CS 4530: Fundamentals of Software Engineering

Lesson 9.3 Static Program Analysis

Jonathan Bell, Adeel Bhutta, Ferdinand Vesely, Mitch Wand
Khoury College of Computer Sciences

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Explain good uses for static analyzers
  - List limitations of static analyzers
  - Explain when to integrate static analysis into builds.

Why Static Analysis?

- CWE-798: Use of Hard-coded Credentials

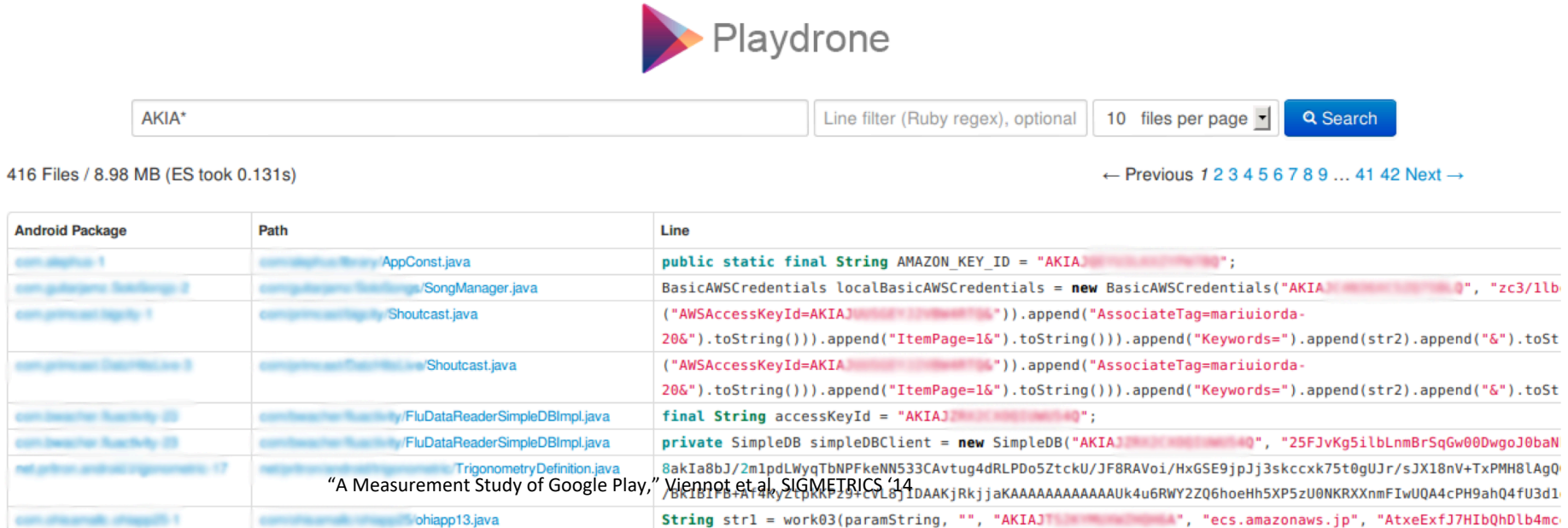
```
<SCRIPT>
function passWord() {
var testV = 1;
var pass1 = prompt('Please Enter Your Password',' ');
while (testV < 3) {
if (!pass1)
history.go(-1);
if (pass1.toLowerCase() == "letmein") {
alert('You Got it Right!');
window.open('protectpage.html');
break;
}
testV+=1;
var pass1 =
prompt('Access Denied - Password Incorrect, Please Try Again.','Password');
}
if (pass1.toLowerCase()!="password" & testV ==3)
history.go(-1);
return " ";
}
</SCRIPT>
<CENTER>
<FORM>
<input type="button" value="Enter Protected Area" onClick="passWord()">
</FORM>
</CENTER>
```

Why Static Analysis?

- CWE-798: Use of Hard-coded Credentials: Study of 1.1m Android Apps

	Amazon	Facebook	Twitter	Bitly	Flickr	Foursquare	Google	LinkedIn	Titanium
Total candidates	1,241	1,477	28,235	3,132	159	326	414	1,434	1,914
Unique candidates	308	460	6,228	616	89	177	225	181	1,783
Unique % valid	93.5%	71.7%	95.2%	88.8%	100%	97.7%	96.0%	97.2%	99.8%

Table 5: Credentials statistics from June 22, 2013 and validated on November 11, 2013. A credential may consist of an ID token and secret authentication token.



"A Measurement Study of Google Play," Viennot et al, SIGMETRICS '14

Why Static Analysis?

AWS urges developers to scrub GitHub of secret keys

By Munir Kotadia
Mar 24 2014
10:18AM

13 Comments

RELATED ARTICLES

Amazon to block police use of facial recognition for a year

Amazon turns to Chinese firm on US blacklist to meet thermal camera needs

UN experts demand probe into alleged Saudi hack of Amazon boss Bezos

Glenn Gore moves from AWS to Affinidi

Devs hit with unexpected bills after leaving secret keys exposed.

Amazon Web Services (AWS) is urging developers using the code sharing site GitHub to check their posts to ensure they haven't inadvertently exposed their log-in credentials.

Thousands of 'secret keys', which unlock access to private Amazon Web Services accounts are currently available unencrypted to members of the public with just two clicks of a mouse.

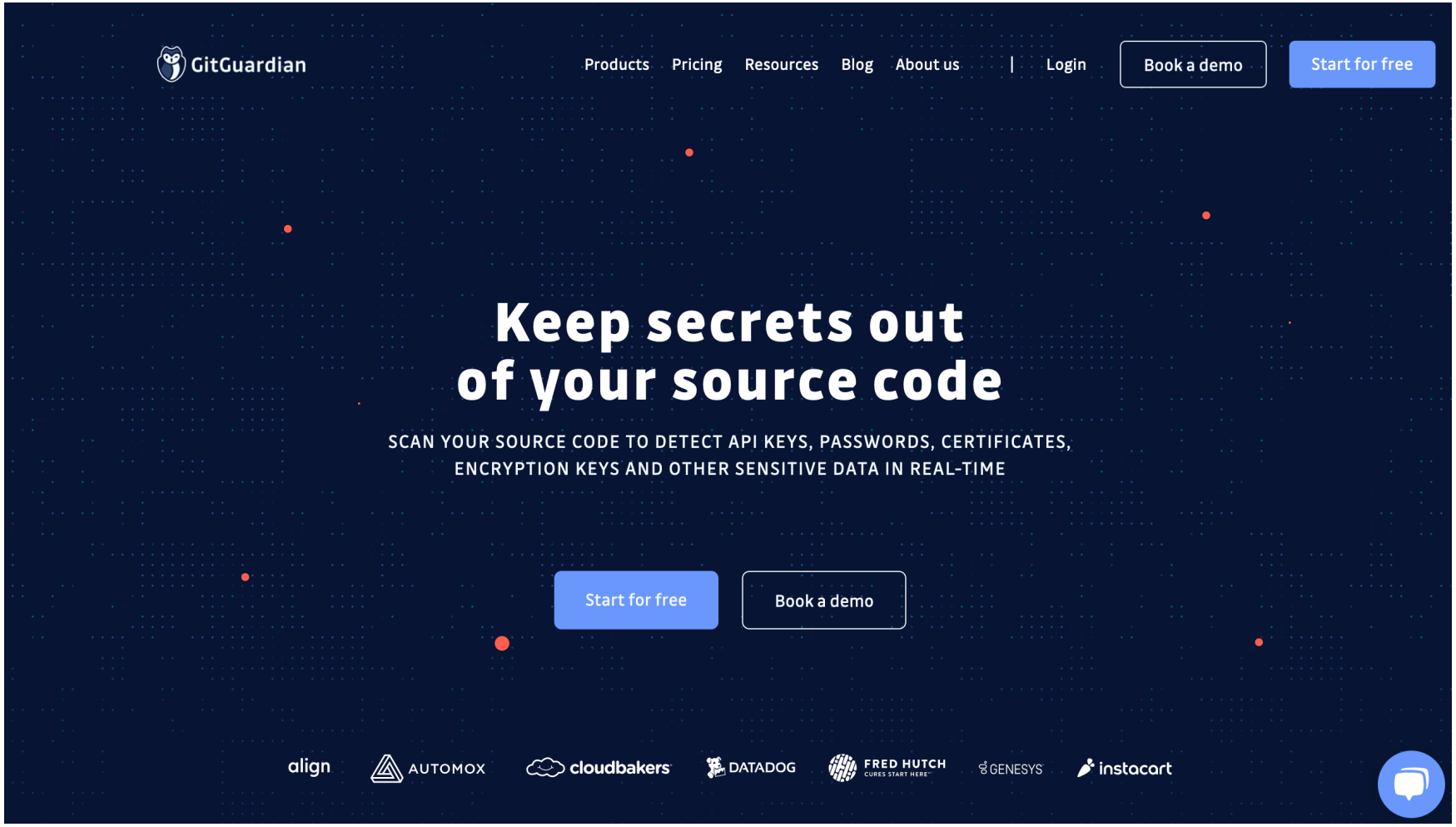
The secret keys are issued by Amazon Web Services when users open an account and provide applications access to AWS resources.

When opening an account, users are told to "store the keys in a secure location" and are warned that the key needs to remain "confidential in order to protect your account".

AWS reminds subscribers that "anyone who has your access key has the same level of access to your AWS resources that you do. Consequently, we go to significant lengths to protect your access keys, and in keeping with our shared-responsibility model, you should as well."

Why Static Analysis?

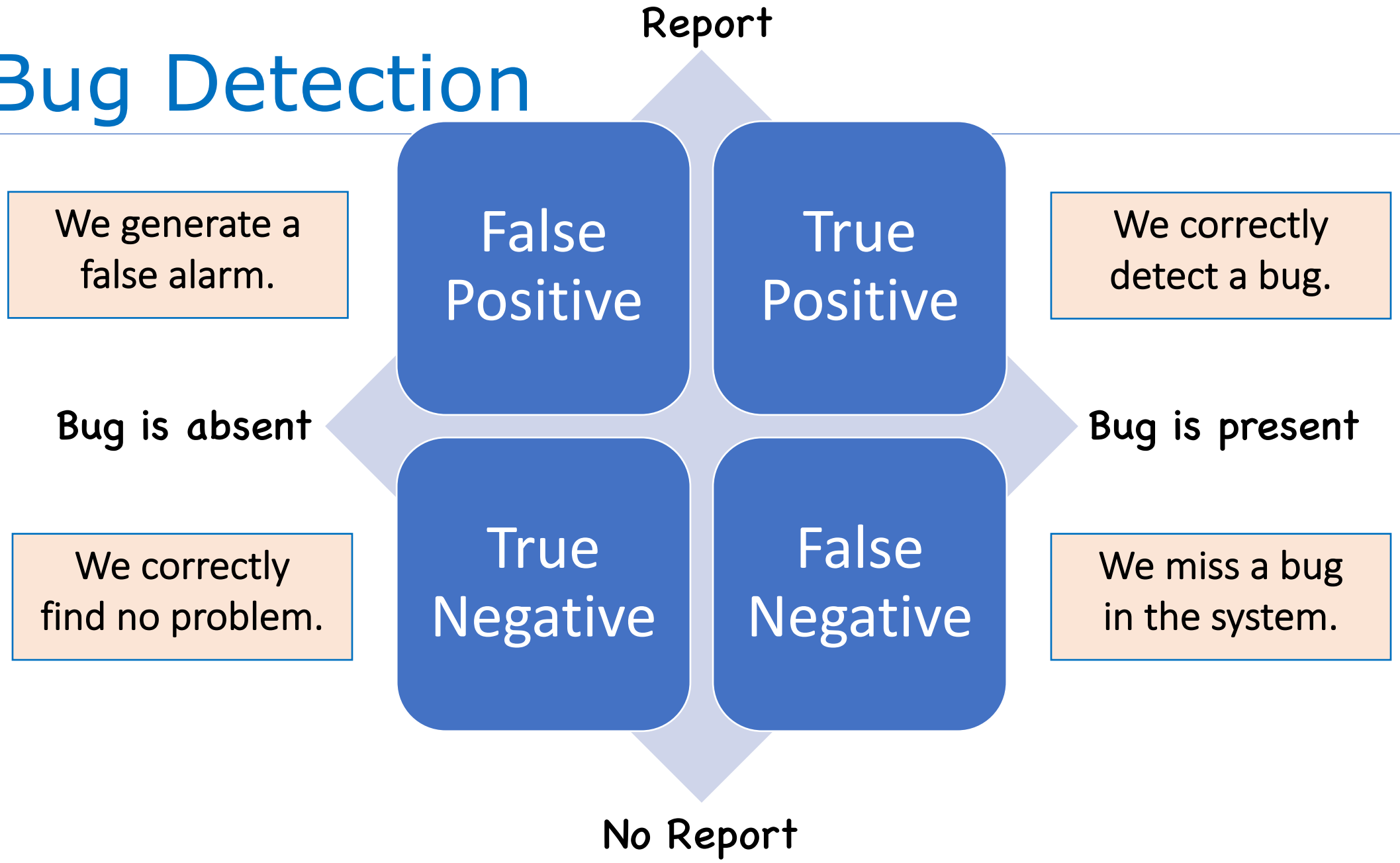
- GitGuardian (Launched in 2017)



Why Static Analysis?

- A9:2017-Using Components with Known Vulnerabilities

Bug Detection



Static Program Analysis

- Bug finders and partial verification use static program analysis
  - Reads in the program (just like a compiler);
  - Analyze to determine properties:
    - E.g., are all open resources eventually closed?
  - ”static” means ”without running the program”.
- All non-trivial properties are undecidable
  - Approximations are always necessary: make a choice
    - E.g., miss some closes of open resources, or
    - Miss some open resources not being closed.

Compromises with Static Analysis

- Getting precise results may take **time**:
  - Many algorithms are exponential in precision measures.
- Getting precise results may require **whole** program:
  - If parts of the program loaded at runtime:
    - Analysis results may be very imprecise, or (worse)
    - Incorrect, if they assume the whole program is available.
- Getting precise results may require intervention:
  - Code may need to be **annotated** with information:
    - E.g., this method may return an open resource.

Effects of Analysis Imprecision

FALSE NEGATIVES

- The static analysis misses something ”bad” in program:
  - Bug not found.
- Can give a false sense of security.
- Can be reduced, but at the cost of false positives!

FALSE POSITIVES

- The static analysis reports a problem that doesn’t exist:
  - There is no bug.
- Real bugs can be swamped by a flood of spurious reports.
- Programmer time is wasted chasing down false leads.

Google defined “Effective False Positive”

- A report from static analysis is effectively false,
  - If it is ignored by developers;
  - Whether or not it represents a true bug.
- Even if the report is technically correct
  - It may refer to something considered unimportant:
    - E.g., who cares if all the files aren’t closed, if the program is about to exit anyway.
    - E.g., yes, there is a race condition between two logging statements, but that’s not important.
- Even if the report is technically wrong
  - Developers may see potential problem, and fix.

Criteria For Automated Program Analysis

- Efficient and Easy
  - Should not require whole program or annotations.
- Rarely spurious
  - No more than 10% effectively false positive.
- Actionable
  - Should point out things easy to fix.
- Effective
  - Problems should be perceived as important.

Automatically applied during Code Review.

Source: Software Engineering at Google, Chapter 20

Defects Static Analysis can Catch

- Defects that result from inconsistently following simple, mechanical design rules.
  - **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

Key: check compliance to simple, mechanical design rules

Program Verification as an Alternative to Testing

- So “program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra’s Law), what can we do?
- Testing is limited to finite concrete cases;
  - Can we check unbounded symbolic cases?
- Yes … with caveats

Verification Checks Code Against Specification

A Specification: What the system is supposed to do.

B Code: What the system actually does.

- Difficulties:
  - Need a **full formal** specification;
  - Even proving termination is undecidable, let alone proving adherence to a specification.

A Full Formal Specification

- … precisely defines exactly the behavior that the system should have:
  - What the outputs are in terms of the inputs;
  - What behaviors the system should have.
- Wait a minute! That's called a program.
- Yes, a full formal specification is essentially a program, perhaps expressed at a higher level.
  - … with all the complexity that entails,
  - … including bugs!

Inefficiently (w/o algorithms)

We can't avoid Human fallibility.

Verification Doesn’t Prove Presence of Bugs

- Verification fails if
  - Missing lemma for unit behavior, or
  - Cannot verify loop invariant, or
  - Functional specification missing a piece, or
  - Run out of time trying to construct proof, or
  - Specification is wrong.
- Constructing the proof can easily take as long as constructing the software, if not much more.
  - Just because there is no proof does not mean the software has a fault.

Full Verification is (Still) Hard

- Full program verification is still relatively costly in terms of time and expertise
- However, more and more lightweight formal methods are being used as part of development
- Linters
- Static type checkers

Example: ESLint

- Linters = static analyzers for finding problematic patterns in code, stylistic errors, bugs
- ESLint is a popular linter for JavaScript
- Customizable via a set of rules – JS code implementing a check

Possible Problems

These rules relate to possible logic errors in code:

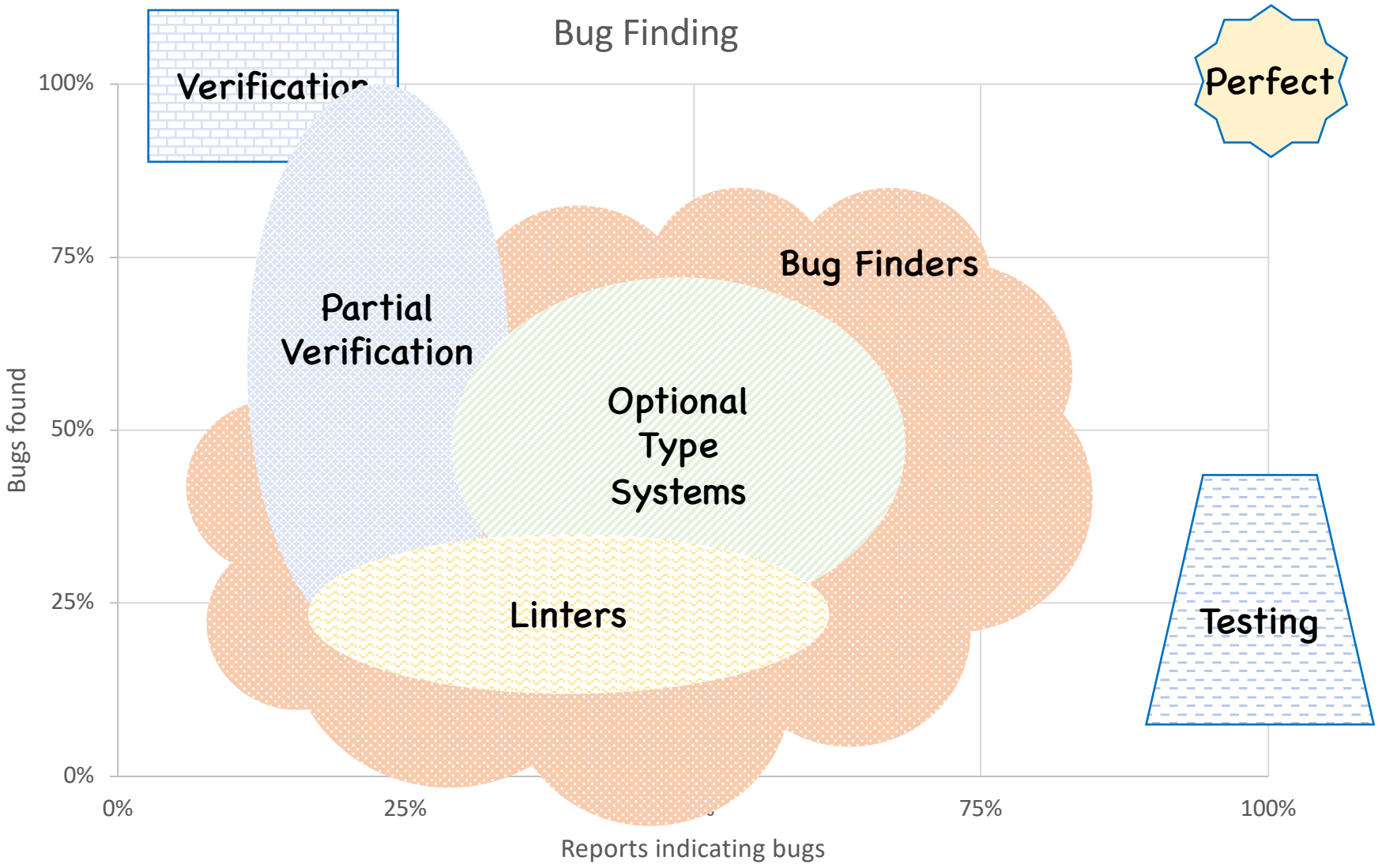
	array-callback-return	enforce `return` statements in callbacks of array methods
✓	constructor-super	require `super()` calls in constructors
✓	for-direction	enforce "for" loop update clause moving the counter in the right direction.
✓	getter-return	enforce `return` statements in getters
✓	no-async-promise-executor	disallow using an async function as a Promise executor
	no-await-in-loop	disallow `await` inside of loops
✓	no-class-assign	disallow reassigning class members

Example: Type systems

- (Static) Type systems and type checkers are also static program analyzers
- JavaScript is a dynamically typed language – types are only known at runtime, as values are computed
- If a piece of code is not executed during testing, we will not know if it contains an inconsistent use of variables
- Even if the code is executed, the error might not be detected

Example: Type systems

- TypeScript adds a type system and checker that can be run before a program is run
- A static type checker reads a program before its run and, essentially, *proves* that the program is free of some classes of bugs
- A type is a formal property of a piece of code
- E.g., “given an argument x of type number and an argument s of type string, function foo will compute a number (or fail)”
- All code is checked for type errors, no type errors at runtime are possible


Bug Finding
Verification
Perfect
Bug Finders
Partial Verification
Optional Type Systems
Linters
Testing
100%
75%
50%
25%
0%
Bugs found
0%
25%
75%
100%
Reports indicating bugs

Review: Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Explain good uses for static analyzers
  - List limitations of static analyzers
  - Explain when to integrate static analysis into builds.