SYNOPSYS®
Silicon to Software™

Static Analysis for Web Security

Aaron Hurst

ahurst@synopsys.com

February 5, 2020

CONFIDENTIAL INFORMATION

The information contained in this presentation is the confidential and proprietary information of Synopsys. You are not permitted to disseminate or use any of the information provided to you in this presentation outside of Synopsys without prior written authorization.

IMPORTANT NOTICE

In the event information in this presentation reflects Synopsys' future plans, such plans are as of the date of this presentation and are subject to change. Synopsys is not obligated to update this presentation or develop the products with the features and functionality discussed in this presentation. Additionally, Synopsys' services and products may only be offered and purchased pursuant to an authorized quote and purchase order or a mutually agreed upon written contract with Synopsys.

Personal History

- Carnegie Mellon BS/MS 2002
- UC Berkeley Ph.D 2008
  - Algorithms to optimize digital logic circuits
- Research Scientist for a few years
  - More of the above…
- Joined Coverity in 2011
  - Developer
  - Architect
  - Manager
- Based in San Francisco

Coverity

- Coverity started in 2002 as a spin-off from Stanford University
  - Started life as a bug-finding static analysis for C/C++
  - Now supports 14 languages and finds bugs + security vulnerabilities
  - Used by thousands of customers across the entire software industry
  - Used to analyze everything from medical devices, airplanes, and cars to bank websites, mobile apps, and IoT devices

See also…

A few billion lines of code later: using static analysis to find bugs in the real world

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Gros C.-H., Kamsky, A., McPeak, S. and Engler, D., 2010. *Communications of the ACM*, *53*(2), pp.66-75.

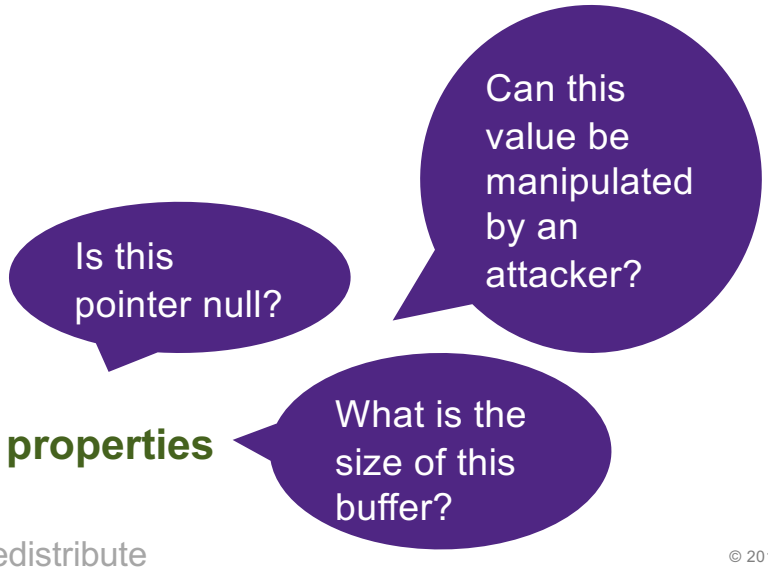
- Synopsys acquired Coverity in 2014 as the foundation for its Software Integrity Group (SIG)
  - SIG now sells several tools and consulting services for $300+ million / year in revenue

SYNOPSYS®

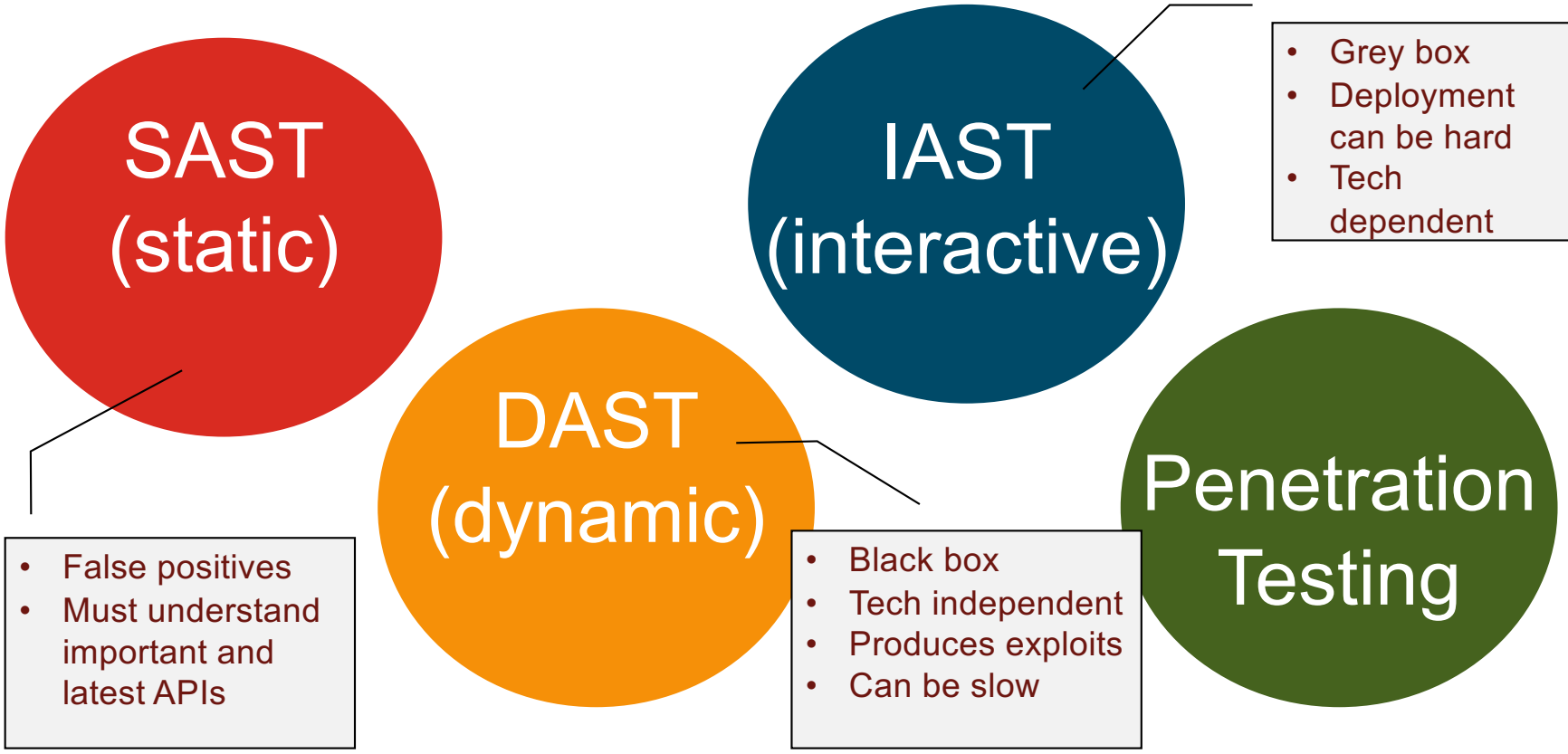
Build secure, high-quality software faster

What is Static Analysis?

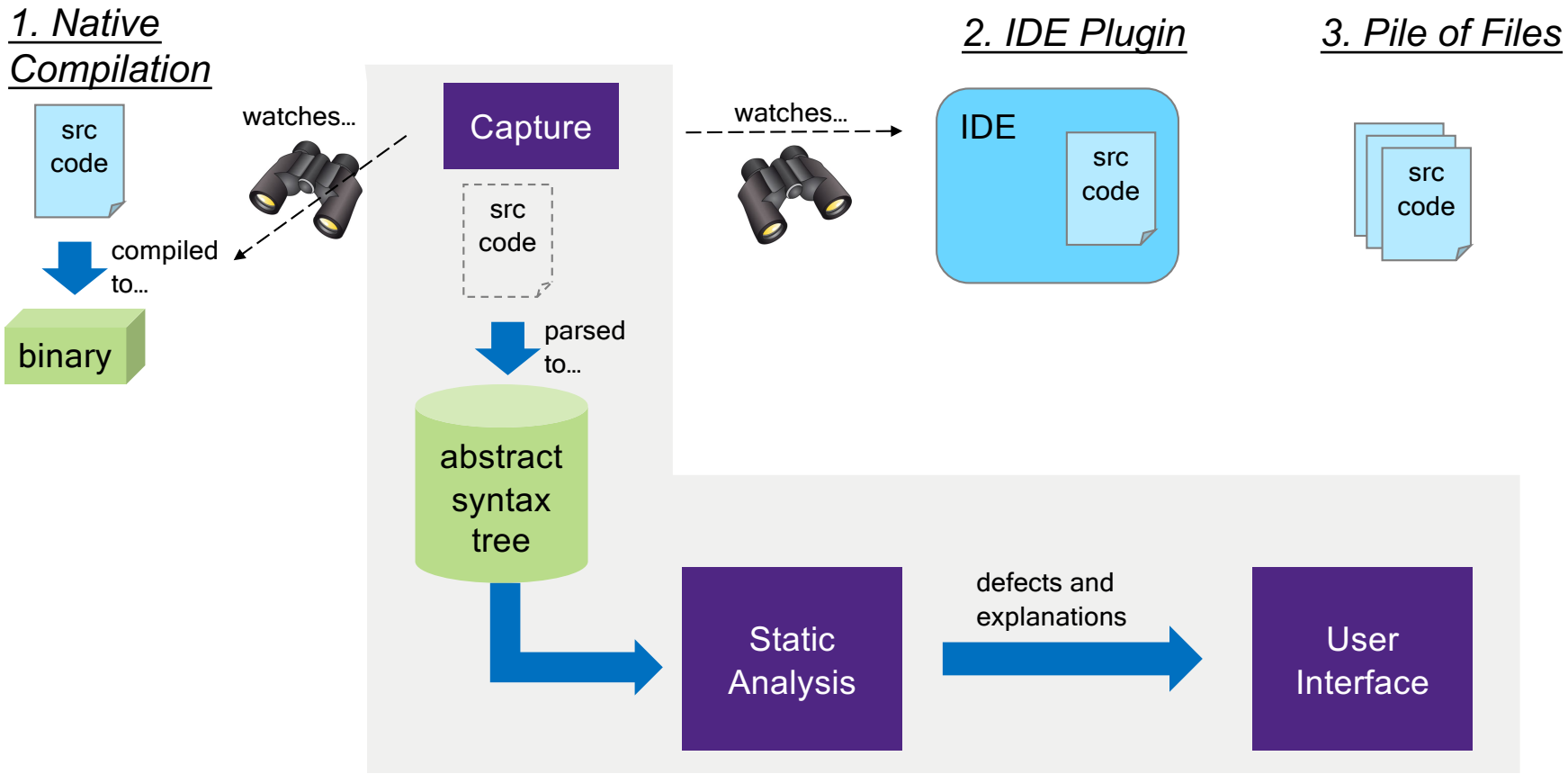
- Analysis of *program code*
  - Rather than running and observing behavior
- Analysis of *all paths* (theoretically)
  - Rather than only those encountered during execution
  - No reliance on a test suite
- Obviously not practical to *simulate* execution
  - Huge state space
  - Slow
  - Halting problem
  - Unknown inputs
- Often need only reason about simpler **abstract properties**



Application Security Testing (AST)

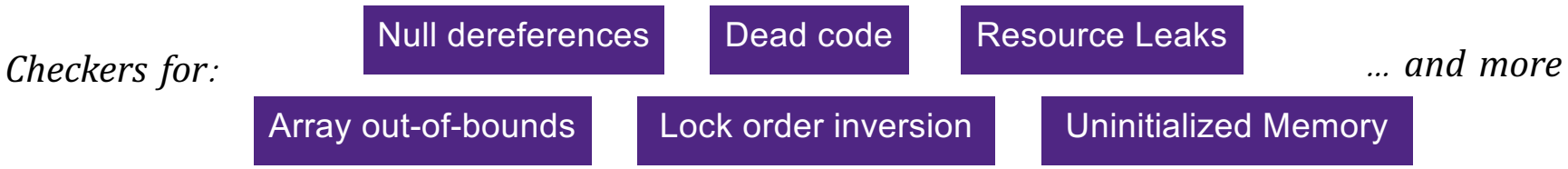


Tool Workflow



Tackling Security

- Coverity proved that static analysis could be successful as a commercial tool



- Around 2011 we started thinking… can we leverage this technology to build a web application security analysis tool?

Top 10 Web Application Security Risks

Label	Vulnerability
A1	Injection
A2	Broken Authentication
A3	Sensitive Data Exposure
A4	XML External Entities (XXE)
A5	Broken Access Control
A6	Security Misconfiguration
A7	Cross-Site Scripting XSS
A8	Insecure Deserialization
A9	Using Components with Known Vulnerabilities.
A10	Insufficient Logging & Monitoring

Involve classifying and **tracking data** within an application

Tracking Data

- **Taint analysis** is general approach to classifying and tracking origins of data

Sources	program actions that **taint** data	reading data from an untrusted connection
Transfer Rules	copy taint from one object to another	object deserialization APIs
Built-in Propagation	pass values through the program	assignment, function calls
Sanitizers	program actions that **un-taint** data	HTML escaping, comparing against a whitelist
Sinks	program actions that are unsafe to use with tainted data	concatenating a SQL query, parsing XML DTDs

On a Real Example

```
const express = require("express");
const app     = express();

app.get("/dbquery",
        function run(req, res, next) {
            const id = req.query.id;
            const query = `select * from User where userid=${id}`;
            const sql = require("mssql");

            sql.connect(getConnectionConfig()).then(
                function() {
                    new sql.Request().query(query).then(
                        // result callback ...
                    );
                });
            res.send("Done");
        });

app.listen(1337, function() {
    console.log("Express listening...");
});
```

Source

(1)

(2)

Sink

A security vulnerability occurs when…
a source reaches a sink

It Becomes Difficult…

- Int***ra*** procedural looks at each function in isolation
  - No or very limited knowledge of other functions
  - Simple, easy, fast
  - Works great for some checks, poorly for data-sensitive

- Int***er*** procedural looks across function boundaries
  - Can know a lot about other functions
  - Complex, hard, slow
  - Necessary for sophisticated data-sensitive checks

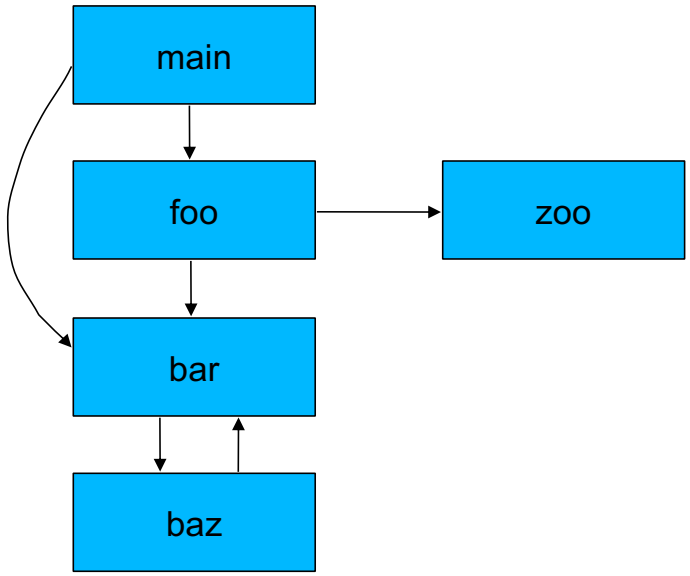
Call graph

```
int main()
{
  foo();
  bar(4);
  return 0;
}
```

```
void bar(int x)
{
  if (x > 0) {
    baz(x-1);
  }
}
```

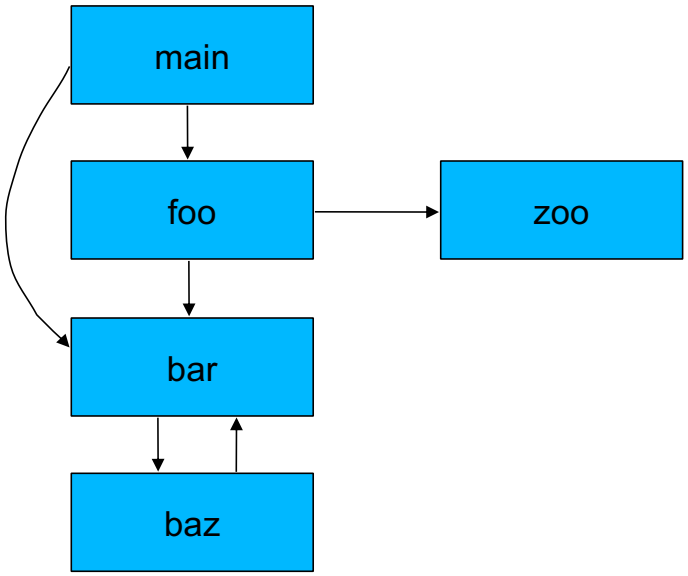
```
void foo()
{
  bar(5);
  zoo();
}
```

```
void baz()
{
  bar();
}
```



Traversing the Call graph

- **Top-down analysis**
  - Work from root towards leaves
  - Accumulate call context/arguments
  - Iterate until fixpoint or a limit

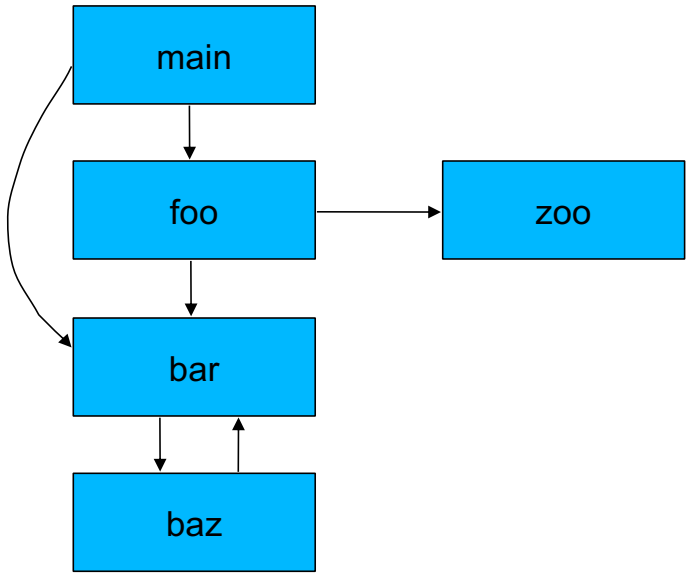


Traversing the Call graph

- **Top-down analysis**
  - Work from root towards leaves
  - Accumulate call context/arguments
  - Iterate until fixpoint or a limit

- **Bottom-up analysis**
  - Work from leaves toward root
  - Break cycles arbitrarily
  - Create a **summary** of what each callee does

- There exist other ways to slice a program



Customers have simple needs…

- Analyze all of their code
  - In all the languages in which they develop
  - As it's accepted by their build and runtime tools
- Don't ask hard questions
  - No code annotations or other help
  - Building code is hard, can't they just point Coverity at git repos?
- Find all the bugs and security vulnerabilities -- **no false negatives (FNs)**
- Only report bugs and vulnerabilities that we care enough to fix – **no false positives (FPs)**
  - …and offer guidance on how to fix them
- Do it **fast** even if I run it on my ten year old laptop

(I exaggerate, but only slightly)

Finding bugs versus finding security vulnerabilities

Finding bugs

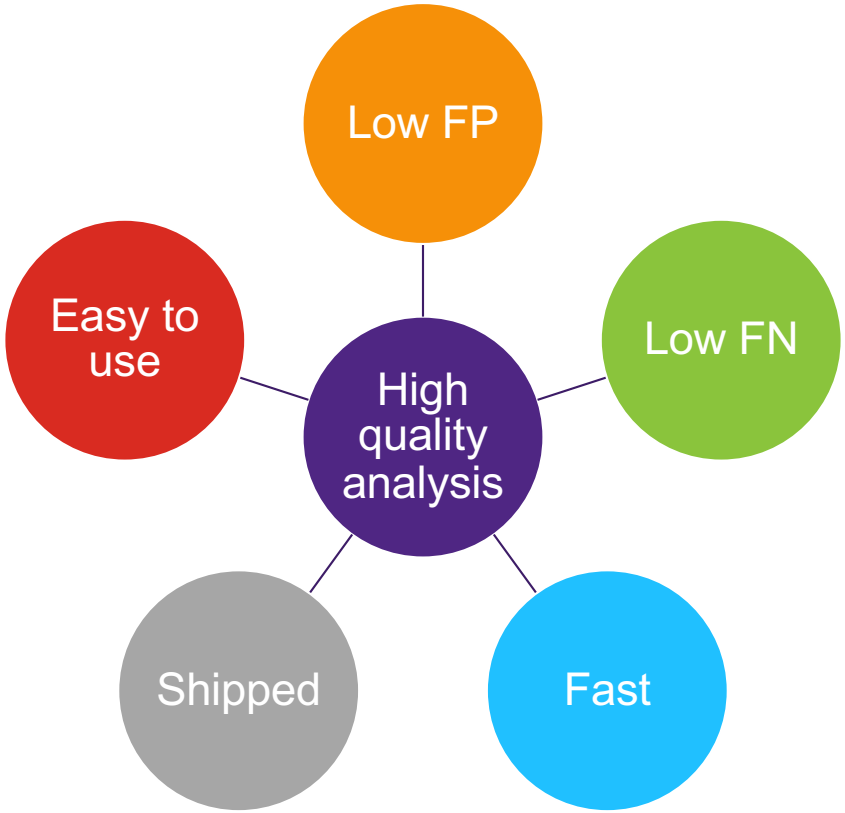
- Programmers versus Complexity
- Users learn to step around bugs
  - Low impact findings may not be worth fixing
- Moderate defect density (~1K / MLoC)
  - FPs waste time ➔ worse cost/benefit
- Emphasis is on high impact, low FP, fast
  - Path + callsite sensitive, interprocedural
  - Unsound analysis to eliminate FPs

Finding security vulnerabilities

- Programmers versus Attackers
- Attackers only need one way in
  - FNs are bad
- Lower density of vulnerabilities (~300 / MLoC)
  - FPs are more tolerable
- Emphasis on thoroughness / low FNs
  - Flow + callsite + field sensitive, interprocedural
  - Soundy analysis*

*soundy in the sense of *"In defense of soundiness: a manifesto"* by Livshits et al

The reality: a constant balancing act

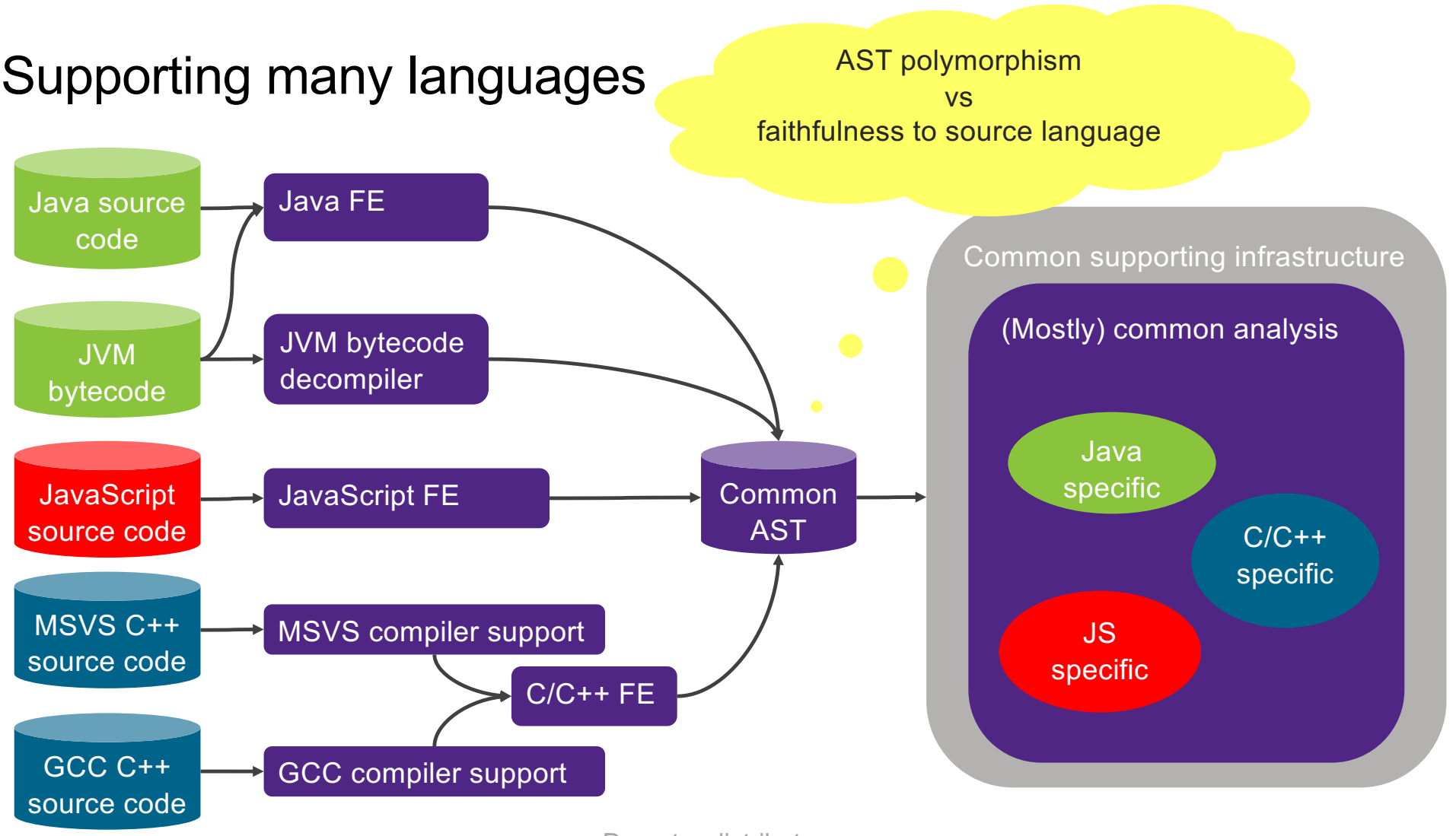


Teams use many languages to build web and mobile apps

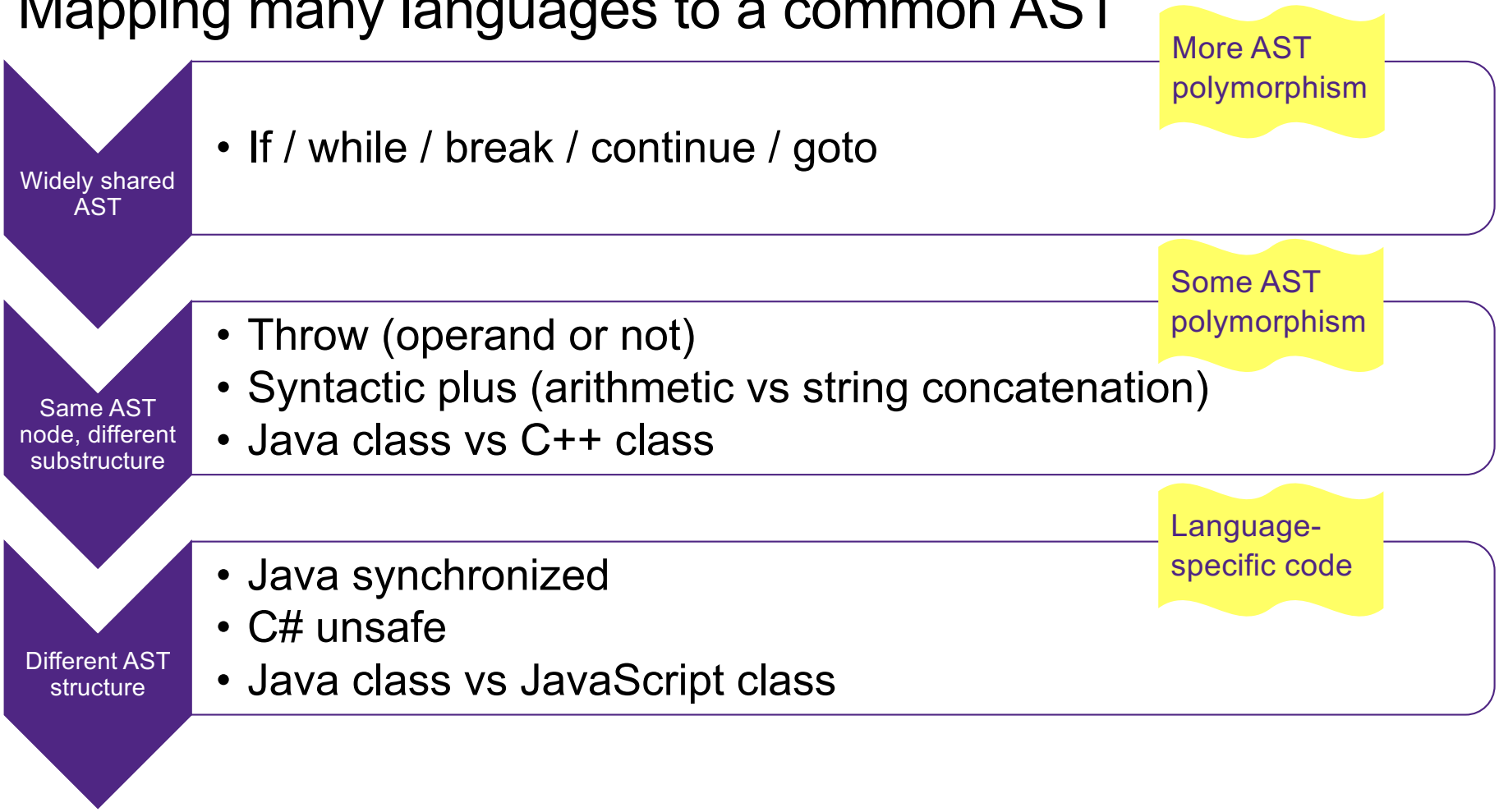
- Web applications and web services
  - Java
  - .NET (C#, Visual Basic)
  - JavaScript (client, Node.js, XS JS)
  - TypeScript
  - PHP
  - Python
  - Ruby
  - Go*
- Mobile applications
  - Android
    - Java
    - C/C++
    - Kotlin*
    - JavaScript (Cordova)*
  - iOS
    - Swift
    - JavaScript (Cordova)*
    - Objective-C

* In development

Supporting many languages



Mapping many languages to a common AST



Modern webapps are built on frameworks

- User code doesn't start at main()
  - User code implements classes/methods/callbacks with specific signatures
  - User code annotates or registers these callbacks programmatically or via configuration files
  - The framework calls into these callbacks in specific lifecycles (implicit control flow)
  - …and passes objects around for you (implicit dataflow)

➔ You can't have sound / soundy webapp security analysis without framework support

Example: Spring MVC + JSPs

Find the XSS!

`HelloController` contains webapp entry points

`hello()` is a webapp entry point at URL `http://<root>/hello`

A HTTP request to `http://<root>/hello?n=Sam` Calls `hello("Sam")`

- src/com/coveri[illegible]mvc/HelloController.java

After `hello()`, control proceeds to a view called "`helloView`"

```
troller
c class HelloController {
@RequestMapping("/hello")
ModelAn ew hello(@RequestParam("n") String name) {
    return new ModelAndVie ("helloView") addObject("helloTo", name);
}
}
```

Store `name` into model property `helloTo`

- WebContent/WEB-INF/views/helloView.jsp

```
<body> Hello, ${helloTo}. </body>
```

Example: Spring MVC + JSPs

Find the XSS!

View resolution involves the framework configuration

- WebContent/WEB-INF/spring-servlet.xml

```
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

- src/com/coveri[illegible]mvc/HelloController.java

After `hello()`, control proceeds to a view called "`helloView`"

```
troller
c class HelloController {
@RequestMapping("/hello")
ModelAn[illegible]ew hello(@RequestParam("n") String name) {
    return new ModelAndView("helloView").addObject("helloTo", name);
}
}
```

- WebContent/WEB-INF/views/helloView.jsp

```
<body> Hello, ${helloTo}. </body>
```


Example: Spring MVC + JSPs

Find the XSS!

- WebContent/WEB-INF/spring-servlet.xml

```
<beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <beans:property name="prefix" value="/WEB-INF/views/" />
        <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

- src/com/coverity/samples/springmvc/HelloController.java

After `hello()`, control proceeds to `helloView.jsp`

Store `name` into model property `helloTo`

```
@Controller
public class HelloController {
    @RequestMapping("/hello")
    ModelAndView hello(@RequestParam("n") String name) {
        return new ModelAndView("helloView").addObject("helloTo", name);
    }
}
```

- WebContent/WEB-INF/views/helloView.jsp

Read model property `helloTo` and use it to construct HTML

```
<body> Hello, ${helloTo}.
```

XSS

Supporting frameworks in static analysis

- Direct analysis of a framework implementation is a non-starter
  - Reflection
  - Parsing of config files
  - Programmatic registration of callbacks
- Supporting a new framework requires
  - Reading and experiment to understand it
  - Design of program analysis to analyze it
- Supporting a framework is a moderate project in the craft of applied program analysis

You’re not analyzing my JavaScript project

Paraphrased interaction with customers

Customer: You’re not analyzing my JavaScript project.

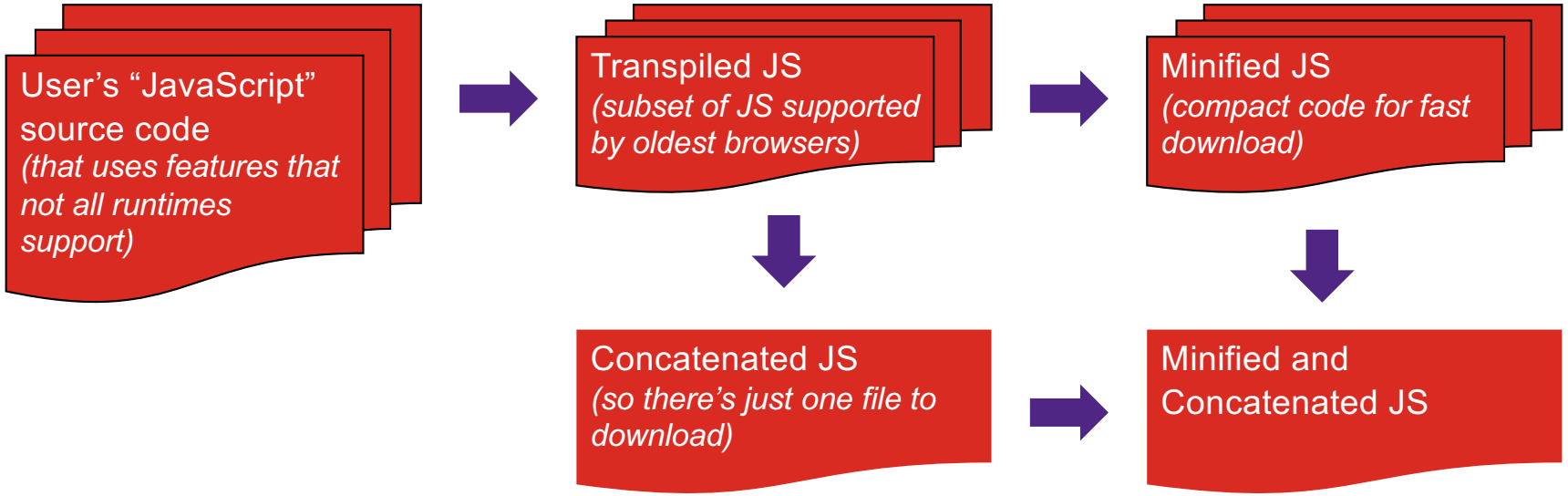
Us: Can you show us an example of what we aren’t analyzing?

Customer: Here you go.

Us: That’s TypeScript code and a bunch of React templates.

(Or fill in Flow / Jade / Pug / EJS / etc. template)

JavaScript: Transpilation, Minification, Concatenation



➔ Coverity gets some arbitrary subset of the { original, transpiled, minified / concatenated } code

- Want to analyze what's closest to the source artifact
- Don't want to analyze duplicates

Analyzing JavaScript

Well-behaved, static languages

- Static types distinguish
  - Field access
  - Collection operations
  - Dictionary operations
- Call sites include a static call target
- Classes are a top level entity
- Imports / namespaces are well defined

JavaScript

- No static types
  - Field access / array access / dictionary access indistinguishable statically
- Call resolution is a whole-program dataflow problem
- Various roll-your-own class, module, dependency injection systems

Finding XSS in template code

How to find the XSS here?

src/app.js

```
const path = require('path');
const express = require('express');
const app = express();


const articleRouter =
require('./router/article');


app.set('views', path.join(__dirname,
'views'));
app.set('view engine', 'pug');


app.get('/article',
articleRouter.index);
```

src/router/article.js

```
const dao = require('../graph/article');

exports.index = (req, res) => {
  const title = req.query.title;
  const article_object = {title};

  dao.getArticle(title, (err, docs) => {
    if (err) {
      return next(err);
    }

    article_object.body = docs.body;
    return res.render('article',
{article_object});
  });
}
```

app code

src/views/article.pug

```
mixin article(obj)
  .article
    .article-wrapper
      h1!= obj.title
      p= obj.body

+article(article_object)
```

template

Thanks to Romain Gaucher for this example

Finding XSS in template code

How to find the XSS here?

src/app.js

```
const path = require('path');
const express = require('express');
const app = express();


const articleRouter =
require('./router/article');


app.set('views', path.join(__dirname,
'views'));
app.set('view engine', 'pug');


app.get('/article',
articleRouter.index);
```

src/router/article.js

```
const dao = require('../graph/article');

exports.index = (req, res) => {
  const title = req.query.title;
  const article_object = {title};

  dao.getArticle(title, (err, docs) => {
    if (err) {
      return next(err);
    }

    article_object.body = docs.body;
    return res.render('article',
{article_object});
  });
}
```

app code

src/views/article.pug

```
mixin article(obj)
  .article

    h1!= obj.title

+article(article_object)
```

template

Thanks to Romain Gaucher for this example

Finding XSS in template code

How to find the XSS here?

src/app.js

```
const path = require('path');
const express = require('express');
const app = express();

const articleRouter =
require('./router/article');
```

view configuration

```
app.set('views', path.join(__dirname,
'views'));
app.set('view engine', 'pug');
```

0

entry point (route)

```
app.get('/article',
articleRouter.index);
```

1

src/router/article.js

```
const dao = require('../graph/article');

exports.index = (req, res) => {
  const title = req.query.title;
  const article_object = {title};

  dao.getArticle(title, (err, docs) => {
    if (err) {
      return next(err);
    }

    article_object.body = docs.body;
    return res.render('article',
{article_object});
  });
}
```

tainted data entering

2

dispatch "model" to the view

3

src/views/article.pug

```
mixin article(obj)
  .article
    .a
      h1!= obj.title
      p= obj.body

+article(article_object)
```

Sink taking "title" property

5

mixin call w/ "article_object" property

4

Thanks to Romain Gaucher for this example

Template Languages

A few examples from the bestiary

EJS

```
<link rel="icon" type="image/png"
href="../images/favicon.png">
    <% if (themeFiles && themeFiles.css) { %>
        <% for(var i=0, l=themeFiles.css.length; i<l; i++)
{%>
    <link href='<%= themeFiles.css[i]%>' rel='stylesheet'
type='text/css'>
        <% } %>
    <% } %>
<title><%- countlyTitle %></title>
```

Handlebars

```
{{#if user}}
    <li class="{{#if isAccountPage}} active {{/if}}"><a
href="/account" title="My account"><span id="loggedUser"
data-user="{{ user.name }}"></span><i class="fa fa-user"></i>
{{ user.name }}</a></li>
    <li><a href="/logout" title="Sign out"><i class="fa fa-
sign-out"></i></a></li>
{{else}}
    <li class="{{#if isLoginPage}} active {{/if}}"><a
href="/login"><i class="fa fa-sign-in"></i> Sign in</a></li>
{{/if}}
```

Jade

```
extends main
include mixins
block vars
    - var hasToolbar = true
block content
    div(ng-app='allcount', ng-controller='EntityViewController')
        +defaultToolbar()
        .container.screen-container(ng-cloak)
            +defaultEditAndCreateForms()
block js
    +entityJs()
```

Nunjucks

```
<link rel="icon" type="image/png" href="./favicon.ico" />
<meta name="viewport" content="width=device-width" />
<% block css %><% endblock %>
<$ 'vendor' | css | safe $>
<$ 'style' | css | safe $>
<$ 'vendor' | js | safe $>
<% block js %><% endblock %>
```

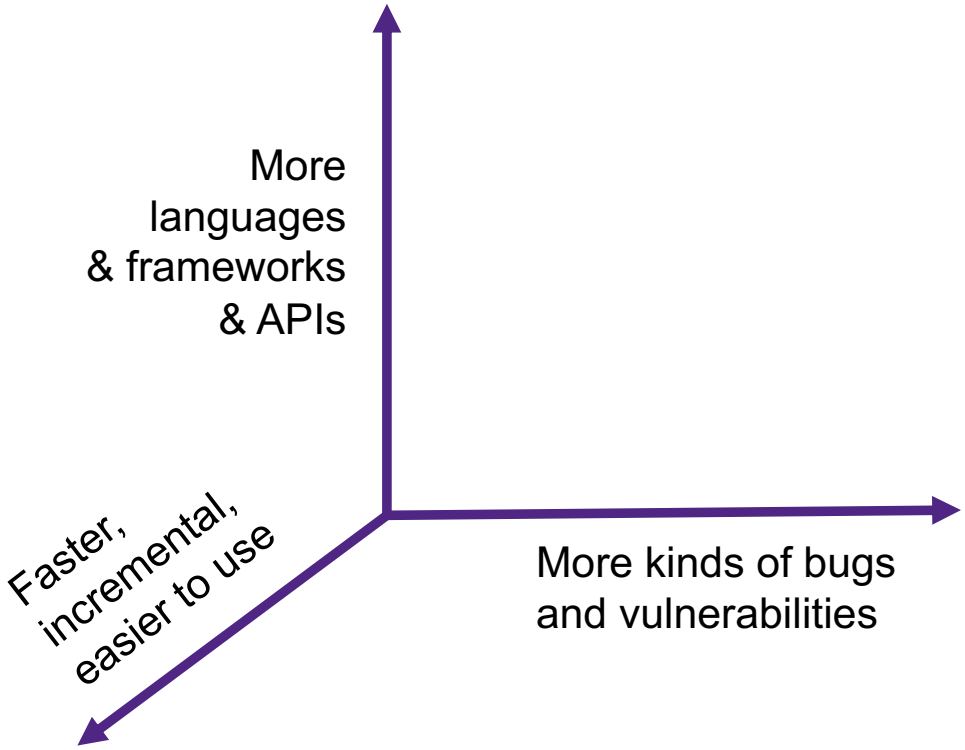
Thanks to Romain Gaucher for this example

What Commercial Static Analysis Security Looks Like

- Analyze the user's code
  - Support the common languages in which users develop
    - Accept what language tools accept
    - Make sense of whatever pile of source artifacts we can get
    - Deal with the dynamic languages and pervasive higher-order functions
  - Analyze source code and bytecode, but prefer source code
  - Include special support for prevalent webapp and mobile app frameworks
    - Analyze framework-specific configuration files
    - Build re-usable abstractions to help scale framework support; apply elbow grease
  - Support the accompanying template languages in the ecosystem
    - Use sneaky automatic dynamic analysis techniques to simplify template support
  - Underapproximate to ensure scaling and save engineer-time
  - Do (most of) it fast in IDE
- Report bugs and security vulnerabilities
  - Build low FP, fast bug finding checkers for high impact issues
  - Build low FN security vulnerability checkers to cover a systematic set of vulnerabilities
- Profit

Adventures in Commercial-Grade Static Analysis

A boundless supply of interesting problems



Interested?

Share your resume with us!

Follow
#lifeatsynopsys

SYNOPSYS®

Acknowledgements

Thanks to my colleagues at Synopsys for examples, slides, and feedback on earlier drafts

- **Simon Goldsmith (who crafted many of these slides)**
- Romain Gaucher
- Cameron Forbis
- Rody Kersten
- Mitch Mlinar
- Tushar Sharma
- John Fitzpatrick
- Wanying Luo
- Abhishek
- Marc-Andre Laverdiere
- Dzintars Avots
- Arun Mattikalli
- Joanna Bujes
- Clarence Cromwell
- Brett van Swelm
- Sang Phan
- Lijesh Krishnan
- …and the many smart people that have worked with me over the years to design, develop, evaluate, market, sell, and deploy Coverity