Static Analysis and Reflection

Manu Sridharan
(and many collaborators)
IBM Research

What is Reflection?

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

What is Reflection?

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

- Operating on code entities via strings
  - Allocation, invoking methods, accessing fields

What is Reflection?

```
class Factory {
  Object make(String x) {
    return Class.forName(x).newInstance();
  }
}
```

- Operating on code entities via strings
  - Allocation, invoking methods, accessing fields
- Why?
  - Control via configuration files (frameworks)
  - Meta-programming (generic `toString`)
  - No good reason (some uses of JavaScript `eval`)

Reflection Dilemma

	Model	**Ignore**
Precision		
Recall		
Scalability		

Reflection Dilemma

	Model	**Ignore**
Precision		
Recall		
Scalability		

Reflection Dilemma

	Model	**Ignore**
Precision	✗	✓
Recall	✓	✗ (key behaviors?)
Scalability	✗	✓

Reflection Dilemma

	Model	**Ignore**
Precision	✗	✓
Recall	✓	✗ (key behaviors?)
Scalability	✗	✓

<u>Challenge</u>: Strike right balance **for client**

Our Reflection Story

- **Clients**
  - Java taint analysis
  - JavaScript taint analysis (call graphs)
  - JavaScript IDE tools

- **Approaches**
  - Static: model via code analysis
  - Specifications: use additional artifacts
  - Dynamic: observe behavior, record or generalize

Java Taint Analysis

Reflection Handling in TAJ

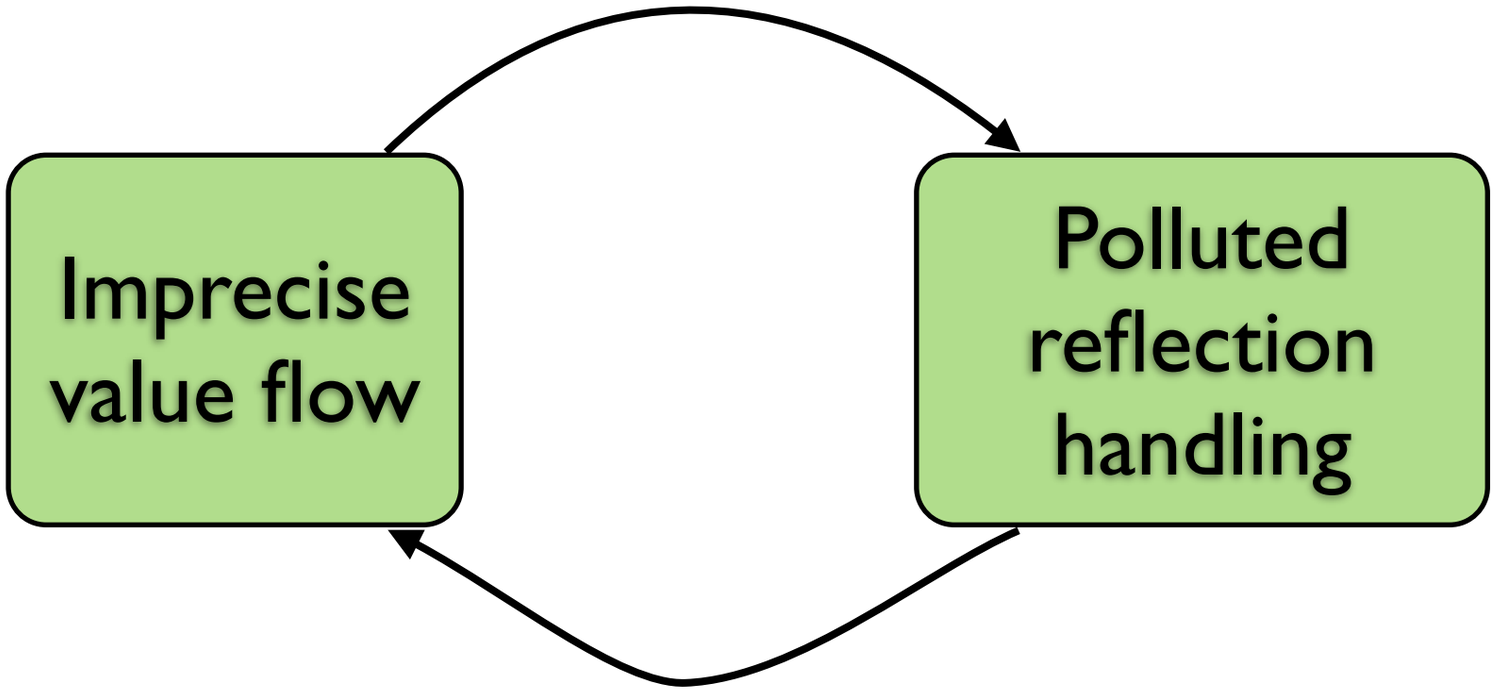
Reflection Handling in TAJ

- Taint Analysis for Java
  - Pointer analysis for call graphs + aliasing
  - Soundness not required
  - But, ignoring reflection is too unsound

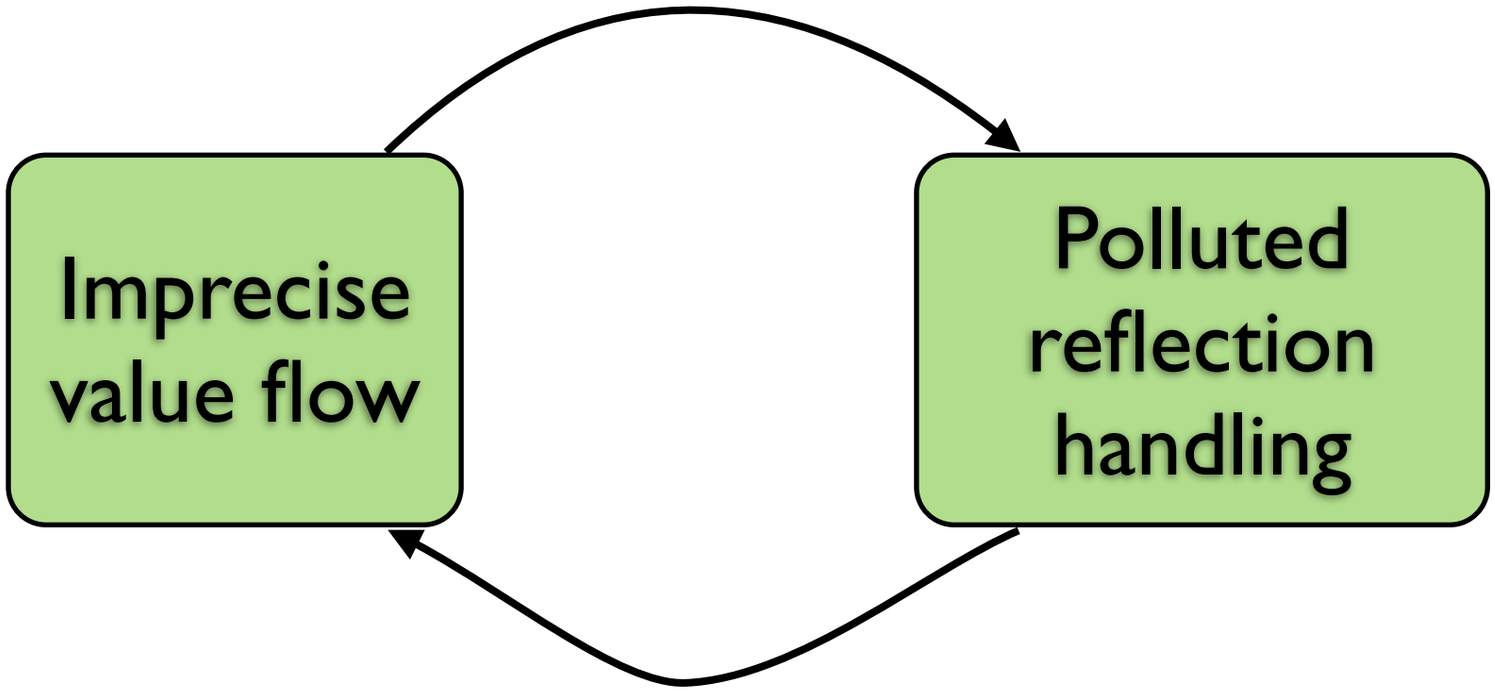
Reflection Handling in TAJ

- Taint Analysis for Java
  - Pointer analysis for call graphs + aliasing
  - Soundness not required
  - But, ignoring reflection is too unsound
- Enhanced pointer analysis with reflection handling
  - Track string constants, `Class` / `Method` objects
  - Generate synthetic IR for reflective operations
    - For `c.newInstance()`, if `c` is `Class<Foo>`, model as **`new`** `Foo()`
  - As in Livshits et al., APLAS'05

A Vicious Cycle...



A Vicious Cycle...



Quadratic blowup

- Huge analysis time / memory
- Highly imprecise result

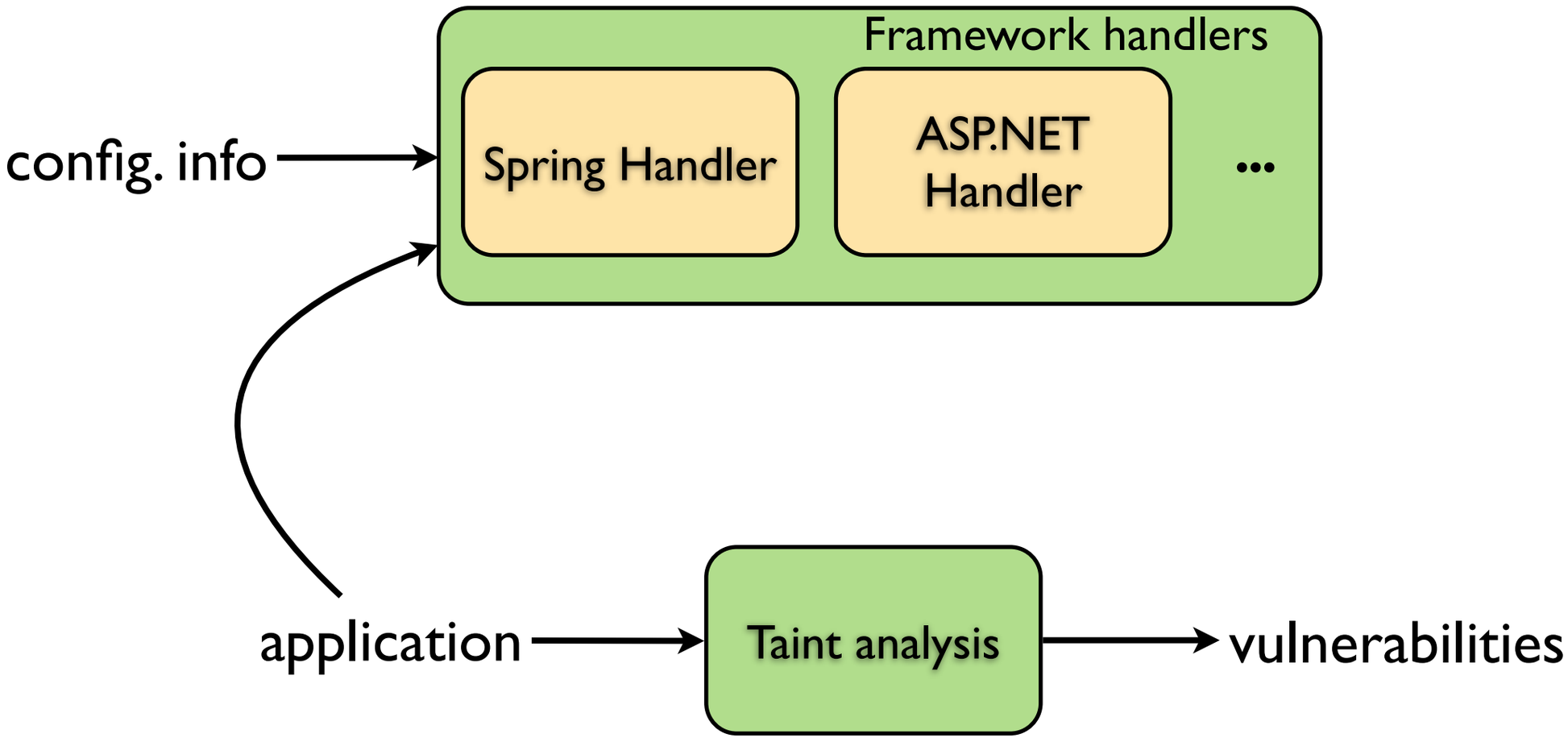
"Fixing" the problem

- Tried bounding pointer analysis, but fragile
- In the end, dumped pointer analysis
  - Instead, heuristic type-based call graph
    - Track aliases during taint analysis
    - See Tripp et al., FASE'13
  - Hand-tuned reflection handling for frameworks
    - But many frameworks in practice...
    - Nasty reflection based on config files

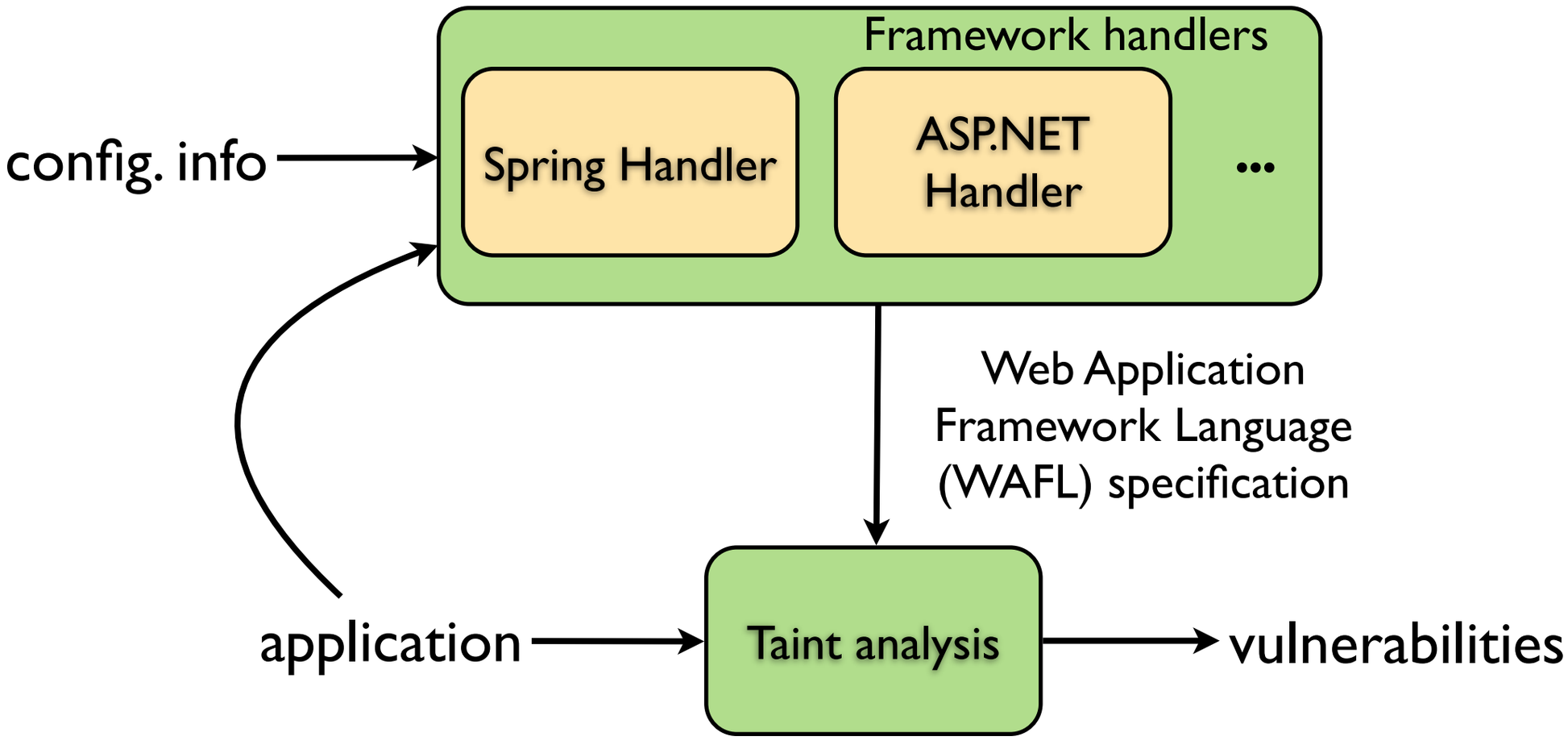
A Framework for Frameworks (OOPSLA'11)

A Framework for Frameworks (OOPSLA'11)

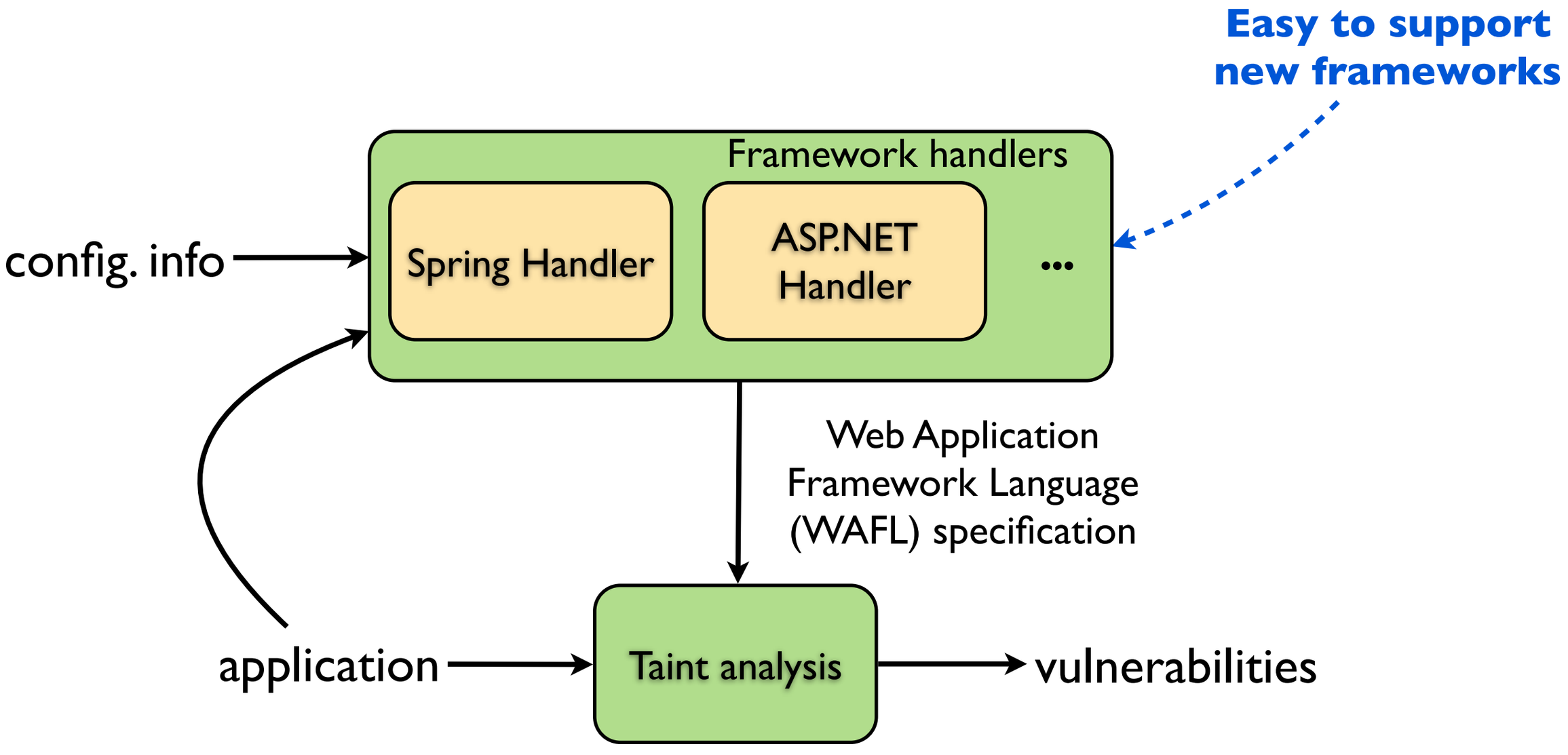
A Framework for Frameworks (OOPSLA'11)



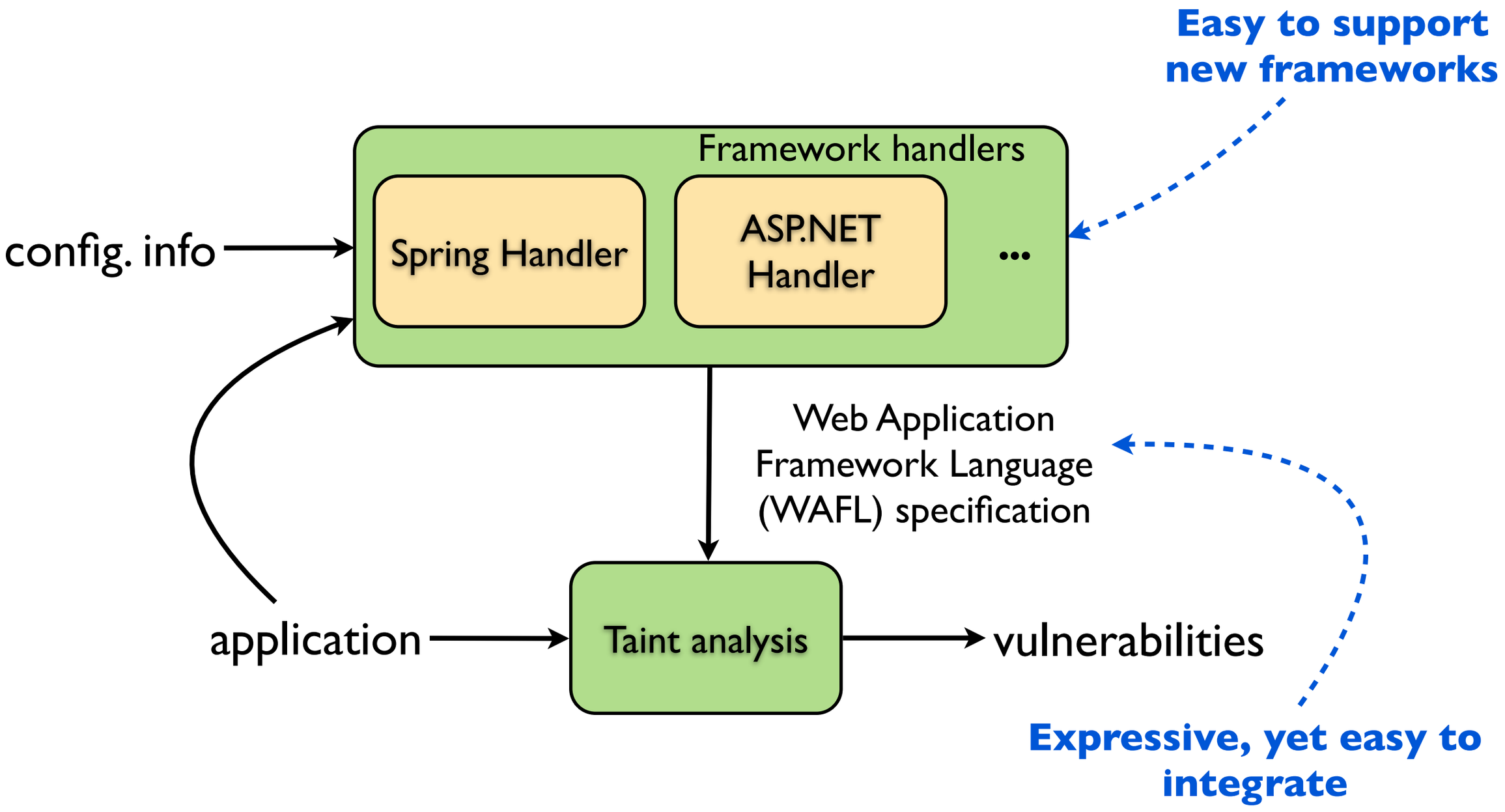
A Framework for Frameworks (OOPSLA'11)



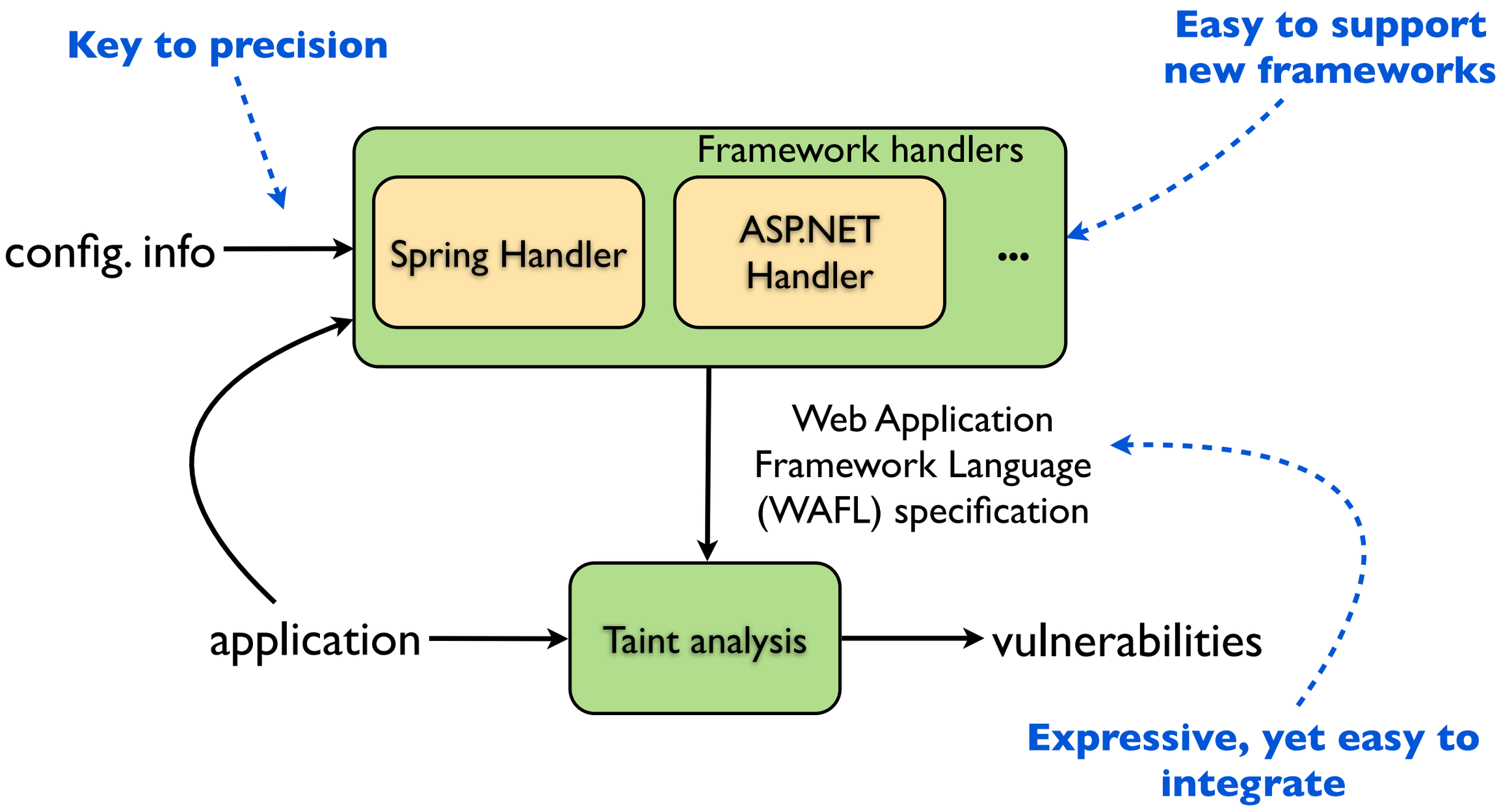
A Framework for Frameworks (OOPSLA'11)



A Framework for Frameworks (OOPSLA'11)



A Framework for Frameworks (OOPSLA'11)



Example: edit profile

Example: edit profile

Code

Configuration

Example: edit profile

Code

```
// for user data
class UserForm {
  String firstName,lastName;
  // getters and setters...
}

// updates profile
class UserAction
  implements IAction {
  String exec(HttpRequest req,
              Object form) {
    UserForm uf = (UserForm) form;
    updateDB(uf);
    ...
  }
}
```

Configuration

Example: edit profile

Code

```
// for user data
class UserForm {
  String firstName,lastName;
  // getters and setters...
}

// updates profile
class UserAction
  implements IAction {
  String exec(HttpRequest req,
              Object form) {
    UserForm uf = (UserForm) form;
    updateDB(uf);
    ...
  }
}
```

Configuration

```
<action url="/edit"
        type="UserAction"
        formtype="UserForm">
</action>
```

In English: When "/edit" is visited, create a UserForm object (reflection), set its fields using request data (reflection), and pass it to UserAction.exec() (reflection).

WAFL: synthetic methods

```
fun entrypoint UserAction_entry(request) {
  UserForm f = new UserForm();
  f.setFirstName(request.getParam("firstName"));
  f.setLastName(request.getParam("lastName"));
  (new UserAction()).exec(request, f);
}
```

WAFL: synthetic methods

```
fun entrypoint UserAction_entry(request) {
  UserForm f = new UserForm();
  f.setFirstName(request.getParam("firstName"));
  f.setLastName(request.getParam("lastName"));
  (new UserAction()).exec(request, f);
}
```

- Simple structure: no branches, loops, etc.
  - Eases integration with analysis engine
  - Taint analysis usually flow insensitive anyway
- Based on both app code and config info

What about dynamic?

- The Tamiflex approach (Bodden et al., ICSE'11)
  - Log runtime reflective operations
  - Use log to transform code
  - Ensures soundness for tested inputs
- Difficulties
  - Running server code can be hard!
  - Need inputs to cover behaviors

Can we do better?

- F4F quite successful
- But, requires writing framework handlers
- Can we further automate?
- Maybe generalize from dynamic I/O?
- **Important problem**

JavaScript Taint Analysis

(or, Getting Tamed by jQuery)

Pointer Analysis Needed

```
var x = {};
// initialize object properties
x.foo = function f1() { return 23; }
x.bar = function f2() { return 42; }
x.foo(); // invokes f1
```

Pointer Analysis Needed

```
var x = {};
// initialize object properties
x.foo = function f1() { return 23; }
x.bar = function f2() { return 42; }
x.foo(); // invokes f1
```

- No declared types; objects can gain or lose fields

Pointer Analysis Needed

```
var x = {};
// initialize object properties
x.foo = function f1() { return 23; }
x.bar = function f2() { return 42; }
x.foo(); // invokes f1
```

- No declared types; objects can gain or lose fields
- Pointer analysis needed for call graphs
  - Most method calls are "virtual"
  - Cannot narrow call targets via types / arity

Dynamic Property Accesses

```
var f = p() ? “foo” : “baz”;
// writes to o.foo or o.baz
o[f] = “Hello!”;
```

Dynamic Property Accesses

```
var f = p() ? “foo” : “baz”;
// writes to o.foo or o.baz
o[f] = “Hello!”;
```

- Used frequently inside frameworks

Dynamic Property Accesses

```
var f = p() ? “foo” : “baz”;
// writes to o.foo or o.baz
o[f] = “Hello!”;
```

- Used frequently inside frameworks
- Increases worst-case analysis complexity!

Dynamic Property Accesses

```
var f = p() ? “foo” : “baz”;
// writes to o.foo or o.baz
o[f] = “Hello!”;
```

- Used frequently inside frameworks
- Increases worst-case analysis complexity!
- Leads to significant blowup in practice

Correlated Accesses

```
function extend(dest,src) {
  for (var prop in src)
    // correlated accesses
    dest[prop] = src[prop];
}
```

Correlated Accesses

```
function extend(dest,src) {
  for (var prop in src)
    // correlated accesses
    dest[prop] = src[prop];
}
```

- Correlated: `prop` has same value at both accesses

Correlated Accesses

```
function extend(dest,src) {
  for (var prop in src)
    // correlated accesses
    dest[prop] = src[prop];
}
```

- Correlated: `prop` has same value at both accesses
- Standard points-to analysis misses correlation
  - Analysis merges all properties of `src`
  - For frameworks, leads to "quadratic blowup"

Function Extraction + Context Sensitivity

```
function extend(dest,src) {
  for (var prop in src)
    dest[prop] = src[prop];
}
```

Function Extraction + Context Sensitivity

```
function extend(dest,src) {
  for (var prop in src)
    dest[prop] = src[prop];
}
```

```
function extend(dest,src) {
  for (var prop in src)
    // extract accesses into
    // fresh function
    (function ext(p) {
      dest[p] = src[p];
     })(prop);
}
```

Function Extraction + Context Sensitivity

```
function extend(dest,src) {
  for (var prop in src)
    dest[prop] = src[prop];
}
```

```
function extend(dest,src) {
  for (var prop in src)
    // extract accesses into
    // fresh function
    (function ext(p) {
      dest[p] = src[p];
     })(prop);
}
```

ext contexts: p == “foo”, p == “baz”, ...

Function Extraction + Context Sensitivity

```
function extend(dest,src) {
  for (var prop in src)
    dest[prop] = src[prop];
}
```

```
function extend(dest,src) {
  for (var prop in src)
    // extract accesses into
    // fresh function
    (function ext(p) {
      dest[p] = src[p];
     })(prop);
}
```

- Analyze new functions with clone per property name
  - Similar to object sensitivity / CPA
- Details in ECOOP'12

`ext` contexts: `p == "foo"`, `p == "baz"`, ...

Results: Scalability

Framework	Baseline$^-$	Baseline$^+$	Correlations$^-$	Correlations$^+$
dojo	* (*)	* (*)	3.1 (30.4)	6.7 (*)
jquery	*	*	78.5	*
mootools	0.7	*	3.1	*
prototype.js	*	*	4.4	4.5
yui	*	*	2.2	2.1

- Dramatic improvements with Correlations$^-$
  - Useful for an under-approximate call graph
- For '+' configs, issues remain with `call` / `apply`

Unnecessary Reflection

```
var e = "blur,focus,load".split(",");

for(var i=0;i<e.length;i++) {
  var o = e[i];
  jQuery.fn[o] = function() { ... };
  jQuery.fn["un"+o] = function() { ... };
  jQuery.fn["one"+o] = function() { ... };
}
```

Unnecessary Reflection

```
var e = "blur,focus,load".split(",");

for(var i=0;i<e.length;i++) {
  var o = e[i];
  jQuery.fn[o] = function() { ... };
  jQuery.fn["un"+o] = function() { ... };
  jQuery.fn["one"+o] = function() { ... };
}
```

```
jQuery.fn.blur = function() { ... };
jQuery.fn.unblur = function() { ... };
jQuery.fn.oneblur = function() { ... };
jQuery.fn.focus = function() { ... };
jQuery.fn.unfocus = function() { ... };
jQuery.fn.onefocus = function() { ... };
jQuery.fn.load = function() { ... };
jQuery.fn.unload = function() { ... };
jQuery.fn.oneload = function() { ... };
```

Dynamic Determinacy Analysis

Dynamic Determinacy Analysis

- Much reflection can be rewritten away
  - E.g., `eval` of constant, jQuery initialization

Dynamic Determinacy Analysis

- Much reflection can be rewritten away
  - E.g., `eval` of constant, jQuery initialization
- Pure static detection hard (needs a call graph!)

Dynamic Determinacy Analysis

- Much reflection can be rewritten away
  - E.g., `eval` of constant, jQuery initialization
- Pure static detection hard (needs a call graph!)
- Idea: *Prove* fixed behavior based on *dynamic* analysis
  - Find expressions "untainted" by inputs
  - Similar to dynamic information flow
  - See PLDI'13 paper for details

Dynamic Determinacy Analysis

- Much reflection can be rewritten away
  - E.g., `eval` of constant, jQuery initialization
- Pure static detection hard (needs a call graph!)
- Idea: *Prove* fixed behavior based on *dynamic* analysis
  - Find expressions "untainted" by inputs
  - Similar to dynamic information flow
  - See PLDI'13 paper for details
- Analyzed jQuery! But...version 1.0
  - Challenge: non-deterministic event handlers

JavaScript IDE Tools

Challenges

- Developers demand rich IDE functionality
  - Code navigation (jump to declaration)
  - Smart completion
  - Refactoring
- Hard to build these features for JavaScript
  - Reflection, lack of types, etc.
- For IDE, must be fast

Idea: Under-Approximate

Idea: Under-Approximate

- *Deliberately ignore* "hard" features of JS
  - I.e., ignore reflection

Idea: Under-Approximate

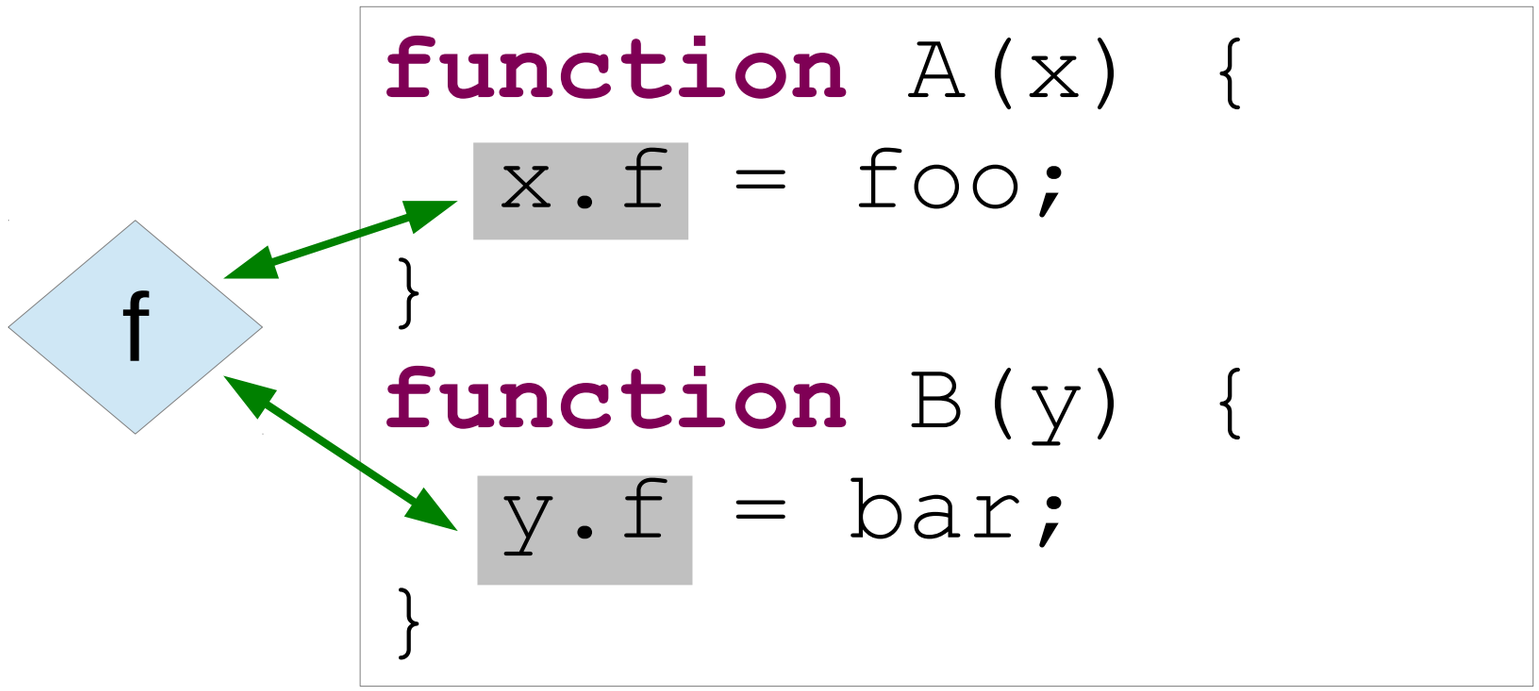
- *Deliberately ignore* "hard" features of JS
  - I.e., ignore reflection
- Ok to miss some behaviors in IDEs
  - Even Java refactorings ignore reflection

Idea: Under-Approximate

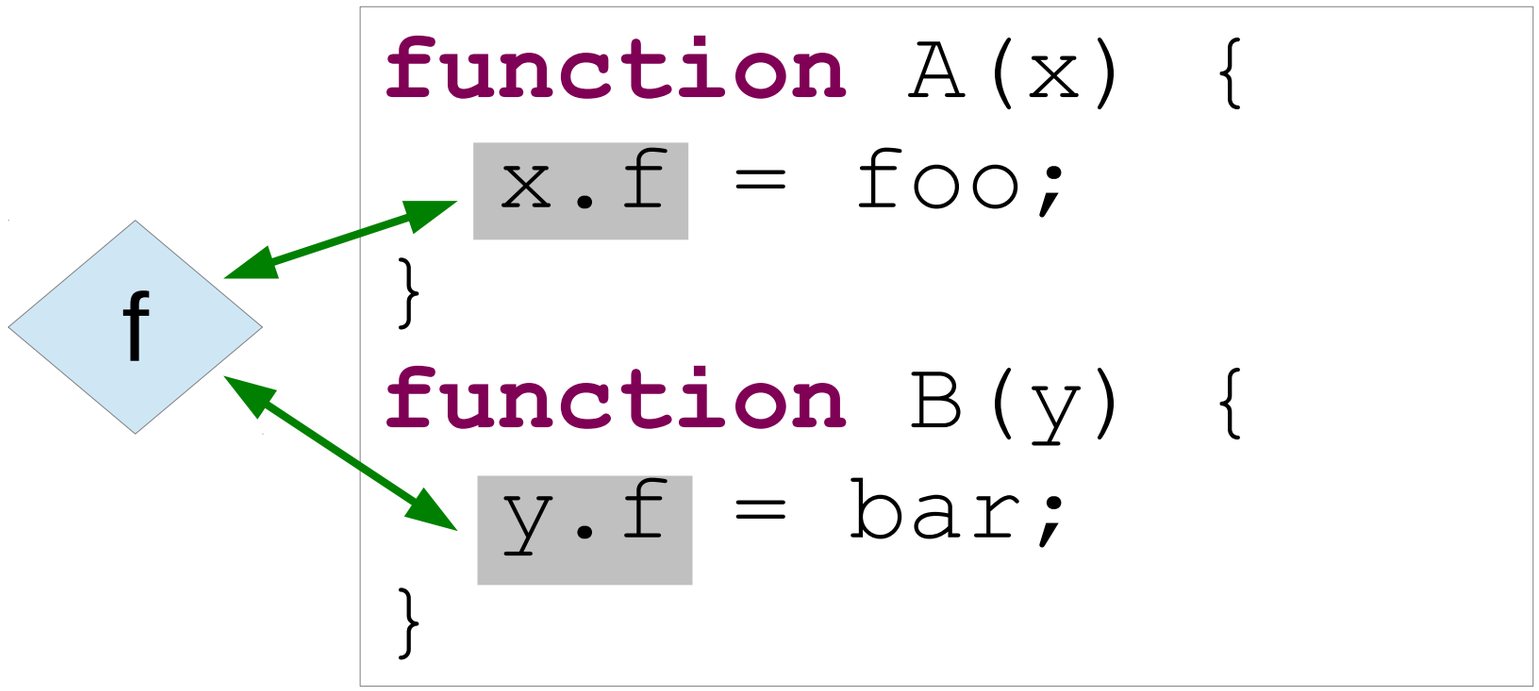
- *Deliberately ignore* "hard" features of JS
  - I.e., ignore reflection
- Ok to miss some behaviors in IDEs
  - Even Java refactorings ignore reflection
- Design analysis to scale well and capture *most* behaviors

Field-Based Call Graphs

Field-Based Call Graphs



Field-Based Call Graphs



Ignore dynamic accesses x[p]

Why Does This Work?

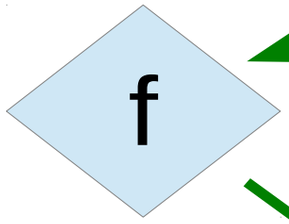
```
1 function extend(dst,src) {
2   for (var x in src) {
3     dst[x] = src[x];
4   }
5 }
6 foo.f = function() { /*...*/ };
7 extend(bar,foo);
8 bar.f();
```

Why Does This Work?

```
1 function extend(dst,src) {
2   for (var x in src) {
3     dst[x] = src[x];
4   }
5 }

6 foo.f = function() { /*...*/ };

7 extend(bar, foo);

8 bar.f();
```

Why Does This Work?

```
1 function extend(dst,src) {
2   for (var x in src) {
3     dst[x] = src[x];
4   }
5 }
6 foo.f = function() { /*...*/ };
7 extend(bar,foo);
8 bar.f();
```



call edge

Evaluation

- Compared with dynamic call graphs
  - No other usable static technique
  - Best-effort coverage
- Measured precision, recall, and runtime

Benchmarks

Benchmark	*Framework*	*LOC*	*Precision*	*Recall*	*Time*
3dmodel	none	4.9k	93%	100%	0.26s
pacman	none	3.5k	94%	100%	0.47s
pdfjs	none	31.7k	77%	99%	5.62s
coolclock	jQuery	6.9k	89%	98%	1.32s
fullcalendar	jQuery	12.3k	84%	93%	2.85s
htmledit	jQuery	3.6k	81%	84%	0.80s
markitup	jQuery	6.5k	82%	94%	1.28s
pong	jQuery	3.6k	78%	93%	0.83s
flotr	Prototype	4.9k	72%	83%	1.76s
beslimed	MooTools	4.8k	78%	84%	1.06s

Benchmarks

Benchmark	*Framework*	*LOC*	*Precision*	*Recall*	*Time*
3dmodel	none	4.9k	93%	100%	0.26s
pacman	none	3.5k	94%	100%	0.47s
pdfjs	none	31.7k	[illegible]	[illegible]	5.62s
coolclock	jQuery	6.9k	[illegible]	[illegible]	1.32s
fullcalendar	jQuery	12.3k	84%	93%	2.85s
htmledit	jQuery	3.6k	81%	84%	0.80s
markitup	jQuery	6.5k	82%	94%	1.28s
pong	jQuery	3.6k	78%	93%	0.83s
flotr	Prototype	4.9k	72%	83%	1.76s
beslimed	MooTools	4.8k	78%	84%	1.06s

Seconds, not hours

Benchmarks

Benchmark	*Framework*	*LOC*	*Precision*	*Recall*	*Time*
3dmodel	none	4.9k	93%	100%	0.26s
pacman	none	3.5k	94%	100%	0.47s
pdfjs	n		77%	99%	5.62s
coolclock	jQuery	6.9k	89%	98%	1.32s
fullcalendar	jQuery	12.3k	84%	93%	2.85s
htmledit	jQuery	3.6k	81%	84%	0.80s
markitup	jQuery	6.5k	82%	94%	1.28s
pong	jQuery	3.6k	78%	93%	0.83s
flotr	Prototype	4.9k	72%	83%	1.76s
beslimed	MooTools	4.8k	78%	84%	1.06s

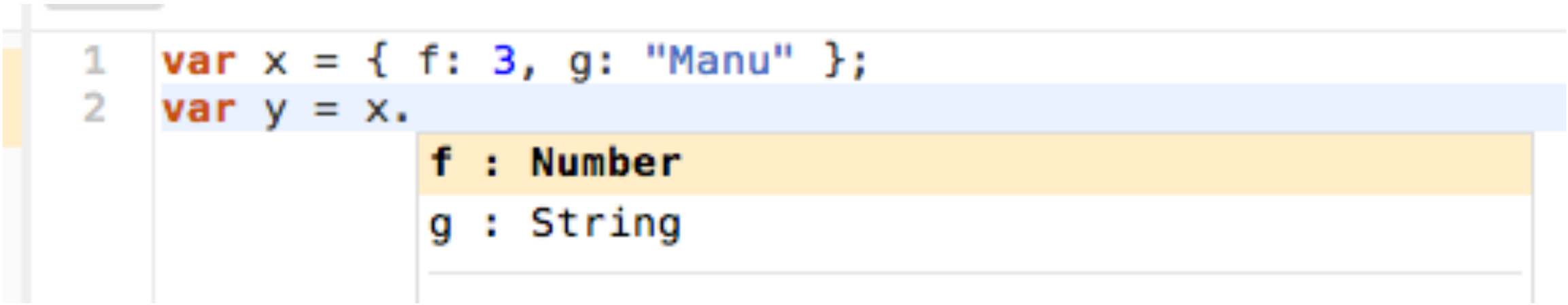
>70% Precision

Benchmarks

Benchmark	*Framework*	*LOC*	*Precision*	*Recall*	*Time*
3dmodel	none	4.9k	93%	100%	0.26s
pacman	none	3.5k	94%	100%	0.47s
pdfjs	none			99%	5.62s
coolclock	jQuery	6.9k	89%	98%	1.32s
fullcalendar	jQuery	12.3k	84%	93%	2.85s
htmledit	jQuery	3.6k	81%	84%	0.80s
markitup	jQuery	6.5k	82%	94%	1.28s
pong	jQuery	3.6k	78%	93%	0.83s
flotr	Prototype	4.9k	72%	83%	1.76s
beslimed	MooTools	4.8k	78%	84%	1.06s

>80% Recall

Smart Completion



(Eclipse Orion: `eclipse.org/orion`)

Smart Completion

```
1  var x = { f: 3, g: "Manu" };
2  var y = x.
```

```
f : Number
g : String
```

(Eclipse Orion: `eclipse.org/orion`)

- Again, flow analysis required, hard to scale
  - Unlike call graph, needs flows of all objects

Smart Completion

```
1  var x = { f: 3, g: "Manu" };
2  var y = x.
```

```
f : Number
g : String
```

(Eclipse Orion: `eclipse.org/orion`)

- Again, flow analysis required, hard to scale
  - Unlike call graph, needs flows of all objects
- Observation: most nasty reflection *occurs in libraries*

Smart Completion

```
1  var x = { f: 3, g: "Manu" };
2  var y = x.
```

```
f : Number
g : String
```

(Eclipse Orion: `eclipse.org/orion`)

- Again, flow analysis required, hard to scale
  - Unlike call graph, needs flows of all objects
- Observation: most nasty reflection *occurs in libraries*
- **Approach: dynamic API inference for libraries**
  - Run instrumented library on unit tests
  - Record observed types and use for flow analysis

Smart Completion

```
1  var x = { f: 3, g: "Manu" };
2  var y = x.
```

```
f : Number
g : String
```

(Eclipse Orion: `eclipse.org/orion`)

- Again, flow analysis required, hard to scale
  - Unlike call graph, needs flows of all objects
- Observation: most nasty reflection *occurs in libraries*
- **Approach: dynamic API inference for libraries**
  - Run instrumented library on unit tests
  - Record observed types and use for flow analysis
- Compared to hand-written models, less effort and more complete

Conclusions

Lessons Learned

Lessons Learned

- Details of reflection handling matter!
  - Can dominate more common analysis parameters (*-sensitivity)

Lessons Learned

- Details of reflection handling matter!
  - Can dominate more common analysis parameters (*-sensitivity)
- Best solutions specialized to client
  - Varying performance, soundness needs

Lessons Learned

- Details of reflection handling matter!
  - Can dominate more common analysis parameters (*-sensitivity)
- Best solutions specialized to client
  - Varying performance, soundness needs
- Quadratic blowup complicates debugging
  - Delta debugging helps

Next Steps

- Better abstractions / guidance for clients
- Refine under-approximate approaches
  - With help from user?
- Better language constructs?
  - E.g., MorphJ (Huang and Smaragdakis, PLDI'08)

Thanks!