

# Unified information-flow and points-to analysis

Neville Grech  
nevillegrech@gmail.com



University of Athens  
University of Malta

Funded by the Reach High scholars programme.  
Financed by the European Union and Government of Malta.

# Terminology (static analyses)

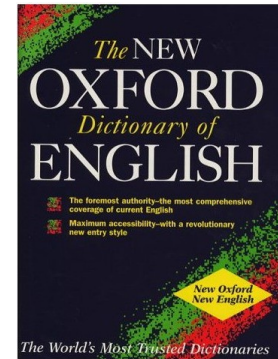
**Points-to analysis:** Which objects can a variable point to?

**Information flow analysis:**

- **Data flow analysis:** To which variables do values flow in a procedure?
- **Taint analysis:** *Which variables can be changed through user input?*
- **Program Dependency analysis:** *Which control flows and data does an program element depend on?*

# *Unified* information-flow and points-to analysis

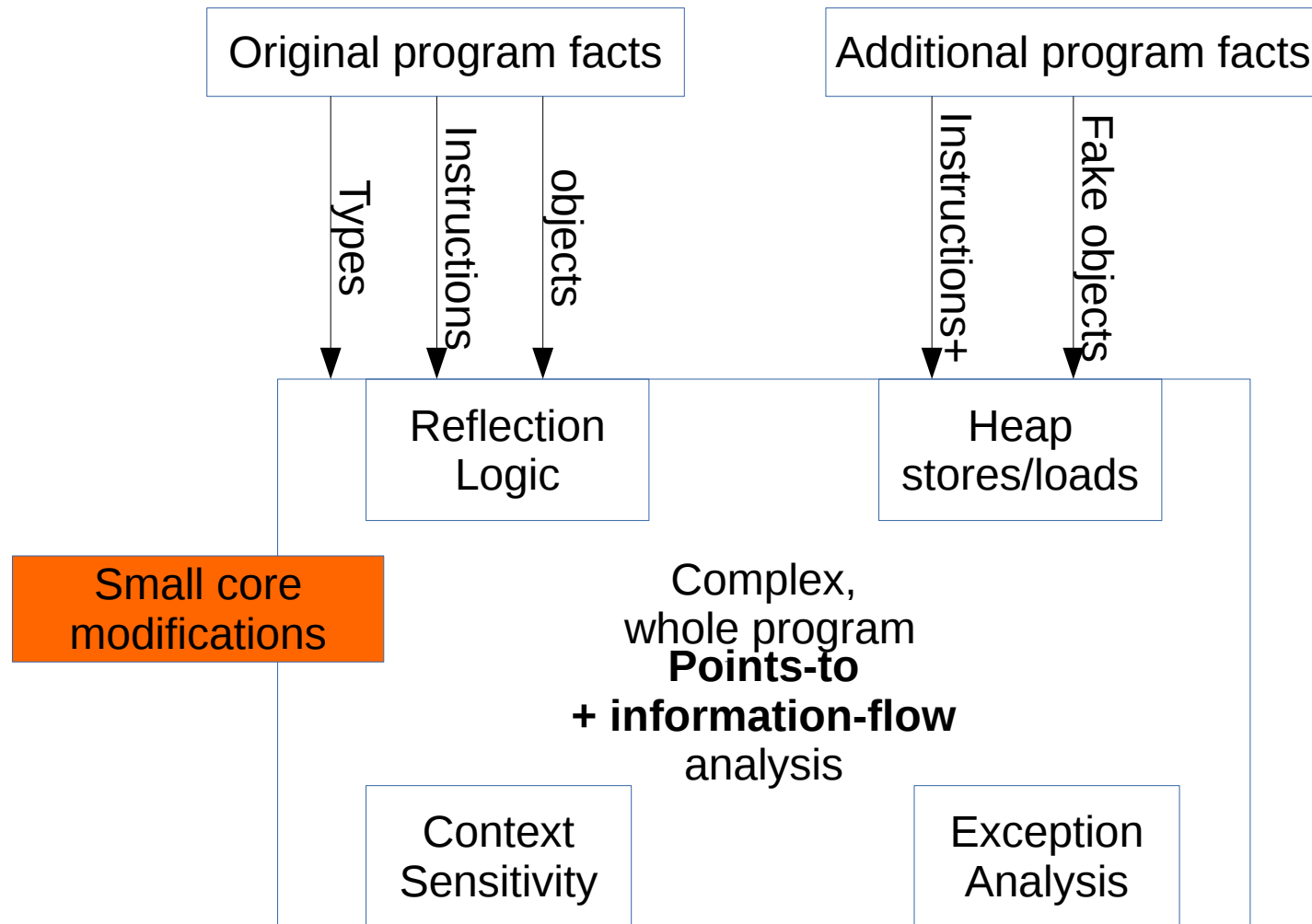
## Parsing the title again...



**Unified:** That is or has been made into one from separate parts; united, combined, consolidated.

I.E. Information flow analysis is not a client of points-to analysis in this work.

# The big picture



# Summary of approach

## Information flow

Represent *information* as artificial abstract objects.

*Information* is introduced at call sites of predefined input functions.

Run modified points-to analysis framework, Doop.

## Dual in points-to

Same abstraction as heap Allocation.

As opposed to an allocation site.



(Y. Smaragdakis, M. Bravenboer,  
G. Kastrinis, G. Balatsouras)

## **Datalog-based pointer analysis framework for Java**

**Declarative: less programming, more specification**

**Sophisticated, very rich set of analyses**

subset-based analysis, fully on-the-fly call graph discovery, field-sensitivity, context-sensitivity, call-site sensitive, object sensitive, thread sensitive, context-sensitive heap, abstraction, type filtering, precise exception analysis

**Support for full semantic complexity of Java**

jvm initialization, reflection analysis, threads, reference queues, native methods, class initialization, finalization, cast checking, assignment compatibility

# Why unify analyses?

**We've seen we can exploit interesting *synergies*:**

- **The very same algorithm can compute, simultaneously, two interlinked but separate analyses...**
- **Existing work on points-to analysis (e.g., reflection) applies to information-flow analysis without extra work.**
- **Performance synergy (less repeated computation).**

# Why unify analyses?

**Reuse** most of the logic that deals with the semantics of the language, in particular:

- Logic for various heap reads and writes (loading and storing, parameter passing...).
- Logic dealing with reflective operations.
- Various flavours of context sensitivities (e.g. Obj, Call-site, etc.).



# Technical approach: redefinitions

## Pointer analysis:

`VarPointsTo (var:Variable, heap:HeapAllocation, ...)`

## Pointer + information flow analysis:

`FlowsToVar (var:Variable, hinfo:HeapOrInfo, ...)`

**where** `HeapOrInfo` **is an abstraction of**

**+ heap allocation site** (`HeapAllocation`)

**+ sites where external information is introduced**  
(`Information`).

`FlowsToVar` **now subsumes** `VarPointsTo` **relation on**  
`HeapAllocation`, **and information flow.**

# Technical approach: introducing information

```
Information(invocation, type) ←  
    InformationSourceMethod(method),  
    CallGraphEdge(invocation, method, ...),  
    MethodReturnType(method, type).
```

```
HeapOrInfo(...) ← Information(...).
```

```
HeapOrInfo(...) ← HeapAllocation(...).
```

# Context sensitive FlowsToVar

```
invocation: Object to = x.method();
```

```
Record(ctx, invocation, hctx),  
InformationObject(hctx, invocation),  
FlowsToVar(to, invocation, ctx, hctx) ←  
  TaintSourceMethod(method, ...),  
  CallGraphEdge(invocation, method, ctx),  
  AssignReturnValue(to, invocation).
```


# More modifications

**The unification abstraction breaks in some parts:**

- **Sanitization.**
- **Call graph construction in the presence of artificial heap objects (`Information`).**
- **More complex information transfer than assignments.**
- **Information can move through objects of unrelated types.**


# Information transfer example

```
void doGet(Request req, Response resp) {
```

```
     String str = req.getParameter("name"); // src  
    StringBuilder strB = new StringBuilder();
```

```
     strB = strB.append(str); 
```

```
     strB = strB.append(" Doe");
```

```
         PrintWriter writer = resp.getWriter();
```

```
        writer.println(strB.toString()); /* Leak */
```

```
    } 
```

# **Analysing open programs (Future work)**

**Information flow analysis has many applications:  
analyzing libraries, webapps, etc.**

**Inversion of control is prevalent here.**

**We need to create mock abstract objects to  
initialize the program.**

**We need to bridge control flow discontinuity (e.g.  
for Android apps).**



Thank you!

More questions/  
discussion?

