ShiftLeft

Elegant and Scalable Code Querying with

Code Property Graphs

Fabian Yamaguchi
October 4, 2019, Connected Data London

ocular ShiftLeft

Fabian Yamaguchi

Chief Scientist, ShiftLeft Inc.

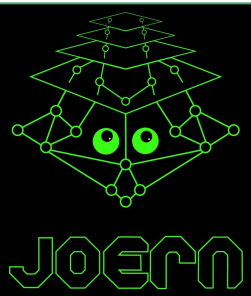
Github: fabsx00
Twitter: @fabsx00
Email: fabs@shiftleft.io

PhD, University of Goettingen
Loves code analysis, vulnerability discovery, machine learning, and graphs!

What is Ocular? What is Joern?

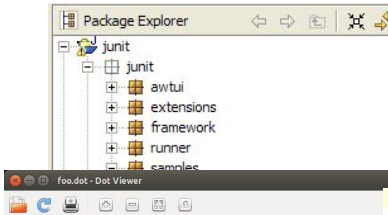
- Goal: provide query language to describe patterns in code
  - to identify bugs and vulnerabilities
  - to help in deeply understanding large programs
- Think of it as an extensible Code Analysis Machine
- Programmable in JVM-based languages (e.g., Java/Scala/Kotlin)
- You can write scripts, language extensions and libraries on top of it
- Joern is Ocular's free-spirited open-source brother

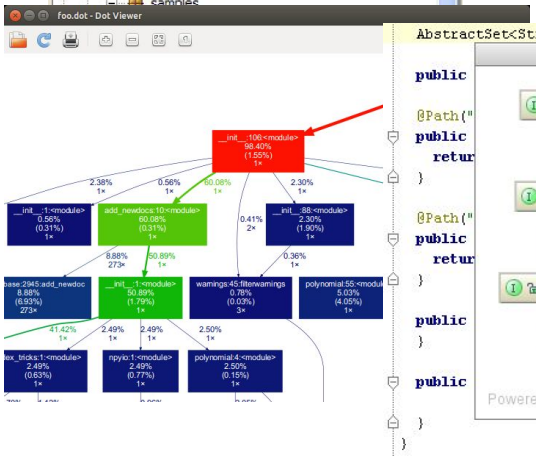
  This talk is about the technology behind these engines

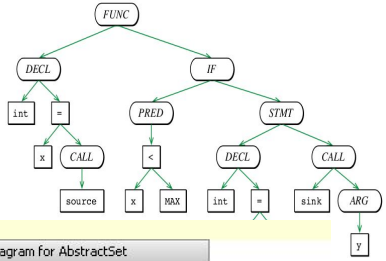


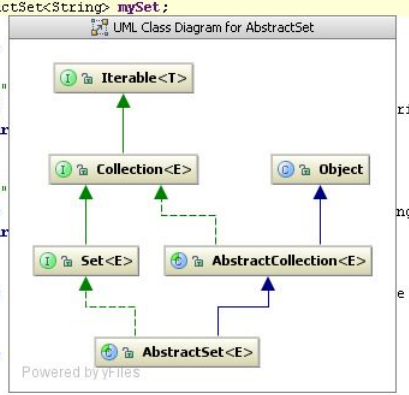
Low level graph representations of programs

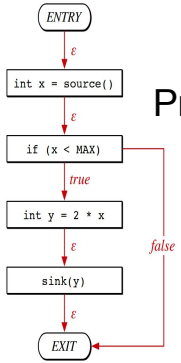
- Each graph provides a different perspective on the code
- **Can we merge them?**

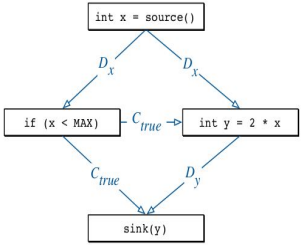




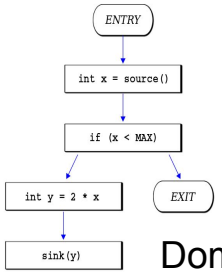








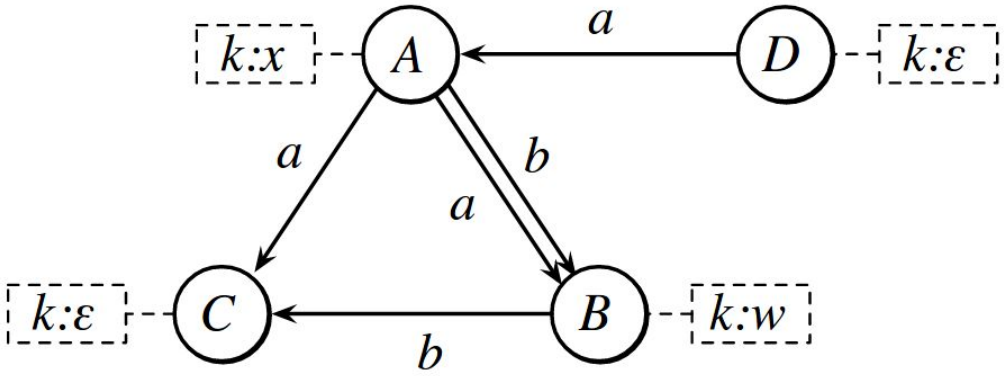
Program dependence graphs



Dominator tree

Combining graphs with “Property Graphs”

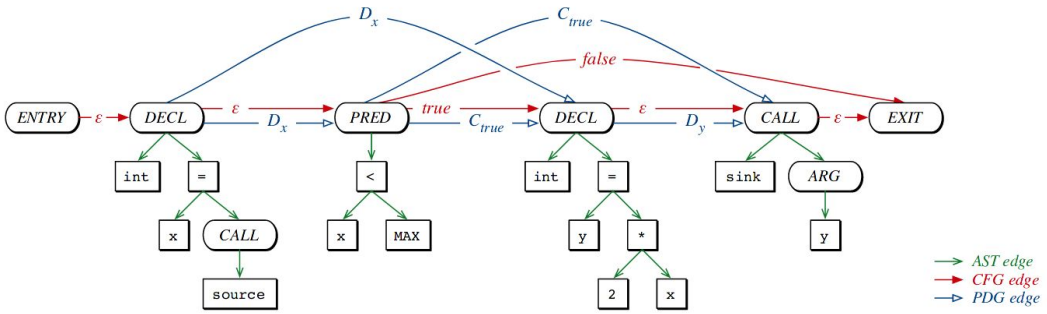
- *“A property graph is a directed edge-labeled, attributed multigraph”*
- Attributes allow data to be stored in nodes/edges
- **Edge labels allow different types of relations to be present in one graph**



Code Property Graph (2014) - rudimentary concept presented in

Modeling and Discovering Vulnerabilities with Code Property Graphs

Fabian Yamaguchi*, Nico Golde†, Daniel Arp* and Konrad Rieck*
*University of Göttingen, Germany
†Qualcomm Research Germany

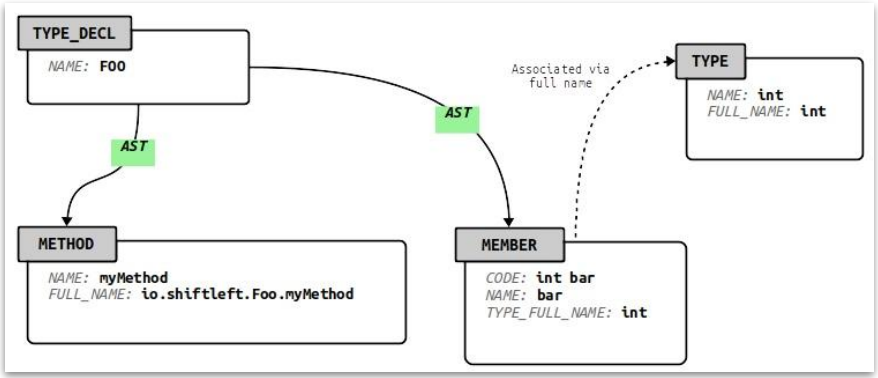


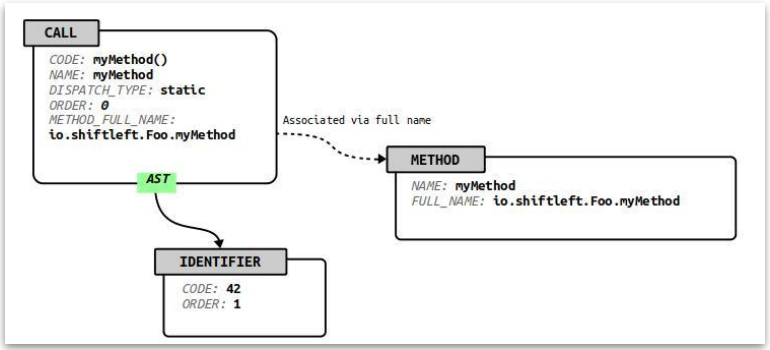
Specification - Key Design Ideas

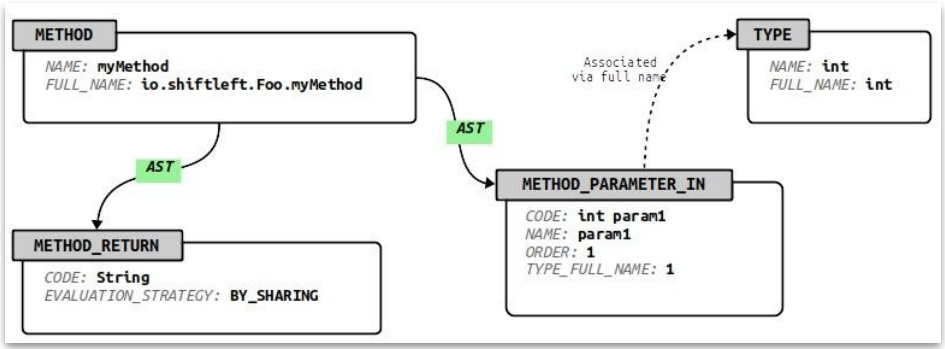
- Specification that works over programming languages
- Delay as much of the graph creation to second-stage
- Provide generic representation for core programming language concepts
  - Methods/Functions
  - Types
  - Namespaces
  - Instructions
  - Call sites
- Encode control flow structures only via a control flow graph
- Model only local program properties and leave global program representations for later analysis stages

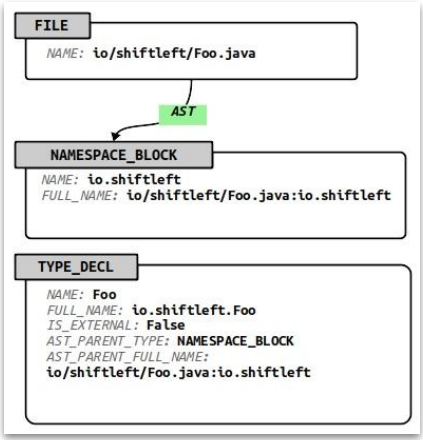
https://github.com/ShiftLeftSecurity/codepropertygraph

OSS Specification for first-stage code property graph



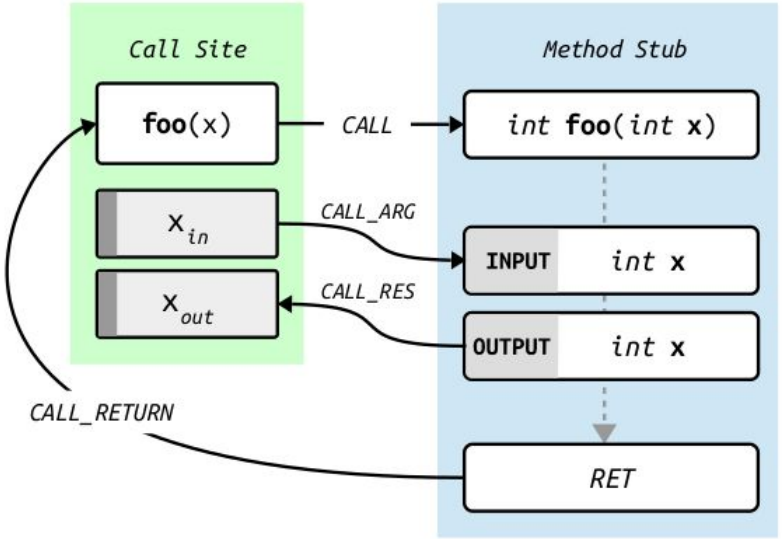




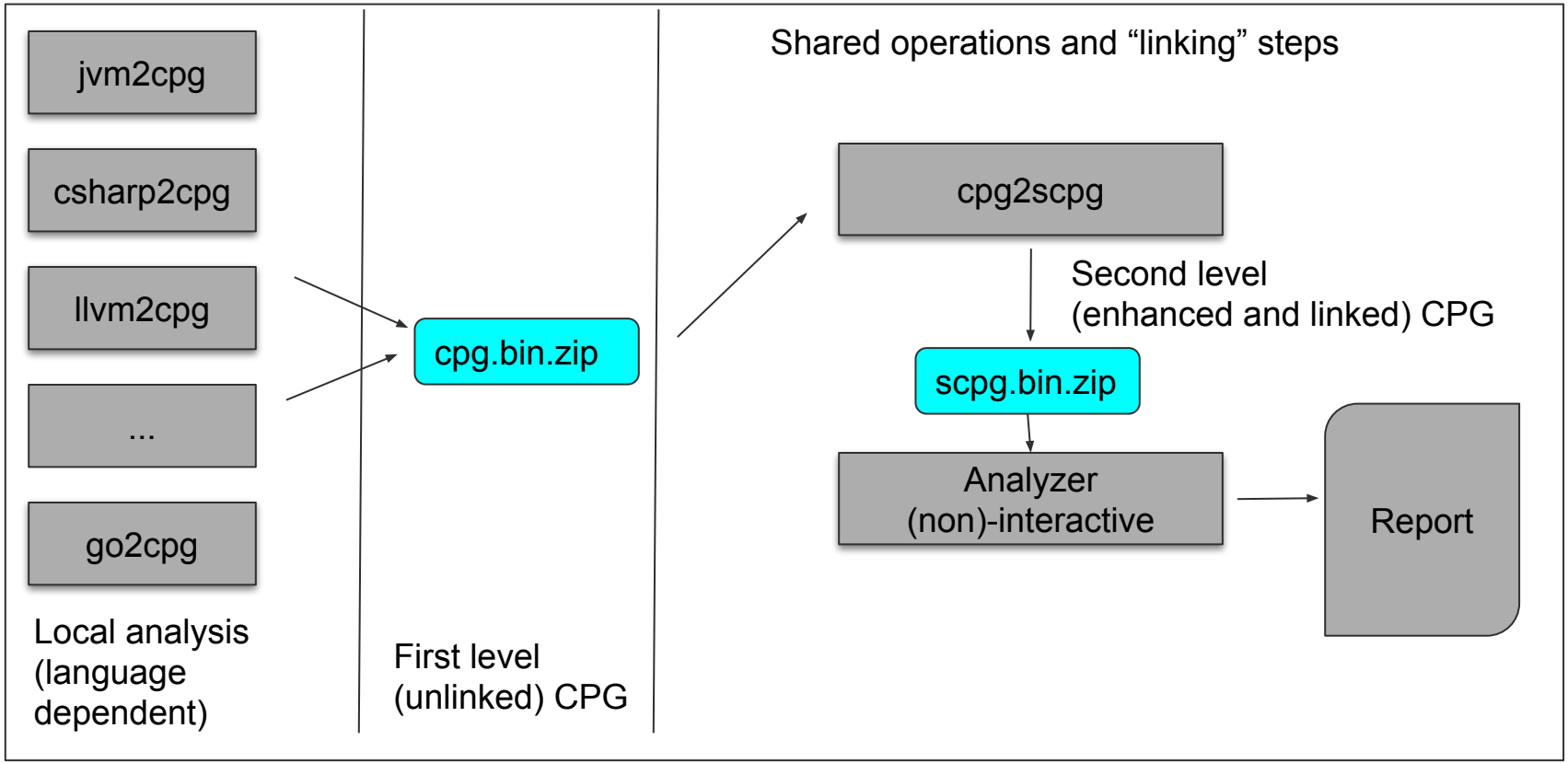


A “container” for code over arbitrary instruction sets

- Define only a common format for representing code
- Allow arbitrary instruction set (given by semantics) as a parameter
- Represent all code using only
  - call sites and method stubs
  - call edges, and control flow edges
  - data-flow semantics via data flow edges

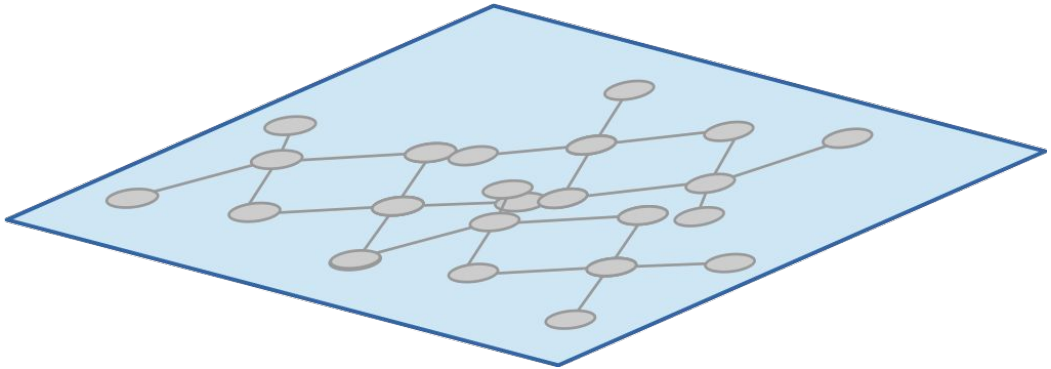


Second stage: “linking”



Base layer of the code property graph

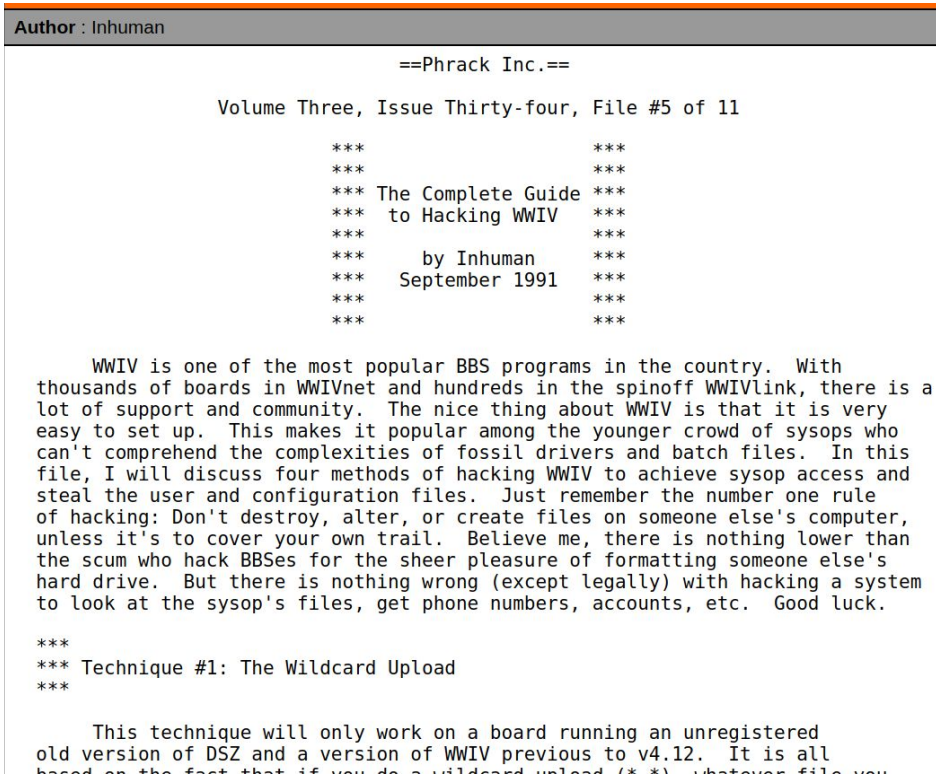
- Production quality version of 2014 code property graph
- Language-independent intermediate representation of control-flow and data-flow semantics
- Interprocedural, flow-sensitive, context-sensitive, field-sensitive data-flow tracker available that operates on this representation
- Heuristics and street smarts to terminate in < 10 minutes



ShiftLeft

This is already pretty powerful...

Typical vulnerability type associated with Zip files



Author : Inhuman

==Phrack Inc.==

Volume Three, Issue Thirty-four, File #5 of 11

*** *** *** The Complete Guide *** *** to Hacking WWIV *** *** *** by Inhuman *** *** September 1991 *** *** *** *** ***

WWIV is one of the most popular BBS programs in the country. With thousands of boards in WWIVnet and hundreds in the spinoff WWIVlink, there is a lot of support and community. The nice thing about WWIV is that it is very easy to set up. This makes it popular among the younger crowd of sysops who can't comprehend the complexities of fossil drivers and batch files. In this file, I will discuss four methods of hacking WWIV to achieve sysop access and steal the user and configuration files. Just remember the number one rule of hacking: Don't destroy, alter, or create files on someone else's computer, unless it's to cover your own trail. Believe me, there is nothing lower than the scum who hack BBSes for the sheer pleasure of formatting someone else's hard drive. But there is nothing wrong (except legally) with hacking a system to look at the sysop's files, get phone numbers, accounts, etc. Good luck.

***
*** Technique #1: The Wildcard Upload
***

This technique will only work on a board running an unregistered old version of DSZ and a version of WWIV previous to v4.12. It is all

- **Entry names of archives must be checked for control characters (e.g., “../”) to avoid path traversal**
- First presented in phrack in ‘91
- Reported again with shiny logo as “Zip-Slip” by startup in 2018
- https://www.youtube.com/watch?v=Ry_yb5Oipq0 for details

How does this surface in Java code?

- Names of archive entries are used unchecked as path names when opening a file
- Concrete example: return values of **ZipEntry.getName** are used as path names in a **java.io.File constructor**

```
File f = new File(entry.getName())
```

Sufficient to model control flow, data flow, and method invocations correctly.

Query

```
ocular> val src = cpg.method.fullName("java.util.zip.*getName.*").methodReturn
  val snk = cpg.method.fullName(".*File.<init>").parameter.index(1)
  snk.reachableBy(src).flows.passesNot(".*escape|sanitize|contains.*")
```

Typical XML-related vulnerability: XXE

- **“XML External Entity Processing” vulnerability**
- **The attacker controls an XML file processed by the application and it does NOT disable resolving of external entities**
- If this is given, the attacker can provide an XML file that reads and sends sensitive files to the attacker as part of the parsing process
- OWASP cheat sheet says:

XMLInputFactory (a StAX parser)

StAX parsers such as `XMLInputFactory` allow various properties and features to be set.

To protect a Java `XMLInputFactory` from XXE, do this:

```
xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false); // This disables DTDs entirely for that factory
xmlInputFactory.setProperty("javax.xml.stream.isSupportingExternalEntities", false); // disable external entities
```

Query

```
ocular> val snk = cpg.method.fullName(".*XMLStreamReader.*next.*")
                .parameter.index(0)
  val src =
cpg.method.fullName(".*XMLInputFactory.*create.*").methodReturn

  snk.reachableBy(src).flows.l.andNot{
    // No calls to setProperty
    cpg.method.name("setProperty").fullName(".*XML.*").callIn.l
  }
```

ShiftLeft

… but not powerful enough.

Higher level abstractions - Example: Microservice

Humans tend to describe vulnerabilities on much higher levels of abstraction!

“Form”

“Route”

```
<form action=”/foo/bar”>
<input type=”hidden”
name=”csrfprotect”>
...
<input name=”foo”
htmlEscaped=”true”>
</form>
```

“Input field”

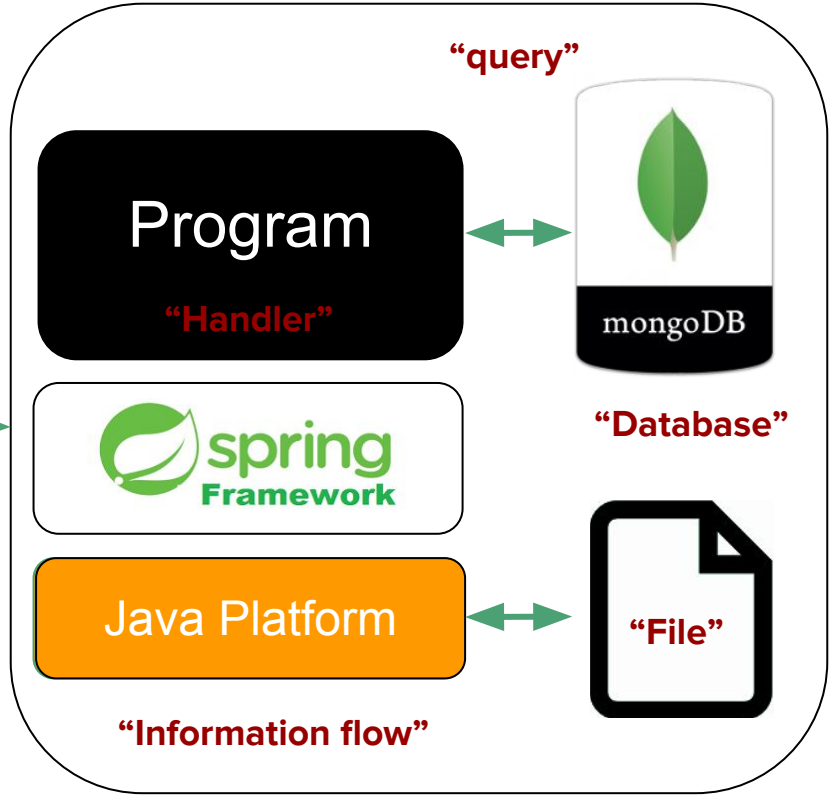
“HTML attribute”

“request”

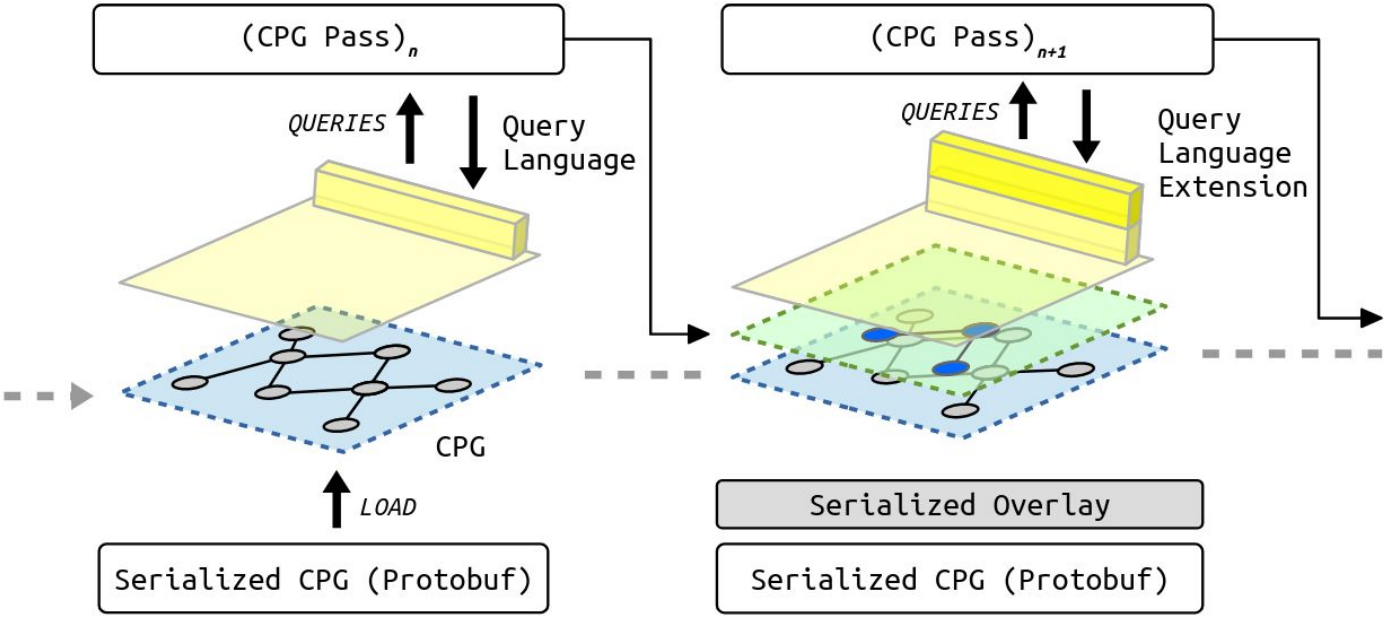
GET /foo/bar?...

{“foo” : “bar”, ...}

“Json response”



Higher-level abstractions



Handlers/Routes (Vert.x framework)

```
@Override
public void start(Future<Void> done) {
  // Create a router object.
  Router router = Router.router(vertx);
  router.get("/health").handler(rc -> rc.response().end("OK"));

  // This is how one can do RBAC, e.g.: only admin is allowed
  router.get("/greeting").handler(ctx ->
    ctx.user().isAuthorized("booster-admin", authz -> {
      if (authz.succeeded() && authz.result()) {
        ctx.next();
      } else {
        log.error("AuthZ failed!");
        ctx.fail(403);
      }
    }));
...
}
```

- Lambda expression flows into first argument of `handler`
- Route (as string) flows into first argument of `get`.
- Return value of `get` flows into instance parameter of `handler` (or the other way around)

Extraction of handlers/routes

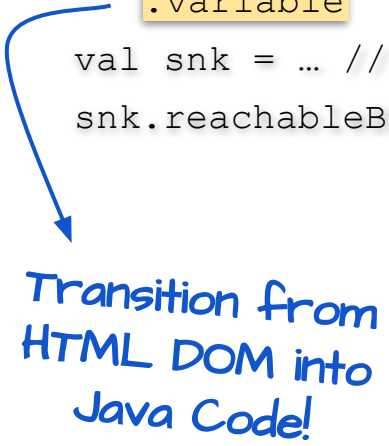
```
val HANDLER_METHOD_NAME = "io.vertx.ext.web.Route.handler:io.vertx.ext.web.Route(io.vertx.core.Handler)"

val firstParamOfHandler = cpg.method.fullNameExact(HANDLER_METHOD_NAME).parameter.index(1)
val flowsOfMethodRefsIntoHandler = firstParamOfHandler.reachableBy(cpg.methodRef).flows.l
val handlerNameRoutePairs = flowsOfMethodRefsIntoHandler.map { flow =>
        val handlerName  = flow.source.node.asInstanceOf[nodes.MethodRef].methodInstFullName
        val route = routeForFlowIntoHandler(flow)(handlerName, route)
        (handlerName, route)
}
def routeForFlowIntoHandler(flow : NewFlow) = {
        val getOrPostSource = cpg.method.fullName("io.vertx.*Router.*(get|post):.*").methodReturn
        Flow.sink.callsite
        .map(x => x.asInstanceOf[nodes.Call]).get
        .start.argument(0).reachableBy(getOrPostSource)
        .flows.source.l
        .flatMap(x => x.callsite.get.asInstanceOf[nodes.Call].start.argument(1).code.l)
        .headOption.getOrElse("")
}
```

Creating DOM Trees from JSP files in a CPG Pass

```
class JspPass(cpg: Cpg) extends CpgPass(cpg) {

override def run(): Iterator[DiffGraph] = {
  Queries.configFiles(cpg, ".jsp").toList.sorted.map {
   case (filename, html) =>
     val diffGraph   = new DiffGraph
     val rootCpgNode =
           createAndAddTreesRecursively(
              Jsoup.parse(html).root(), None, diffGraph)

cpg.configfile.nameExact(filename).headOption.foreach(
       diffGraph.addEdgeFromOriginal(_,
              rootCpgNode, EdgeTypes.CONTAINS))
     diffGraph
  }.toIterator
 }

def createAndAddTreesRecursively(node: Node,
maybeParent: Option[NewDomNode],
    diffGraph: DiffGraph): NewDomNode = {
  implicit val dstGraph: DiffGraph = diffGraph
   val attributes = node.attributes().asScala.map { attr =>
      NewDomAttribute(attr.getKey, attr.getValue) }.toList

   // Add node + attribute nodes
   val newDomNode = NewDomNode(node.nodeName,
attributes)
   diffGraph.addNode(newDomNode)
   newDomNode.start.store()

   // Add AST edge from parent to node
   maybeParent.foreach(
     diffGraph.addEdge(_, newDomNode, EdgeTypes.AST)
   )
   node.childNodes.asScala.foreach(
         createAndAddTreesRecursively(_,
                Some(newDomNode),
                diffGraph))
   newDomNode
  }
}
```

Query for XSS detection using DOM + Bytecode

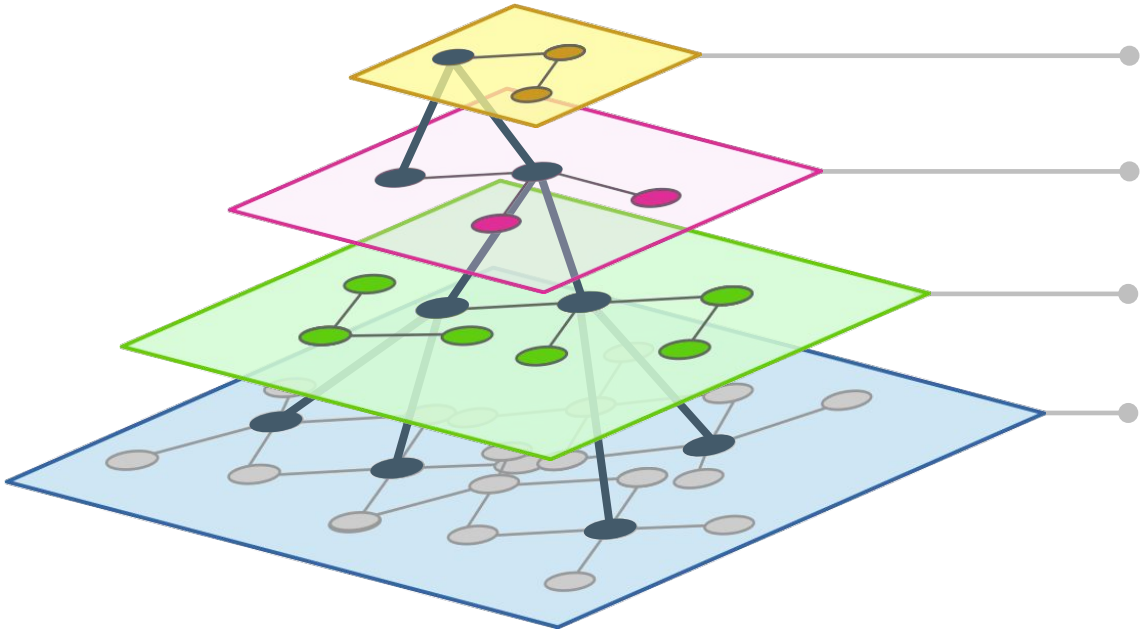
```
ocular> val src = cpg.form.method("post").input
      .filterNot(_.attribute.name("htmlEscaped"))
      .variable
  val snk = … // some output routine in HTML
  snk.reachableBy(src).flows.l
```



Outside information, business logic, and FP reduction

- Literature deals a lot with FPs due to model limitations, e.g., overtainting of collections, conservative call graphs, ...
- In practice, most FPs result from context information, e.g., information about the business logic, that you cannot deduce from the code alone:
  - “This is an internal service that only our admin uses”
  - “Without first convincing the authentication server, this code would never be executed”
  - “Due to $aliens, this integer is always 5 and thus cannot be negative”
- Ability to model the $aliens part is crucial to reduce false positives
- **We do this mostly via passes that tag the graph**

Code Property Graphs Today



Vulnerabilities

Multiple domain-specific layers [...]

Call graph, type hierarchy, **data flows**, configurations, dependencies

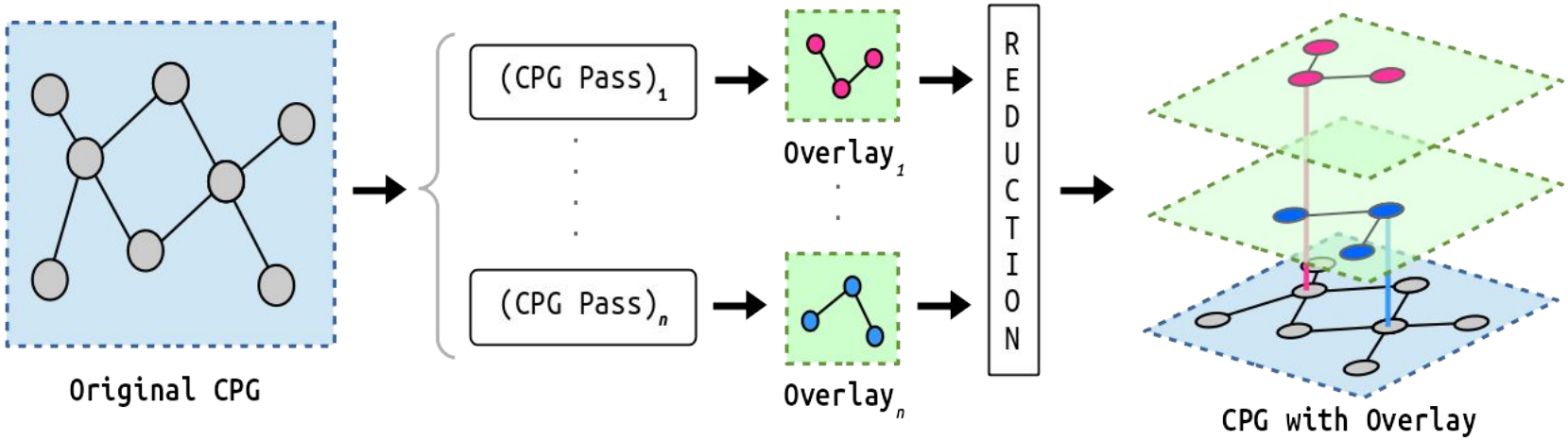
Base layer - low level local program representations: syntax, control flow, methods, types.

Scaling static analysis

- Summaries
  - Scaling static analysis requires “summaries” of program behavior (in order to skip duplicate calculation of facts, e.g., for library methods)
  - Calculating summaries for data flow is common practice
  - **Upper layers of the CPG generalize the concept of a summary**
- Parallelism
  - Processors aren’t getting much faster, but you’re getting more and more cores.
  - Literature has very little to say about multiple cores, let alone multiple cloud instances
  - **CPG passes are a design with parallelism in mind**

Designed for distributed computing

- Passes can be run in a sequence like the passes of a compiler
- The design also allows to run independent passes in parallel though!



ShiftLeft

End of presentation
@ShiftLeftInc @fabsx00
https://shiftleft.io
https://joern.io