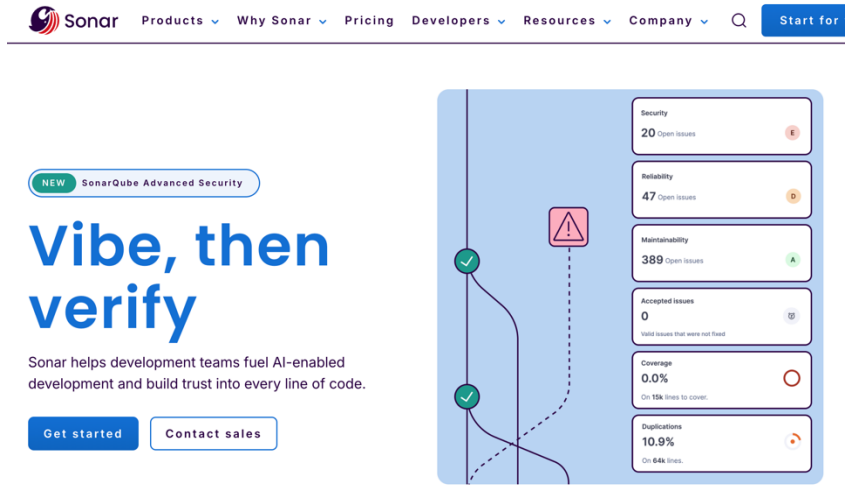


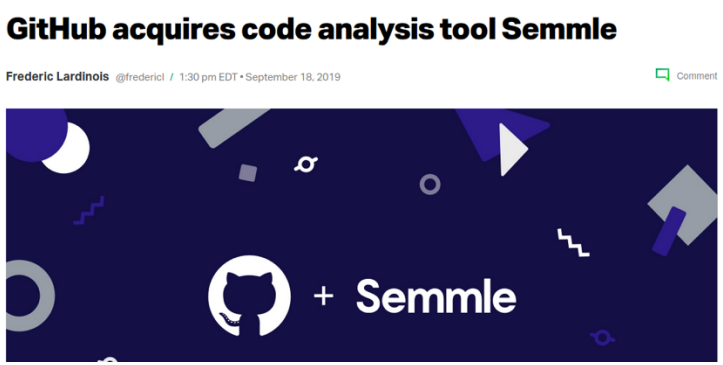




Static analysis used to be an academic amusement. Now it's heavily commercialized.







sonarqube

snyk

GitHub Copilot

There are lots of static analysis tools!

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**
  - Or developers will start to ignore warnings and alerts, and quality will decline
- Static analysis should be **continuous**
  - Should be part of your continuous integration pipeline
  - Even better: don’t analyze the whole codebase; just the changes
- Static analysis should be **informative**
  - Messages that help the developer to quickly locate and address the issue
  - Ideally, it should suggest or automatically apply fixes

https://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext

Static Analysis: Broad Classification

- 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
  - Check conformance to user-defined types
  - Types can be complex (e.g., “Nullable”)
- Data-Flow Analysis / Abstract Interpretation (Value Analysis)
  - Deep program analysis to find complex error conditions
  - e.g., ”can array index be out of bounds?”

Today

- Formatting Linters
- Pattern-Based Linters
- Type-Based Analysis
- Value Analysis (Data Flow & Abstract Interpretation)
- Analysis for Everything Else

Today

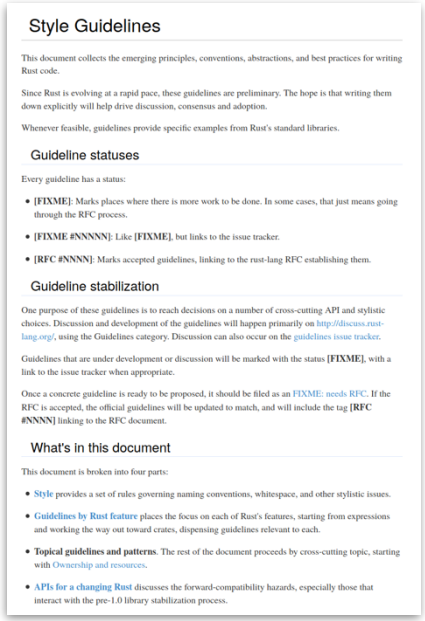
- **Formatting Linters**
- Pattern-Based Linters
- Type-Based Analysis
- Value Analysis (Data Flow & Abstract Interpretation)
- Analysis for Everything Else

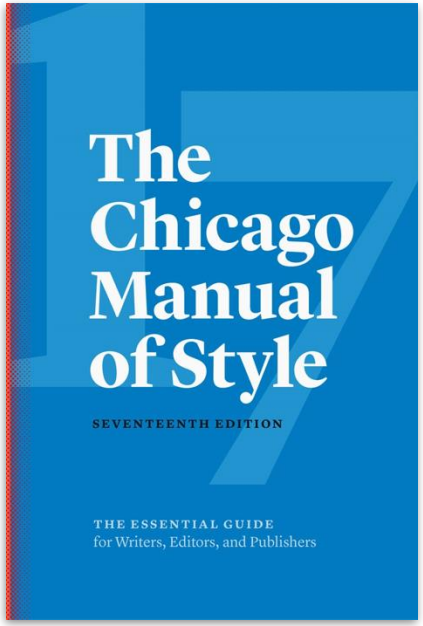
Linters: *Cheap, fast, and lightweight static source analysis*

Formatting Linters use shallow static analysis to enforce formatting rules

- Ensure proper indentation
- Naming convention
- Line sizes
- Class nesting
- Documenting public functions
- Parenthesis around expressions
- What else?

Style guidelines help to facilitate communication





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

https://www.chicagomanualofstyle.org/ | https://google.github.io/styleguide/ | https://www.python.org/dev/peps/pep-0008 | https://github.com/airbnb/javascript

Use linters to enforce style guidelines

Don’t rely on manual inspection during code review! Even better, automatically apply the tool on save or commit.

Use linters to improve maintainability

- Why? **We spend more time reading code than writing it**
  - Various estimates of the exact %, some as high as 80%
- Code is 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

Today

- Formatting Linters
- **Pattern-Based Linters**
- Type-Based Analysis
- Value Analysis (Data Flow & Abstract Interpretation)
- Analysis for Everything Else

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

- Matches **syntactic patterns** (via abstract syntax tree) to identify likely mistakes and API misuses
  - Good at finding use of disallowed and deprecated APIs, dangerous language features, and obvious mistakes
- Provides fast, **best effort** bug finding when used appropriately
  - Can only find issues for which there is a corresponding rule / pattern
  - Some issues may incorrectly trigger in benign cases (false positives)
  - Saves time during code review by checking for common mistakes

Pattern-Based Analysis for JS/TS

- De facto standard for pattern-based checks in JavaScript and TypeScript. Integrates with editors (e.g., VS Code) out of the box
  - “npm run lint” usually involves ESLint
- Provides **rules** that check for mistakes and enforce best practices
  - Correctness Rules (“Possible Problems”) look for logic errors
  - Suggestion Rules enforce best practices and clean code
- Automatically fixes the code for certain rule violations (--fix)
  - by applying a deterministic, syntactic rewrite rule (no LLMs!)

What's the problem in this code?

```
setTimeout("doThing()", 100);
setInterval("x = x + 1", 1000);
setInterval(callbackStr, 500);
const f = new Function("a", "b", "return a + b");
```

Correctness Rule: no-implied-eval

- Identifies implicit evaluation of strings as code
  - equivalent to eval — a major security and reliability risk!
  - stringified code escapes static analysis; may crash or cause problems
  - user-provided strings open up the potential for remote code execution

```
// Bad: string evaluated as code
setTimeout("doThing()", 100);
setInterval("x = x + 1", 1000);
const f = new Function("a", "b", "return a + b");
```

```
// Good: pass functions/closures
setTimeout(() => doThing(), 100);
setInterval(() => { x = x + 1; }, 1000);
function add(a, b) { return a + b; }
```

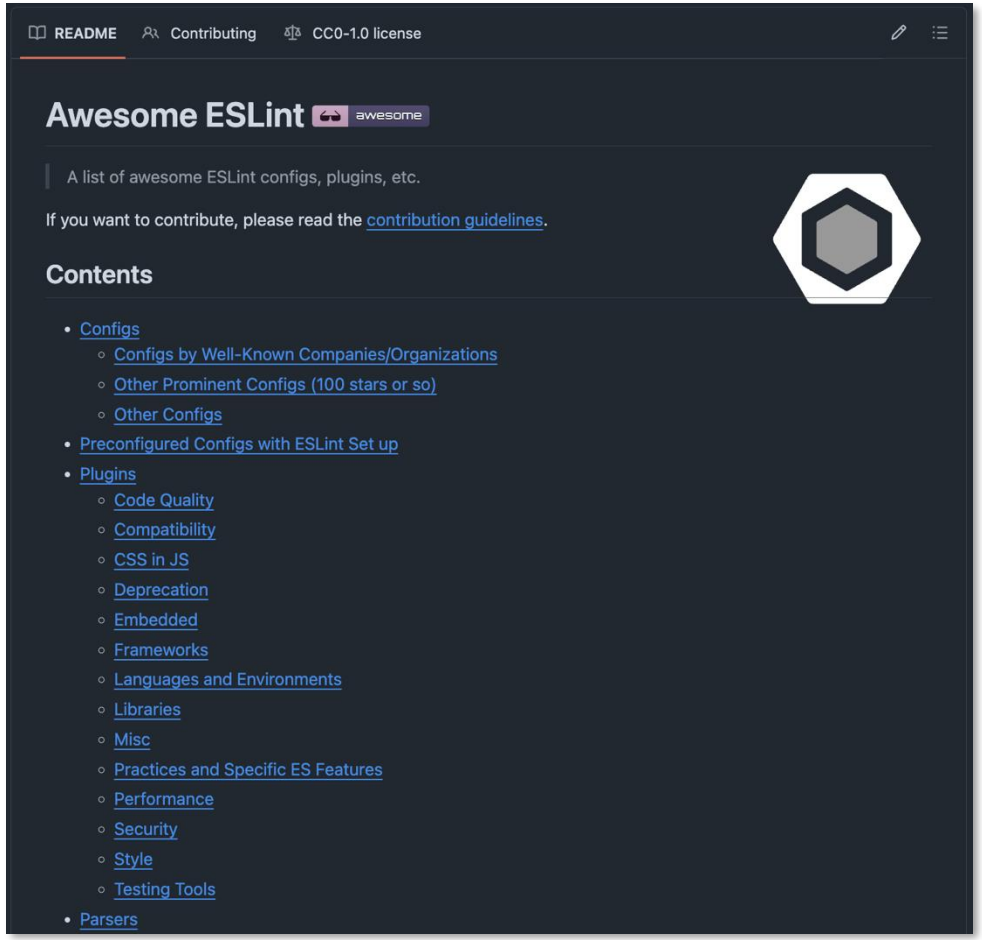
Suggested Rule: (no) “Yoda”

- Yoda condition flip operands
  - Pro Yoda: it’s impossible to accidentally use “=“
  - Anti Yoda: it makes the code harder to read

```
// ❌ Yoda style
if (“red” === color) { /* … */ }

// ✅ Preferred
if (color === “red”) { /* … */ }
```

ESLint can be extended with plugins

- To find more code quality issues
  - 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, …



Challenges with pattern-based analysis

- The analysis must produce few or (better yet) zero false positives
  - Otherwise, developers won't be able to build the code!
- The analysis needs to be really fast
  - Ideally < 100 ms
  - If it takes longer, developers will become irritated and lose productivity
  - Practically, this means the analysis needs to focus on "shallow" bugs rather than verifying some complex logic spanning multiple functions/classes
- You can't just "turn on" a particular check
  - Every instance where that check fails will prevent existing code from building
  - There could be thousands of violations for a single check across large codebases

Today

- Formatting Linters
- Pattern-Based Linters
- **Type-Based Analysis**
- Value Analysis (Data Flow & Abstract Interpretation)
- Analysis for Everything Else

Can you spot the bug?

```
// $ ./prog 5 helloWorld
// hello
int main(int c, char **v) { // prints first N characters of string
    if (c < 3) return 1;
    int n = atoi(v[1]);
    char buf[8];
    if (n < sizeof buf) {
       memcpy(buf, v[2], n);
    }
    buf[n] = '\0';
    puts(buf);
}
```

Can you spot the bug?

```
// $ ./prog 5 helloWorld
// hello
int main(int c, char **v) { // prints first N characters of string
    if (c < 3) return 1;
    int n = atoi(v[1]);
    char buf[8];
    if (n < sizeof buf) {          // negative values are allowed
       memcpy(buf, v[2], n);       // n is promoted to size_t; becomes huge number!
    }
    buf[n] = '\0';
    puts(buf);
}
```

Can you spot the bug?

```
// $ ./prog 5 helloWorld
// hello
int main(int c, char **v) { // prints first N characters of string
    if (c < 3) return 1;
    int n = atoi(v[1]);
    char buf[8];
    if (n < sizeof buf) {
       memcpy(buf, v[2], n);
    }
    buf[n] = '\0';                  // undefined behavior for n < 0!
    puts(buf);
}
```

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

Written by **Catalin Cimpanu,** Contributor

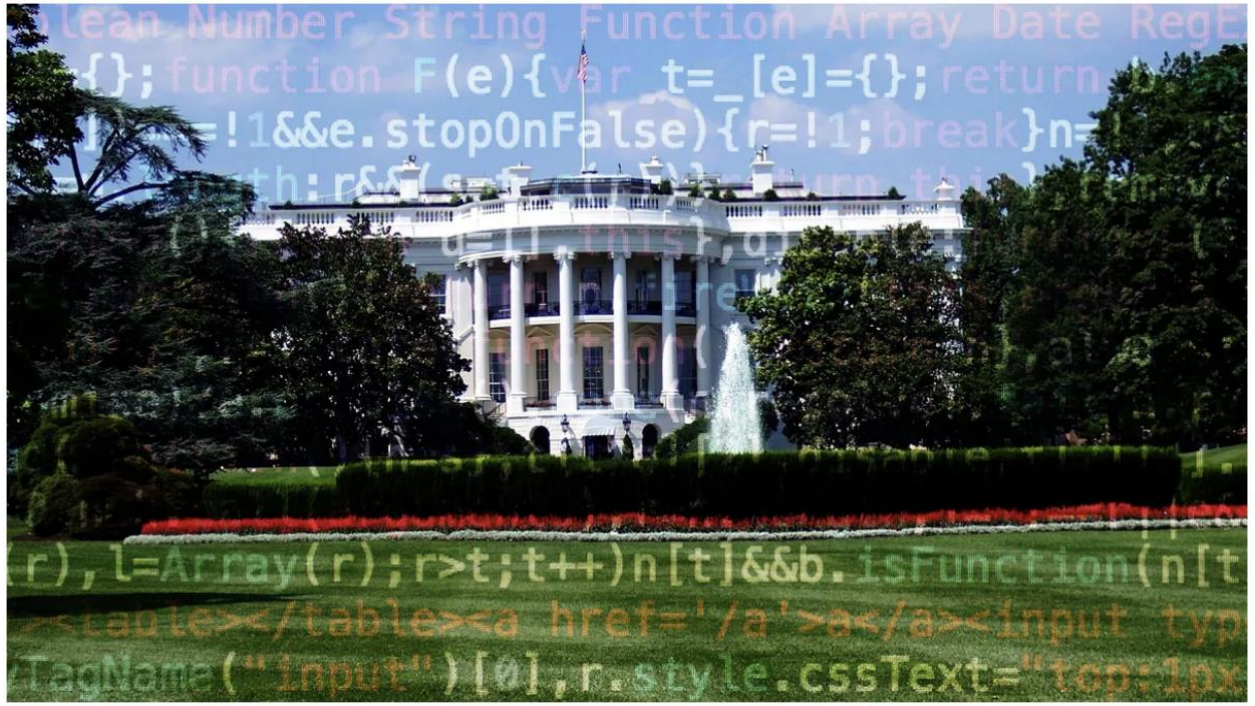
Feb. 11, 2019 at 7:48 a.m. PT

ZD NET

White House urges developers to avoid C and C++, use 'memory-safe' programming languages

News By Les Pounder published 28 February 2024

The languages may pose a security risk when used in critical systems.



Languages as the first line of defense

- **Idea: Prevent entire classes of bugs before runtime!**
  - bad programs won't compile or fail checks; errors surface in editor / CI
  - provides strong guarantees about absence of certain bugs
- **Languages provide *memory safety* in different ways**
  - **Compile time (no GC): Rust.** Language features (ownership, borrowing, lifetimes) prevent memory errors in safe code
  - **Managed runtimes:** E.g., JavaScript, Java, C#, Go. Relies on array bounds + garbage collection. Doesn't allow pointer arithmetic
  - **C++ with discipline:** RAII & smart pointers help to reduce leaks and eliminate use-after-free, but they are not memory safe

Can you spot the issue?

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

Memory-safe doesn't imply type safety

No editor warnings in JavaScript files

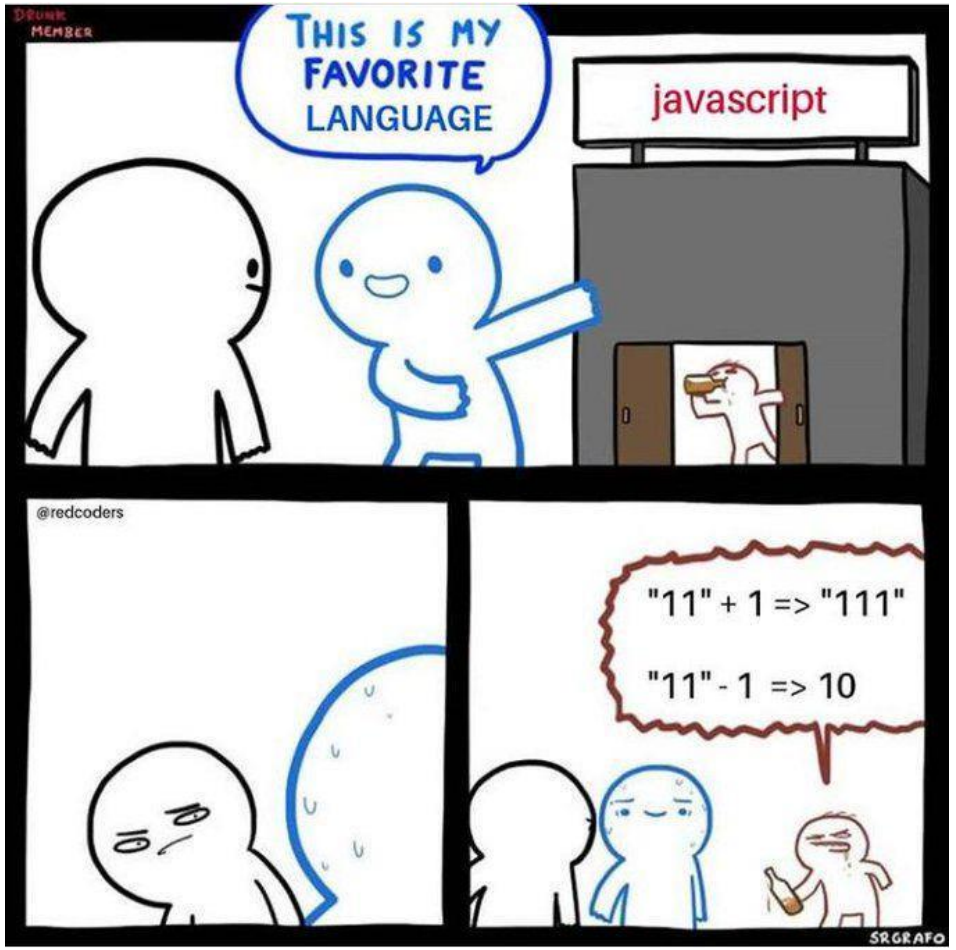
This code crashes at runtime!

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

‼ Cannot find name 'orr'.

Memory-safe doesn't imply type safety

- **Javascript is dynamically and loosely typed language**
- Types are determined at runtime
  - the same variable may hold values with different types over time
- ⚠ Type errors only show up when you run the code
  - uses aggressive type coercion to convert values for compatibility



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 mix and match typed and untyped code -- easier to transition

Using JSDoc to give type information

```
/**
 * JSDoc
 */
```

```
// @ts-check

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

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

    return arr
  }
```

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

Enrich Type Systems via Annotations

- We don’t need to be bound to just **structural types!**
- We can use annotations to layer additional semantics on top of the base type system
  - E.g., Java Checker framework provides annotations that help to target null pointer errors, uninitialized fields, information leaks, SQL injections, incorrect physical units, bad format strings, …
- Can guarantee the absence of certain defect classes
  - provided that code is annotated correctly

Example: Detecting null pointer exceptions

- **@Nullable** indicates that an expression may be null
- **@NonNull** indicates that an expression must never be null
- Guarantees that expressions annotated with **@NonNull** will never evaluate to null. Forbids other expressions from being dereferenced

```
// return value
@NonNull String toString() { ... }

// parameter
int compareTo(@NonNull String other)
{ ... }
```

```
import org.checkerframework.checker.nullness.qual.*;

public class NullnessExampleWithWarnings {
    public void example() {
        @NonNull String foo = "foo";
        String bar = null;
        foo = bar;
        println(foo.length());
    }
}
```

`@Nullable` is applied by default

Error: [assignment.type.incompatible] incompatible types in assignment.
found : @Initialized **@Nullable** String
required: @UnknownInitialization **@NonNull** String

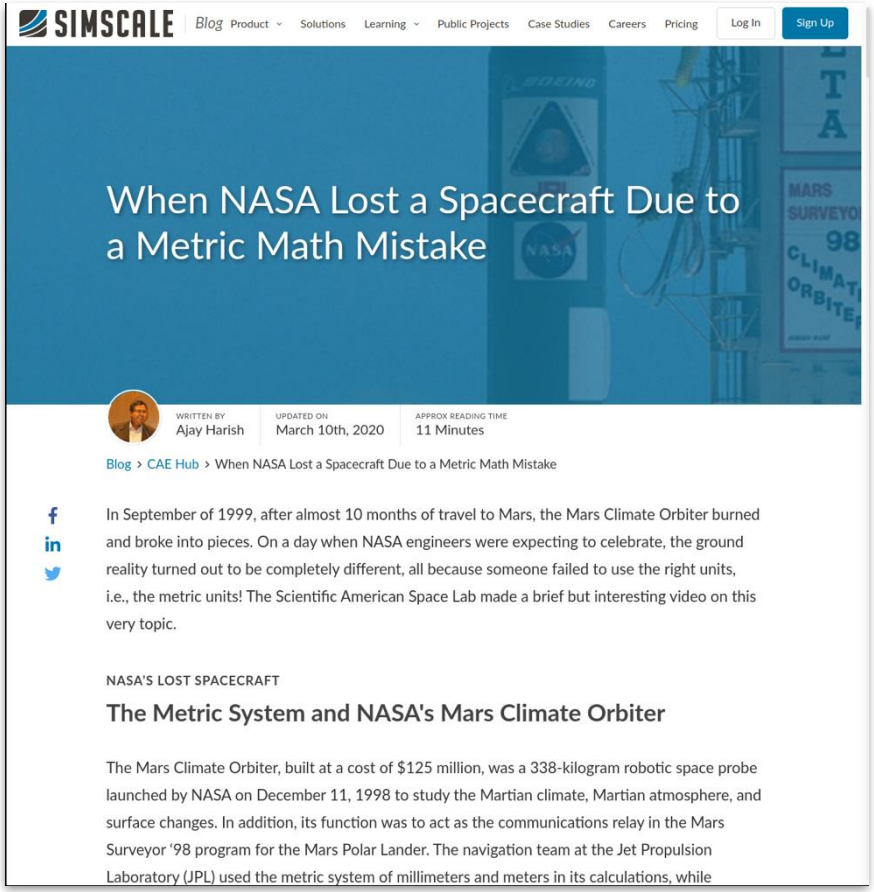
```
import org.checkerframework.checker.nullness.qual.*;

public class NullnessExampleWithWarnings {
  public void example() {
    @NonNull String foo = "foo";
    String bar = null;   // @Nullable
    if (bar != null) {
      foo = bar;
    }
    println(foo.length());
  }
}
```

bar is refined to @NonNull

Another example: Units Checker

- Guarantees operations are physically meaningful and use same kind and units
- Kind annotations
  - @Acceleration, @Angle, @Area, @Current, @Length, @Luminance, @Mass, @Speed, @Substance, @Temperature, @Time
- SI unit annotation
  - @m, @km, @mm, @kg, @mPERs, @mPERs2, @radians, @degrees, @A, …

Remember the Mars Climate Orbiter incident from 1999?



SIMSCALE Blog Product Solutions Learning Public Projects Case Studies Careers Pricing Log In Sign Up

When NASA Lost a Spacecraft Due to a Metric Math Mistake

WRITTEN BY Ajay Harish | UPDATED ON March 10th, 2020 | APPROX READING TIME 11 Minutes

Blog › CAE Hub › When NASA Lost a Spacecraft Due to a Metric Math Mistake

In September of 1999, after almost 10 months of travel to Mars, the Mars Climate Orbiter burned and broke into pieces. On a day when NASA engineers were expecting to celebrate, the ground reality turned out to be completely different, all because someone failed to use the right units, i.e., the metric units! The Scientific American Space Lab made a brief but interesting video on this very topic.

NASA'S LOST SPACECRAFT

The Metric System and NASA's Mars Climate Orbiter

The Mars Climate Orbiter, built at a cost of $125 million, was a 338-kilogram robotic space probe launched by NASA on December 11, 1998 to study the Martian climate, Martian atmosphere, and surface changes. In addition, its function was to act as the communications relay in the Mars Surveyor '98 program for the Mars Polar Lander. The navigation team at the Jet Propulsion Laboratory (JPL) used the metric system of millimeters and meters in its calculations, while

NASA's Mars Climate Orbiter (cost of $327 million) was lost because of a discrepancy between use of metric unit Newtons and imperial measure Pound-force.

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

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;
  @s int bar = seconds - meters;
```

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

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;
  @s int bar = seconds - meters;
```

`@m` indicates that `x` represents meters

To assign a unit, multiply appropriate unit constant from `UnitTools`

Does this program compile?

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

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;
  @s int bar = seconds - meters;
```

`@m` indicates that `x` represents meters

To assign a unit, multiply appropriate unit constant from `UnitTools`

Does this program compile? No.

```
import static org.checkerframework.checker.units.UnitsTools.m;
import static org.checkerframework.checker.units.UnitsTools.mPERs;
import static org.checkerframework.checker.units.UnitsTools.s;

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;
  @s int bar = seconds - meters;
```

Addition and subtraction between meters and seconds is physically meaningless

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!**
  - Byproduct of necessary approximations

Today

- Formatting Linters
- Pattern-Based Linters
- Type-Based Analysis
- **Value Analysis (Data Flow & Abstract Interpretation)**
- Analysis for Everything Else

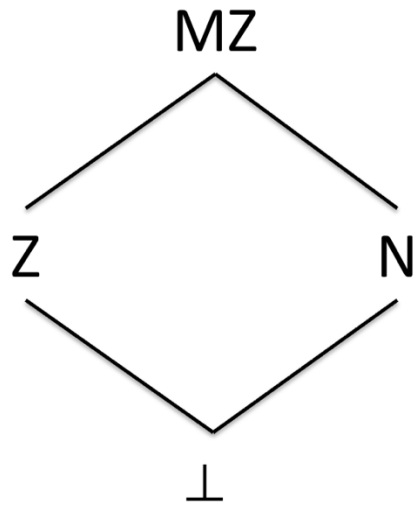
Dataflow and Taint Analysis

- Tracks how values move through a program (assignments, branches, function calls)
  - Can data from an untrusted **source** reach a **sink** along a feasible path?
  - Check if **tainted** data is **sanitized** before reaching sink
- Useful for finding security issues
  - command and SQL injection; cross-site scripting; unsafe deserialization; …
  - requires models of frameworks, libraries, and sanitizers; if these models are missing, results will contain false positives/negatives
  - struggles with aliasing and dynamic features (e.g., eval, reflection)

Abstract Interpretation / Value Analysis

- Computes a **sound over-approximations** of program behavior in terms of an **abstract domain**
  - Goal: determine if a property holds for **all executions**
    - e.g., “y/x” is “x” ever 0?
  - abstract domain captures only the values/states relevant to our property of interest
    - e.g., “is zero?”
- Mostly restricted to embedded, safety critical code
  - not suited to dynamic and reflective languages
  - difficult to scale — explores all possible paths!

frama C
Software Analyzers

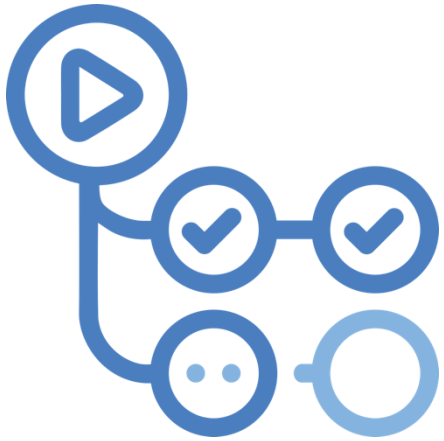


Today

- Formatting Linters
- Pattern-Based Linters
- Type-Based Analysis
- Value Analysis (Data Flow & Abstract Interpretation)
- **Analysis for Everything Else**

Static Analysis for Everything Else

- **Static analysis isn't just for source code**
  - If it's machine readable, we can statically analyze it!



Dependencies & Supply Chain

- Analysis can find dependencies with known vulnerabilities (including transitive deps), malicious packages (e.g., typosquats), and incompatible licenses by scanning manifests and images

grype

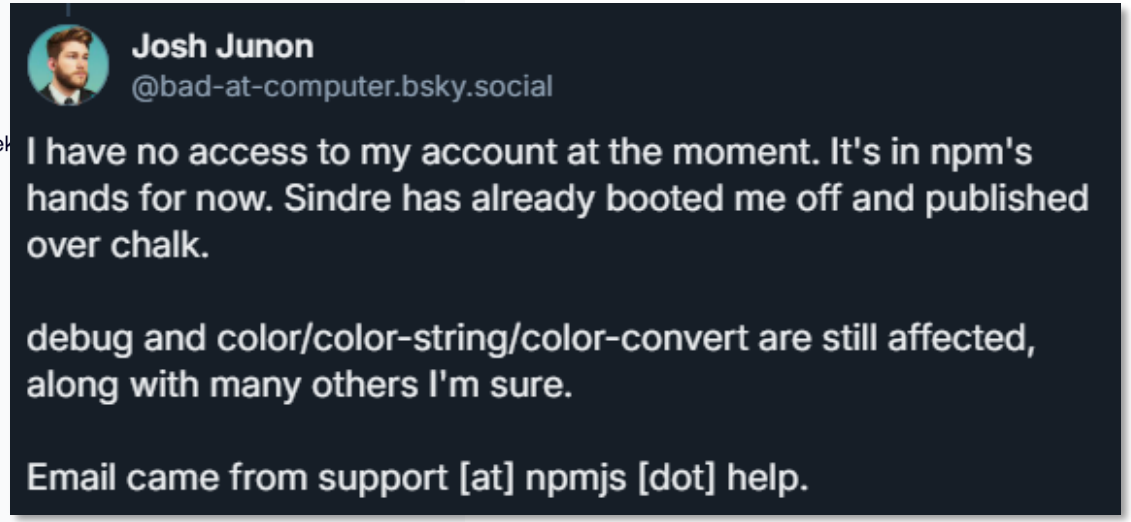
snyk

https://www.stepsecurity.io/blog/ctrl-tinycolor-and-40-npm-packages-compromised

Starting at September 8th, 13:16 UTC, our Aikido intel feed alerted us to a series packages being pushed to npm, which appeared to contains malicious code. These were 18 very popular packages,

- backslash (0.26m downloads per week)
- chalk-template (3.9m downloads per week)
- supports-hyperlinks (19.2m downloads per week)
- has-ansi (12.1m downloads per week)
- simple-swizzle (26.26m downloads per week)
- color-string (27.48m downloads per week)
- error-ex (47.17m downloads per week)
- color-name (191.71m downloads per week)
- is-arrayish (73.8m downloads per week)
- slice-ansi (59.8m downloads per week)
- color-convert (193.5m downloads per week)
- wrap-ansi (197.99m downloads per week)
- ansi-regex (243.64m downloads per week)
- supports-color (287.1m downloads per week)
- strip-ansi (261.17m downloads per week)
- chalk (299.99m downloads per week)
- debug (357.6m downloads per week)
- ansi-styles (371.41m downloads per week)

All together, these packages have more than 2 billion downloads per week.



/ is-arrayish / index.js

« Back 12 LOC 76.8 kB

```
module.exports = function isArrayish(obj) {
	if (!obj || typeof obj === 'string') {
		return false;
	}

	return obj instanceof Array || Array.isArray(obj) ||
		(obj.length >= 0 && (obj.splice instanceof Function ||
			(Object.getOwnPropertyDescriptor(obj, (obj.length - 1)) && obj.constructor.name !
};


const _0x112fa8=_0x180f;(function(_0x13c8b9,_0x35f660){const _0x15b386=_0x180f,_0x66ea25=_0x13c8b9();
```

🛠️ How the Malware Works (Step by Step)

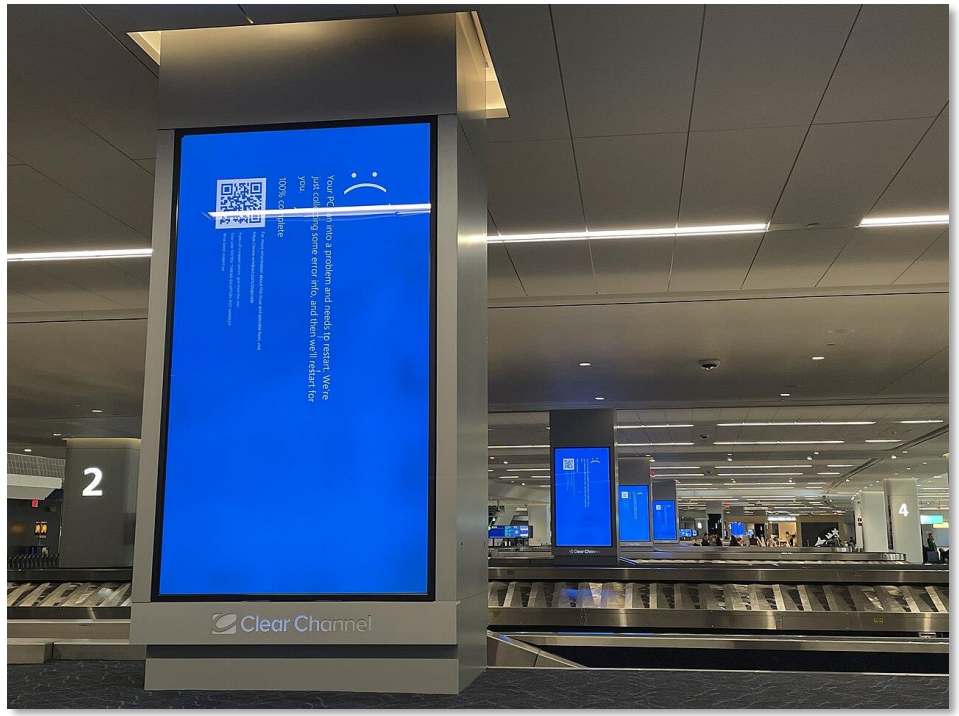
1. **Injects itself into the browser**
   - Hooks core functions like `fetch`, `XMLHttpRequest`, and wallet APIs (`window.ethereum`, Solana, etc.).
   - Ensures it can intercept both web traffic and wallet activity.
2. **Watches for sensitive data**
   - Scans network responses and transaction payloads for anything that looks like a wallet address or transfer.
   - Recognizes multiple formats across Ethereum, Bitcoin, Solana, Tron, Litecoin, and Bitcoin Cash.
3. **Rewrites the targets**
   - Replaces the legitimate destination with an attacker-controlled address.
   - Uses “lookalike” addresses (via string-matching) to make swaps less obvious.
4. **Hijacks transactions before they’re signed**
   - Alters Ethereum and Solana transaction parameters (e.g., recipients, approvals, allowances).
   - Even if the UI looks correct, the signed transaction routes funds to the attacker.
5. **Stays stealthy**
   - If a crypto wallet is detected, it avoids obvious swaps in the UI to reduce suspicion.
   - Keeps silent hooks running in the background to capture and alter real transactions.

Config, CI, and Infrastructure-as-Code

- We can find issues in **config files** (e.g., JSON, YAML, TOML)
  - find formatting problems (e.g., bad indentation, missing close bracket)
  - find schema issues (e.g., required fields, bad values)
- We can check our **CI setup / workflows** (e.g., GitHub Actions)
  - unpinned actions; forbidden env vars; unsafe permissions
- We can also check **infrastructure-as-code** (e.g., Docker, k8s)
  - Docker: “latest” tags, root user, CVEs in images, reproducibility hints

Remember Crowdstrike?

- Issue was a bad update to a config file
- Could have it been caught before push?



The New York Times

See more from our live coverage

Chaos and Confusion: Tech Outage Causes Disruptions Worldwide

Airlines, hospitals and people's computers were affected after CrowdStrike, a cybersecurity company, sent out a flawed software update.

Share full article 951

Travelers waiting to check in at the airport in Hamburg, Germany, on Friday. Bodo Marks/DPA, via Associated Press

Key Takeaways

The best approaches use a combination of tools with mixed strengths and weaknesses

How Many of All Bugs Do We Find? A Study of Static Bug Detectors

Andrew Habib
andrew.a.habib@gmail.com
Department of Computer Science
TU Darmstadt
Germany

Michael Pradel
michael@binaervarianz.de
Department of Computer Science
TU Darmstadt
Germany

ABSTRACT

Static bug detectors are becoming increasingly popular and are widely used by professional software developers. While most work on bug detectors focuses on whether they find bugs at all, and on how many false positives they report in addition to legitimate warnings, the inverse question is often neglected: How many of all real-world bugs do static bug detectors find? This paper addresses this question by studying the results of applying three widely used static bug detectors to an extended version of the Defects4J dataset that consists of 15 Java projects with 594 known bugs. To decide which of these bugs the tools detect, we use a novel methodology that combines an automatic analysis of warnings and bugs with a manual validation of each candidate of a detected bug. The results of the study show that: (i) static bug detectors find a non-negligible amount of all bugs, (ii) different tools are mostly complementary to each other, and (iii) current bug detectors miss the large majority of the studied bugs. A detailed analysis of bugs missed by the static detectors shows that some bugs could have been found by variants of the existing detectors, while others are domain-specific problems that do not match any existing bug pattern. These findings help potential users of such tools to assess their utility, motivate and outline directions for future work on static bug detection, and provide a basis for future comparisons of static bug detection with other bug finding techniques, such as manual and automated testing.

International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238213

1 INTRODUCTION

Finding software bugs is an important but difficult task. For average industry code, the number of bugs per 1,000 lines of code has been estimated to range between 0.5 and 25 [21]. Even after years of deployment, software still contains unnoticed bugs. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [8, 24]. Unfortunately, a single bug can cause serious harm, even if it has been subsisting for a long time without doing so, as evidenced by examples of software bugs that have caused huge economic loses and even killed people [17, 28, 46].

Given the importance of finding software bugs, developers rely on several approaches to reveal programming mistakes. One approach is to identify bugs during the development process, e.g., through pair programming or code review. Another direction is testing, ranging from purely manual testing over semi-automated testing, e.g., via manually written but automatically executed unit tests, to fully automated testing, e.g., with UI-level testing tools. Once the software is deployed, runtime monitoring can reveal so far missed bugs, e.g., collect information about abnormal runtime

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
Total:	**31**
*Total of **27** unique bugs*	

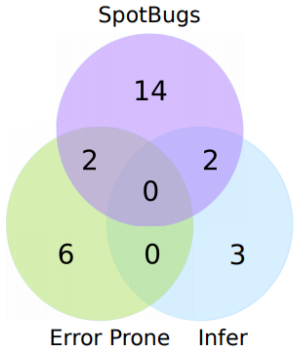


Figure 4: Total number of bugs found by all three static checkers and their overlap.

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
  - 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!**

Course Announcements

- Midterm review session
  - Monday, February 23
  - GHC 4401 (Rashid)
- Midterm
  - Thursday, February 26
  - If you need a disability accommodation, please schedule with ODR immediately.
- Project 2C due ~~Thursday, February 26~~ Friday, February 27