May 2018

Code Property Graphs

A modern queryable representation for code

Fabian Yamaguchi
ShiftLeft Inc.

Strata

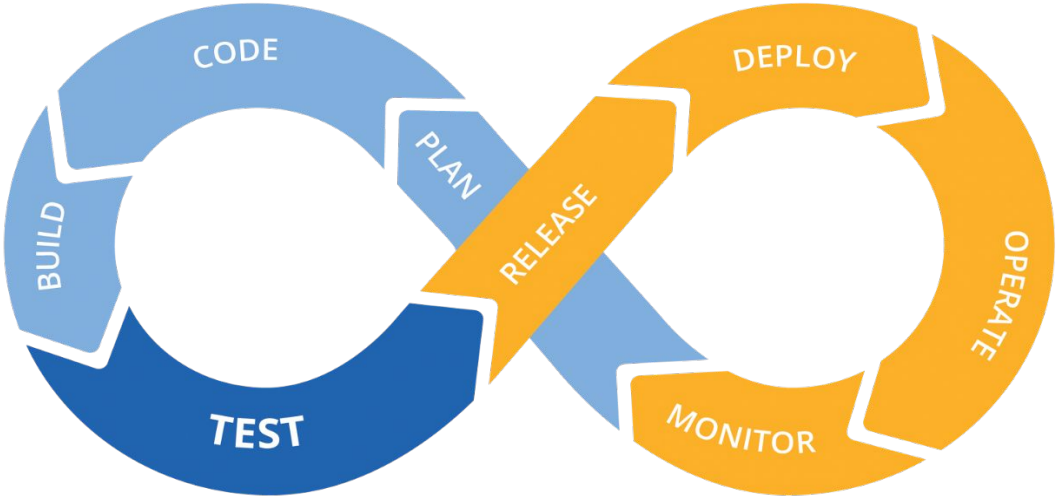
ShiftLeft

About the speaker

- In IT Security for 10+ years - consulting and research
- My first talk at a Data-conference :)
- PhD thesis: **“Pattern-based Vulnerability Discovery”** on using unsupervised machine learning and graph databases for vulnerability discovery
- Chief scientist at ShiftLeft Inc.

Shifting security to the left

- Modern development processes are optimized for fast feature delivery
- Cloud services: multiple deployments a day
- Each deployment is a security-related decision
- **Enforce code-level security policies automatically!**

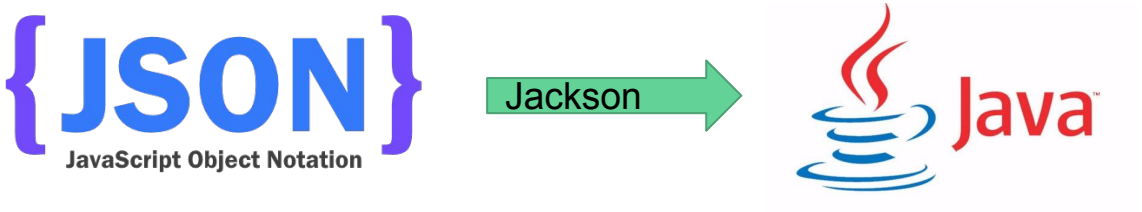


Leaks of sensitive information

Example: enforce a policy for propagation of sensitive data: **“Environment variables must not be written to the logger”**

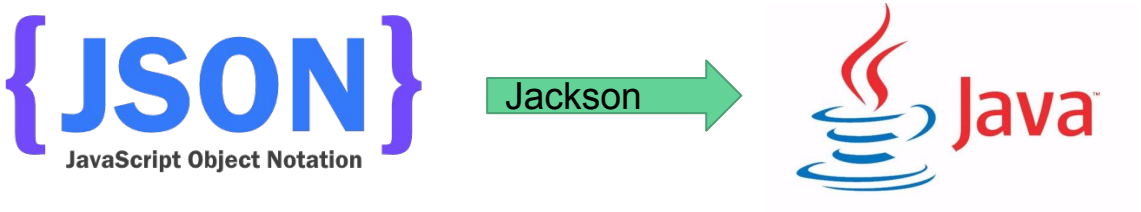
```
public void init() {
    String user = env.getProperty(“sfdc.uname”);
    String pwd = env.getProperty(“sfdc.pwd”);
    ...
    if (!validLogin(user, pwd)) {
        ...
        log.error("Invalid username/password: {}/{}",
                  user, pwd);
    }
}
```

Jackson deserialization vulnerabilities



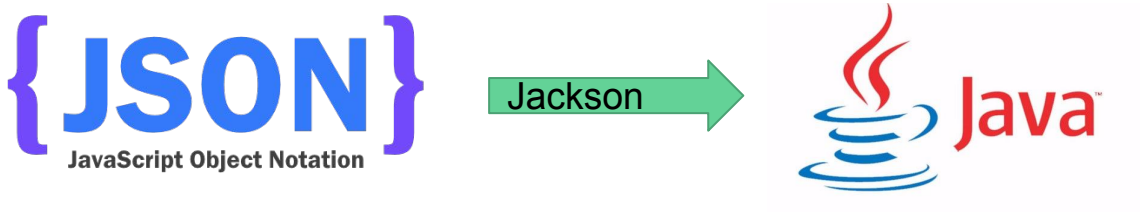
- Critical code execution vulnerabilities. A program is vulnerable if
  1. a vulnerable version of jackson-databind is used
  2. attacker controlled input is deserialized in a call to “readValue”
  3. “Default typing” is enabled globally via a call to “enableDefaultTyping”

Example: Jackson deserialization vulnerabilities




```
50
51    private static Account deserialize(Request request)
52        throws IOException, JsonParseException, JsonMappingException {
53      try {
54        return deserializer.readValue(request.body(), Account.class);
55      } catch (Exception any) {
56        log.warn("Unexpected exception deserializing content: {}", any.getClass());
57        return null;
58      }
59    }
```

Example: Jackson deserialization vulnerabilities




```
private static ObjectMapper deserializer = new ObjectMapper().enableDefaultTyping();
private static ObjectMapper serializer = new ObjectMapper();
private static AccountStore accounts = new AccountStore();
```

Detecting this particular vulnerability types requires modeling program dependencies, syntax, data flow and method invocations.

There you go

```
> cpg.dependency.name(".*jackson-databind.*").version("2.8..*")
.and{
  cpg.call.name(“.*enableDefaultTyping.*”)
}
.and{
   cpg.method.name(“.*readValue.*”).parameter.index(1).reachableBy(
   cpg.parameter.evalType(".*HttpServletRequest.*"))
}.location
```

Data structure at the heart of the approach

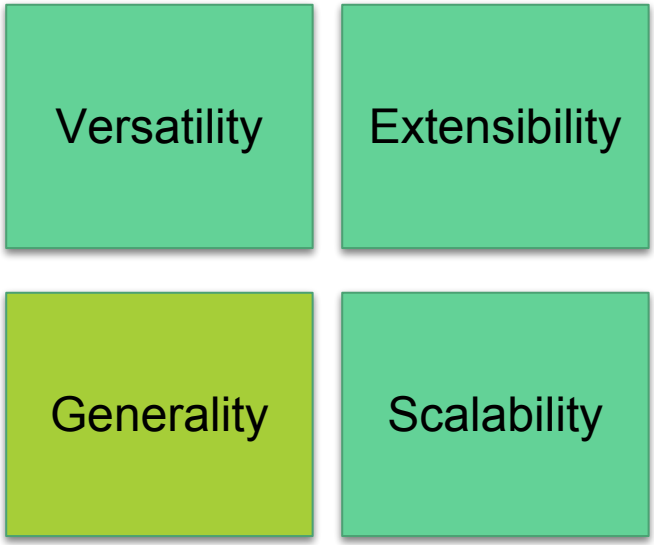
Code Property Graph

```
cpg.method.name("sink").parameter
.reachableBy(cpg.method.name("source"))
```

Database

query-language for code analysis

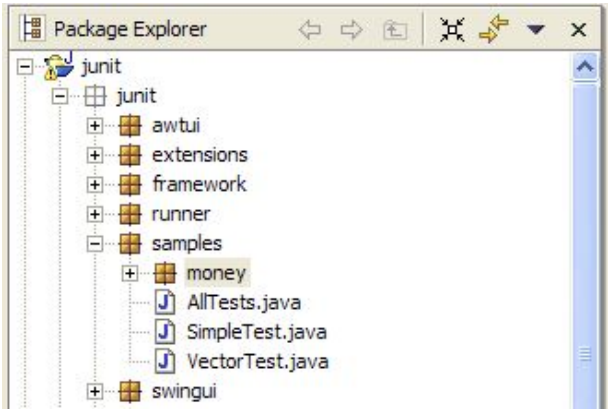
Primary challenges in design of the data structure



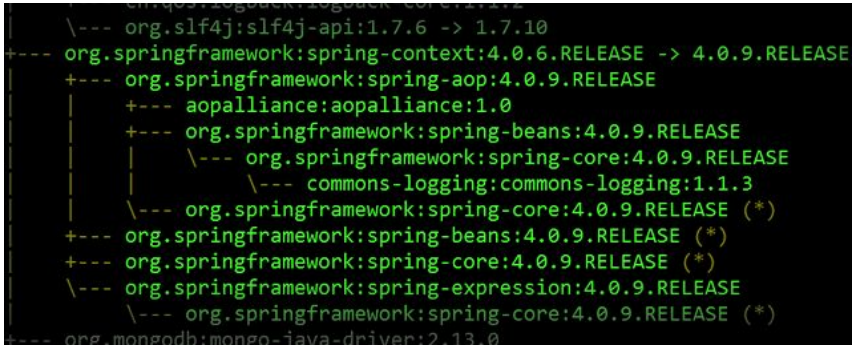
Challenge #1: Versatility

How do we provide a code representation multifaceted enough to deal with the complex combinations of program properties that make it vulnerable?

Program structure (Modules, Packages, Classes, ...)

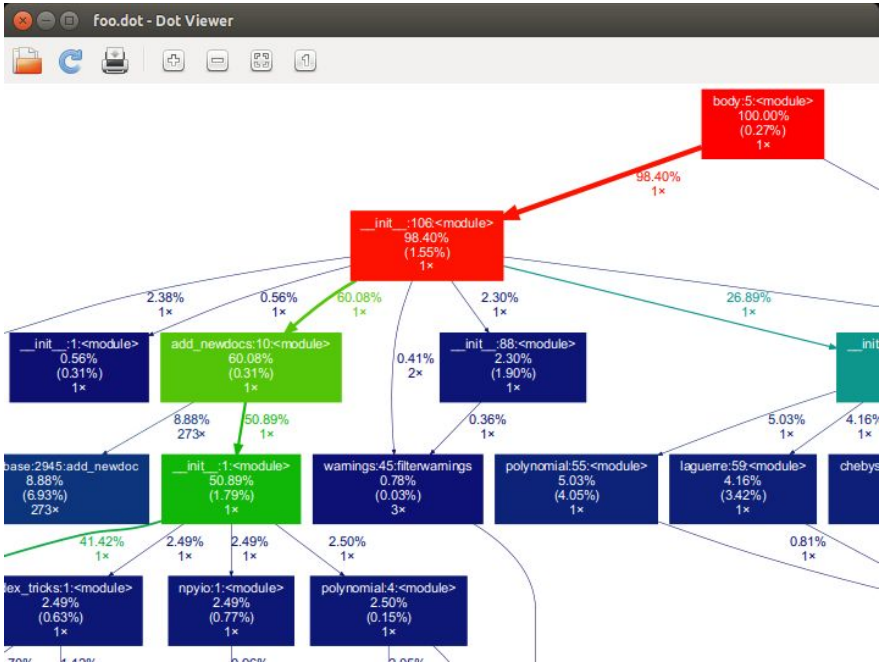


From eclipse.org: Package explorer in Eclipse IDE

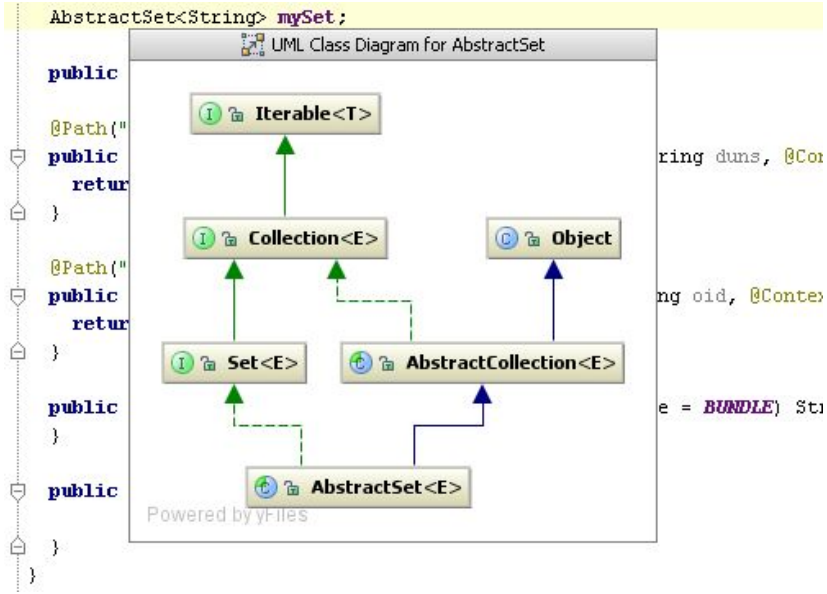


From mykong.com: dependency tree created by the Gradle build tool

Calling relations and type hierarchies

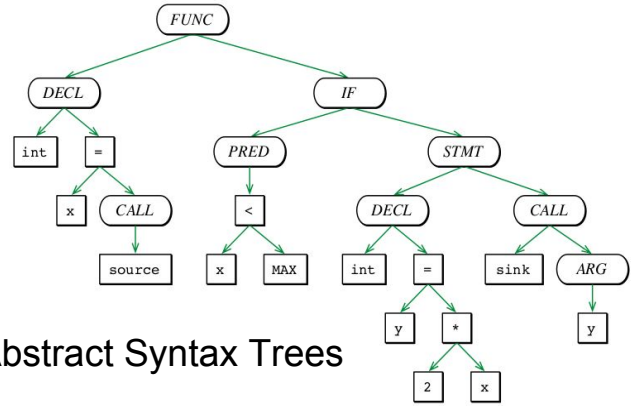


From codeyarns.com: visualization of GNU Profiler output as a call graph using gprof2dot

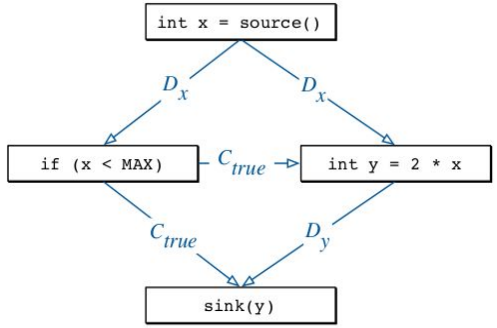


From jetbrains.com: Intellij showing a type hierarchy

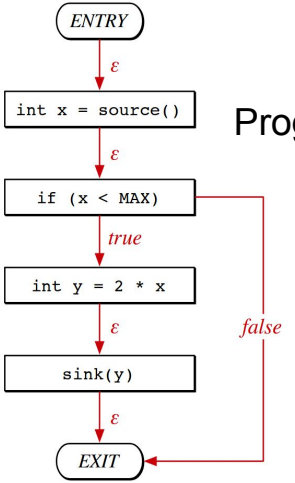
Internal program representations in compilers



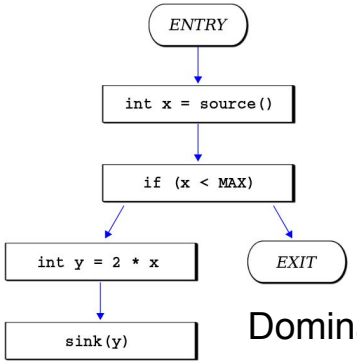
Abstract Syntax Trees



Program dependence graphs



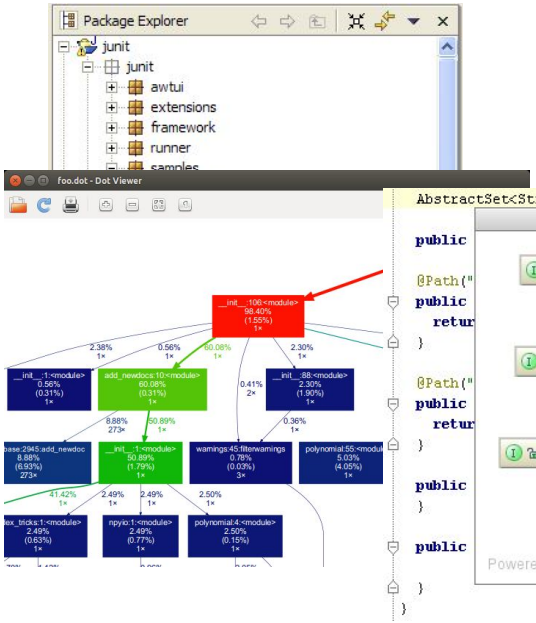
Control flow graphs

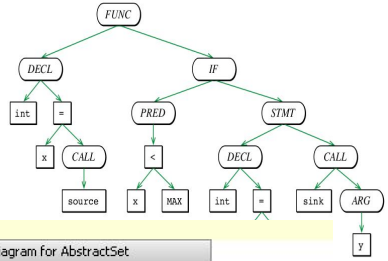


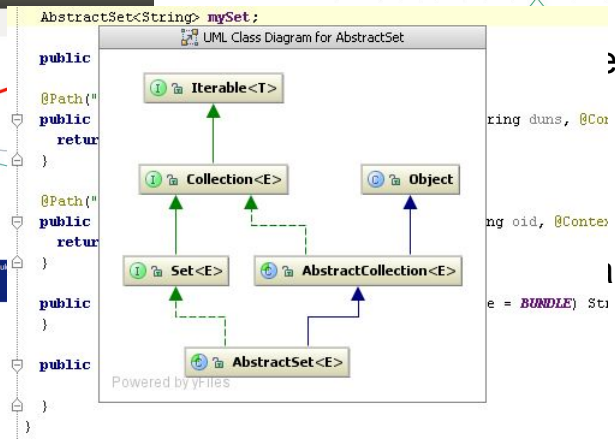
Dominator tree

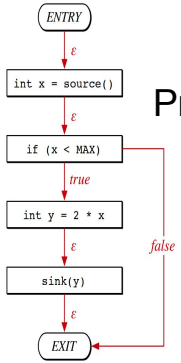
All graphs!

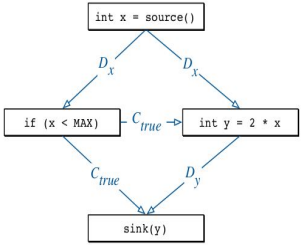
- Each graph provides a different perspective on the code
- **Can we merge them?**



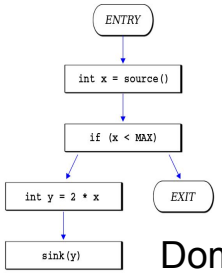








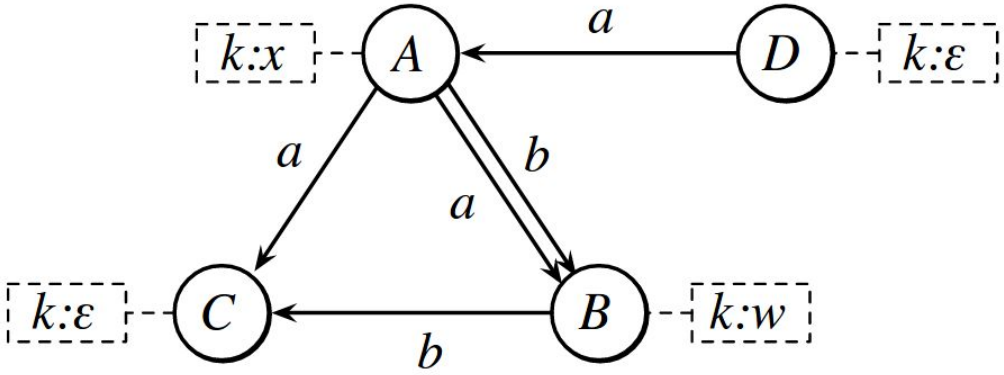
Program dependence graphs



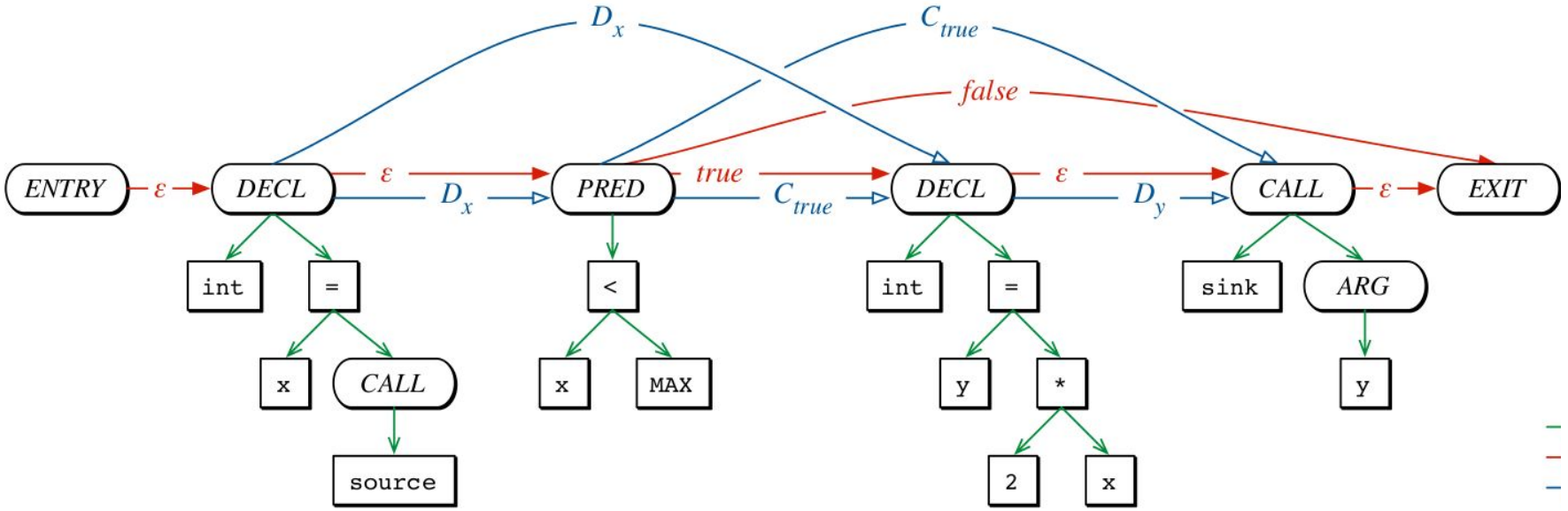
Dominator tree

Combining graphs with “Property Graphs”

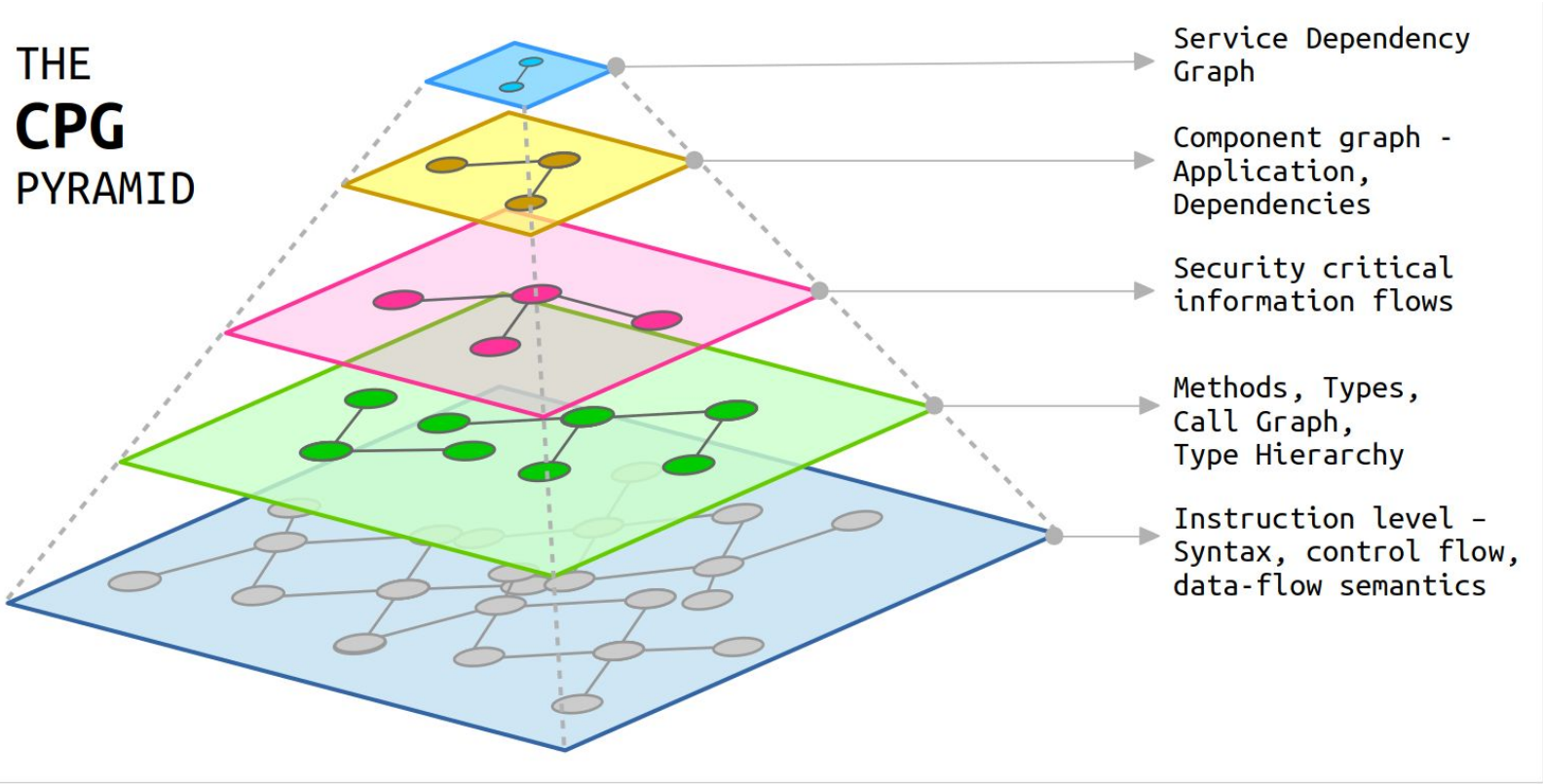
- *“A property graph is a directed edge-labeled, attributed multigraph”*
- Attributes allow data to be stored in nodes/edges
- **Edge labels allow different types of relations to be present in one graph**



Code Property Graphs - Basic concept (2014)



Code Property Graphs Today



Graph databases and Apache Tinkerpop

Graph databases that support Tinkerpop

neo4j

TITAN

OrientDB®

JanusGraph

Azure Cosmos DB

blazegraph™

KeyLines

IBM Graph™

GRAKN.AI

DATASTAX

Stardog

LINKURIOUS
VISUALIZE GRAPH DATA EASILY

Gremlin: Tinkerpop’s query language

- Domain-specific language (DSL) for querying graphs
- Language variants: Gremlin-groovy, Gremlin-Ruby, ...

```
g.V()
.has(“type”, “METHOD”)
.out(“IS_PARAM_OF”)
.value(“name”)
```

Gremlin-Scala

- It is a graph in the background, the challenge is to hide that
- Sanity checking/completion of queries BEFORE execution
- **Requires statically typed language**
- A good match: Scala - statically typed but concise like a scripting language
- Features added for ShiftLeft
  - API to create custom DSLs on top of Gremlin-Scala
  - Type-safe queries via generated keys

```
g.V()
.has("type", "METHOD")
.out("IS_PARAM_OF")
.value("name")
```

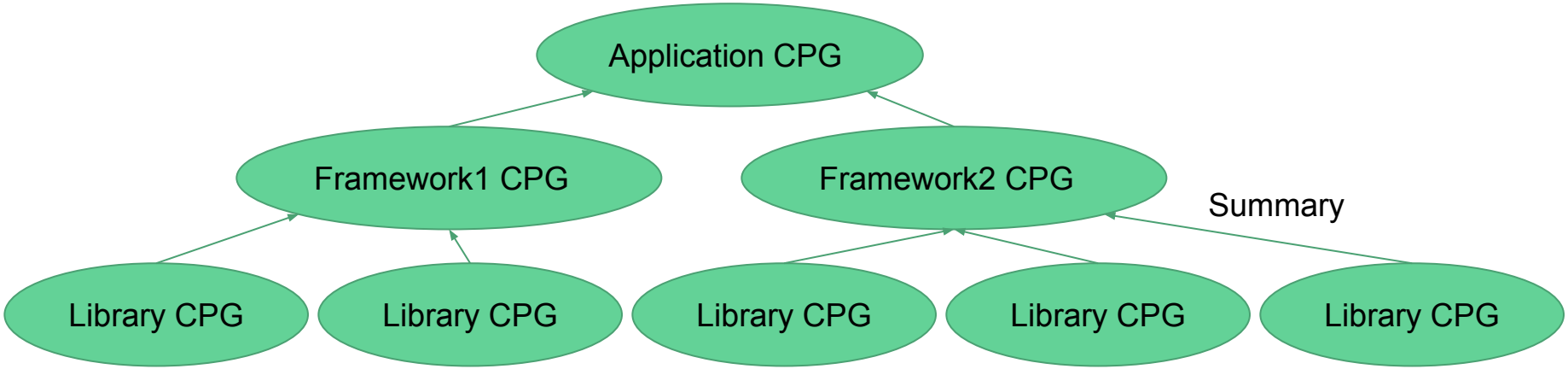
```
Cpg
.method
.parameter
.name
```

Challenge #2: Scalability

- Just use a distributed graph DB! - Not quite
- Unfortunately, query performance will suffers each time you need to transition from data stored on one machine, to data stored on another
- **Graphs need to be partitioned such that**
  - we minimize transitioning between partitions in queries (“edge-cuts”)
  - there isn’t one partition holding all the important nodes and the others are rarely used
- No free lunch here: know your data to get performance

Partitioning code property graphs

- **Partition graphs based on underlying natural code structure**
- **Post order traversal on dependency tree to create summaries, using summaries of dependencies in calculation of depender's CPG**



Tinkergraph in-memory graph database

- Created for automated testing and as a reference implementation
- Memory characteristics not relevant in its design
- Map<String, Property> for properties
- Map<String, Set<Edge>> from edge types to edges for incoming edges
- Map<String, Set<Edge>> from edge types to edges for outgoing edges
- Rough estimate for 1M nodes, 10M edges

  3GB for the graph structure alone (without data!)

ShiftLeft Tinkergraph

- Main idea: a graph schema is good practice anyway.
- Exploit knowledge of the schema for more memory efficient representation
  - Replace HashMap<String, Property> with member variables, e.g., “String name; int age;”
  - Replace HashMap<String, Set<Edge>> by Set<EdgeType> for each edge type
- **70% less memory usage**
- Approximately 15% performance improvement on queries (due to fewer hash table lookups)

Try it out: https://github.com/ShiftLeftSecurity/tinkergraph-gremlin

Summary for graph databases

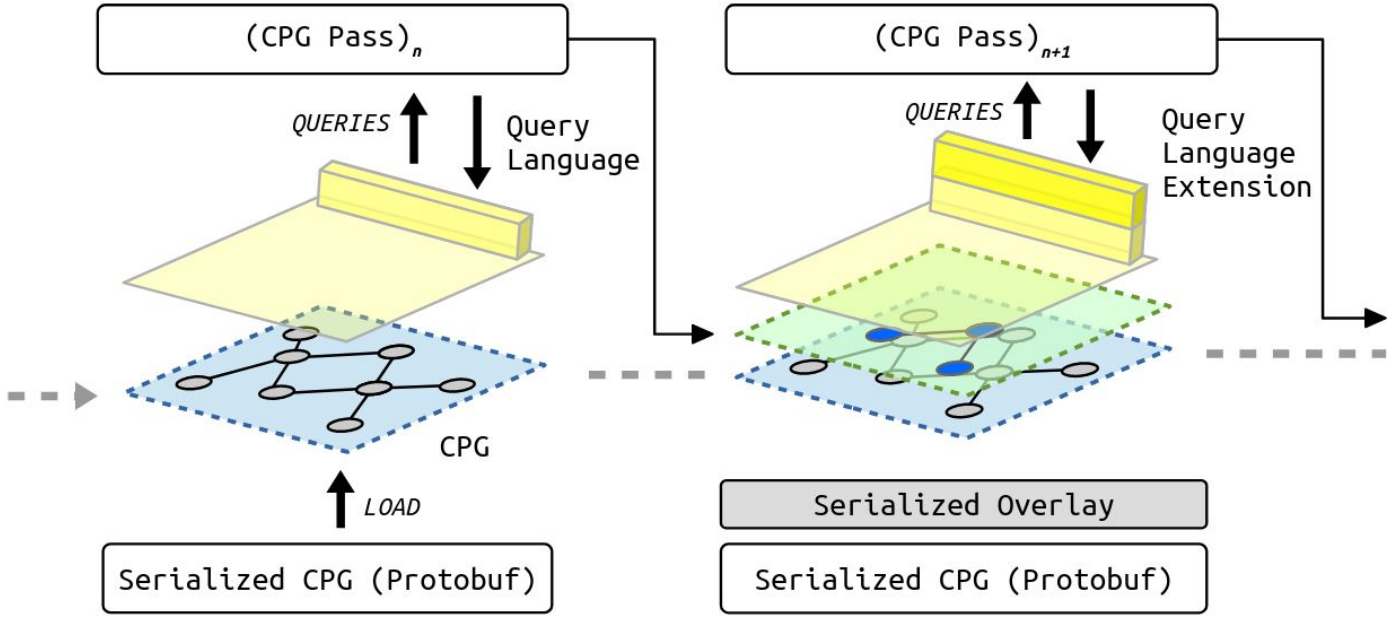
- Existing APIs and graph traversal languages provided a great basis
- Program code can be naturally split into partitions for distributed storage
- Optimized TinkerGraph so that graphs for program components can be held in RAM
- Overlay concept allows us to selectively load views on the graph and thus decrease memory usage even further in practice

Challenge #3: Extensibility

- Working with the feedback of customers and other teams shows:
  There's always a piece of information that somebody needs and that you never thought anyone would ever need. Maybe for a longer time, maybe for a week.
- Need to enable extensibility of the code property graph to
  - add new information to the graph
  - provide DSL elements for the new information
  - remove information that turned out to be irrelevant
- … without degenerating the graph

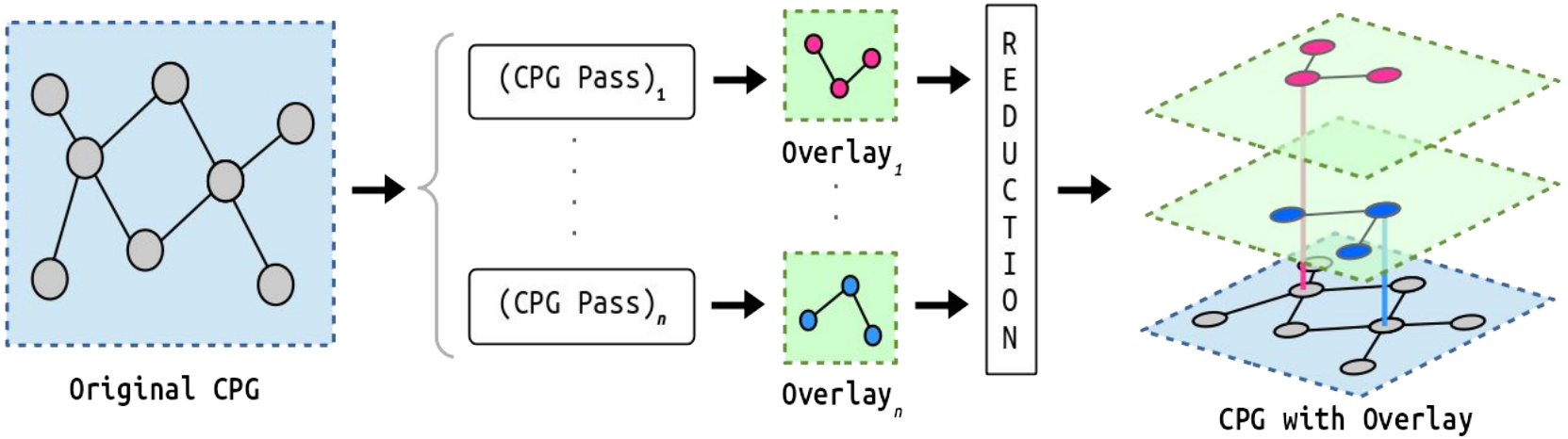
CPG Passes and Overlays

- Idea: create a limited base graph and calculate additional info via plugins that create “graph overlays”



CPG Passes and Parallelism

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



Allowing passes to augment the query language

- Key: Scala's "Pimp my library" pattern: pattern to extend existing classes without modifying the bytecode
- Main trick: **define implicit conversion from library's classes to your own (modifiable) classes**

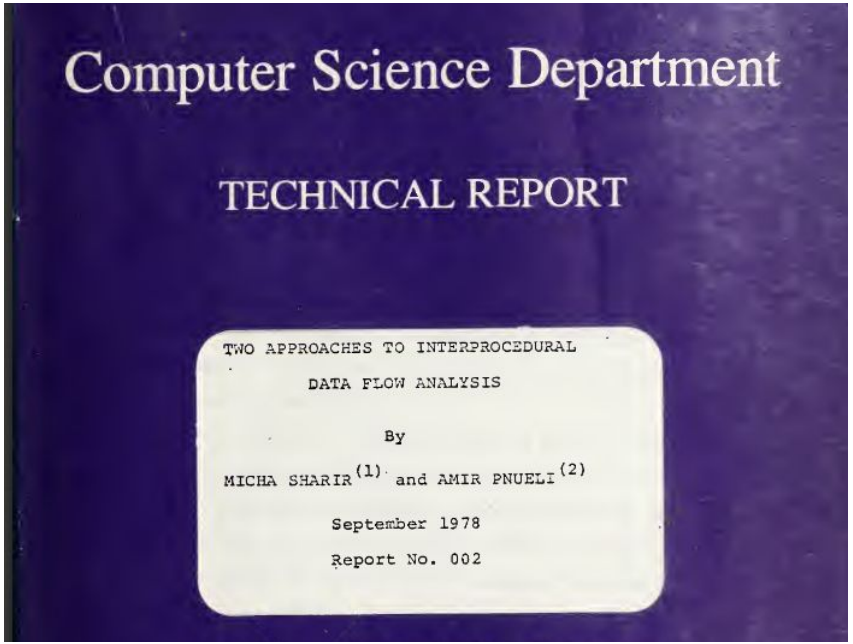
```
// Declare implicit conversion from LibraryClass to MyClass
implicit def enrichClass(method : Method) = new MyClass(method)

class MyClass(method :Method) {
    // Define new step callable on Method
    def summary = { method =>
        method.[...]
    }
}
```

See post by Martin Odersky:
https://www.artima.com/weblogs/viewpost.jsp?thread=179766

Challenge #4: Generality - Language independence

Back to the roots - functional approach to data flow

- Published in 1978
- Foundation for most of today's practical approaches
- Not specific to any language
- Requires
  - **… control flow graph** for all functions
  - **… flow semantics** for all instructions

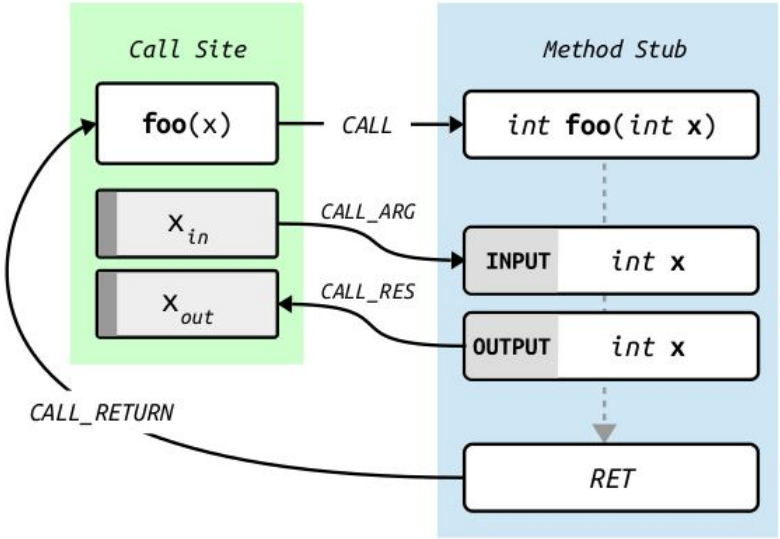


Both can be represented as graphs! :)

See "Reps et al. - Interprocedural Dataflow Analysis via Graph Reachability"

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



Language-independent querying

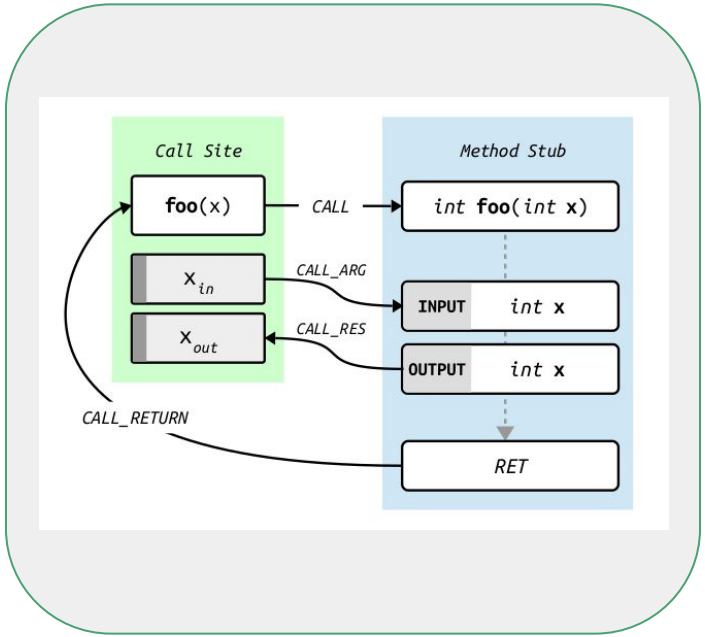
java2cpg

go2cpg

js2cpg

cs2cpg

Language frontends




```
cpg.method.name("...")
.parameter
.reachableBy(...)
```

Language-independent queries!

Conclusion

- Code property graphs: representation of code to allow complex queries based on graph database technology
- Allows identifying vulnerabilities and data leaks in particular
- Extensibility via graph overlays computed by CPG passes
- If you can partition your graphs well, it’s a good idea to do so
- Open-source in-memory graph databases for your partitioned graphs

Questions?

Fabian Yamaguchi <fabs@shiftleft.io> - Twitter: @fabsx00

ShiftLeft

Extensibility via CPG Passes

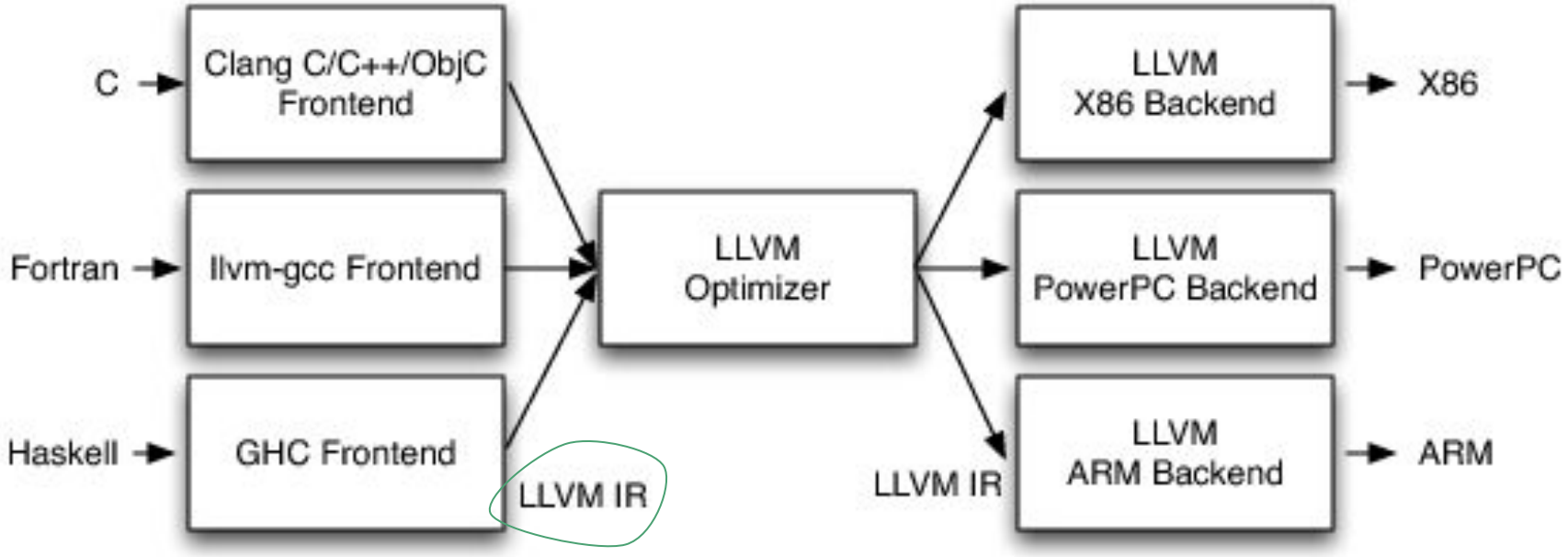
- May define new node and edge types
- May define new language elements
  - … for creating new nodes and edges as results of queries
  - .... for querying this information once the overlay has been applied
- Example passes performed on the base graph:
  - Enhancements of the CPG to speed-up subsequent heavy-duty passes
  - Heuristic determination of possibly attacker-controlled variables driven by a policy
  - Data-flow passes to determine vulnerabilities and data leaks, also driven by a policy
  - Generation of application security summaries - we call them **security profiles**

Ideal case: entire graph of component fits into RAM

- One each (storage/compute)-node we can use an on-disk graph DB
- These graph databases cache nodes/edges in RAM
- For *small* graphs, the entire graph is eventually loaded into RAM
- In-memory graph databases are the more efficient choice here
- How large can graphs be and still fit into RAM?

Challenge #4: Generality - Language independence

Classic approach: intermediate languages



From https://llvm.org

A small sample C-program

```
int main(int argc, char **argv) {
    if(argc < 2) {
        printf("Hello World\n");
    } else {
        printf("%s\n", foo(argv[1]));
    }
    return 0;
}
```

Code for `main` in LLVM IR

More news paper than code

```
define i32 @main(i32 %argc, i8** %argv) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  store i32 0, i32* %1, align 4
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = load i32, i32* %2, align 4
  %5 = icmp slt i32 %4, 2
  br i1 %5, label %6, label %8
; <label>:6                                       ; preds = %0
  %7 = call i32 (i8*, ...)
     @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @.str, i32 0, i32 0))
  br label %14
; <label>:8                                       ; preds = %0
  %9 = load i8**, i8*** %3, align 8
  %10 = getelementptr inbounds i8*, i8** %9, i64 1
  %11 = load i8*, i8** %10, align 8
  %12 = call i8* @foo(i8* %11)
  %13 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @.str.1, i32
0, i32 0), i8* %12)
  br label %14
; <label>:14                                      ; preds = %8, %6
  ret i32 0
}
```