The Secret Sauce in Efficient and Precise Static Analysis

The Beauty of Distributive, Summary-Based Static Analyses (and how to master them)

Eric Bodden

PADERBORN UNIVERSITY
The University for the Information Society

Fraunhofer
IEM

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

SECURE
SOFTWARE ENGINEERING
GROUP

Lisa Nguyen

Fraunhofer IEM

Stefan Krüger

UNIVERSITÄT PADERBORN

Karim Ali

UNIVERSITY OF ALBERTA

Johannes Späth

Fraunhofer IEM

Ben Hermann

UNIVERSITÄT PADERBORN

Johannes Lerch

Continental

Mira Mezini

TECHNISCHE UNIVERSITÄT DARMSTADT

No. 1 goal of this talk

Refute these myths:

~~*As static analysis gets more precise, it becomes less efficient.*~~

~~*Context-sensitive and flow-sensitive analysis is impossible to scale, thus one needs to approximate.*~~

CogniCrypt

www.cognicrypt.de

[ECOOP 18]

Common mistake in static analysis

Client analysis

Highly precise: context-sensitive, field-sensitive, flow-sensitive, …

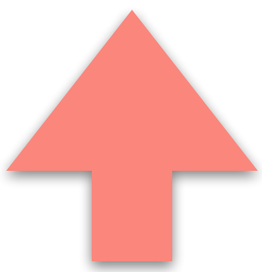
Pointer analysis

Very imprecise: Andersen-style, flow-insensitive, context-insensitive

Common mistake in static analysis

Client analysis

Highly precise: context-sensitive, field-sensitive, flow-sensitive, …



Imprecision

Pointer analysis

Very imprecise: Andersen-style, flow-insensitive, context-insensitive

Example: false positive due to flow-insensitive pointer analysis

```
A a = new A();
A b = new A();

a.x = secret();
leak(b.x);

a = b;
```

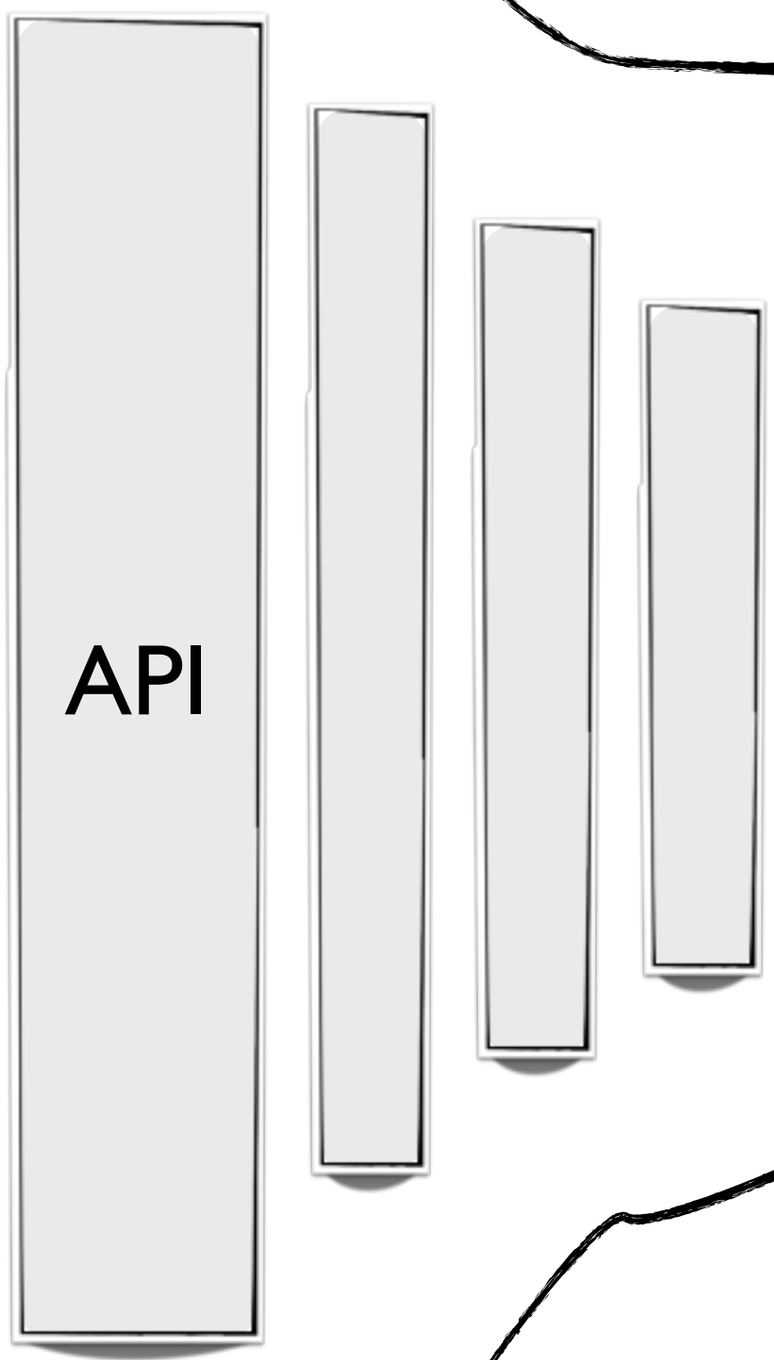
alias created afterwards, yet causes false positive

Lessons

1. **Precise pointer analyses** are key!

But precise analyses *are* hard to scale, no?

They are, as whole-program analysis, but often we can get away with a Demand-driven Analysis



Can an attacker manipulate the string?

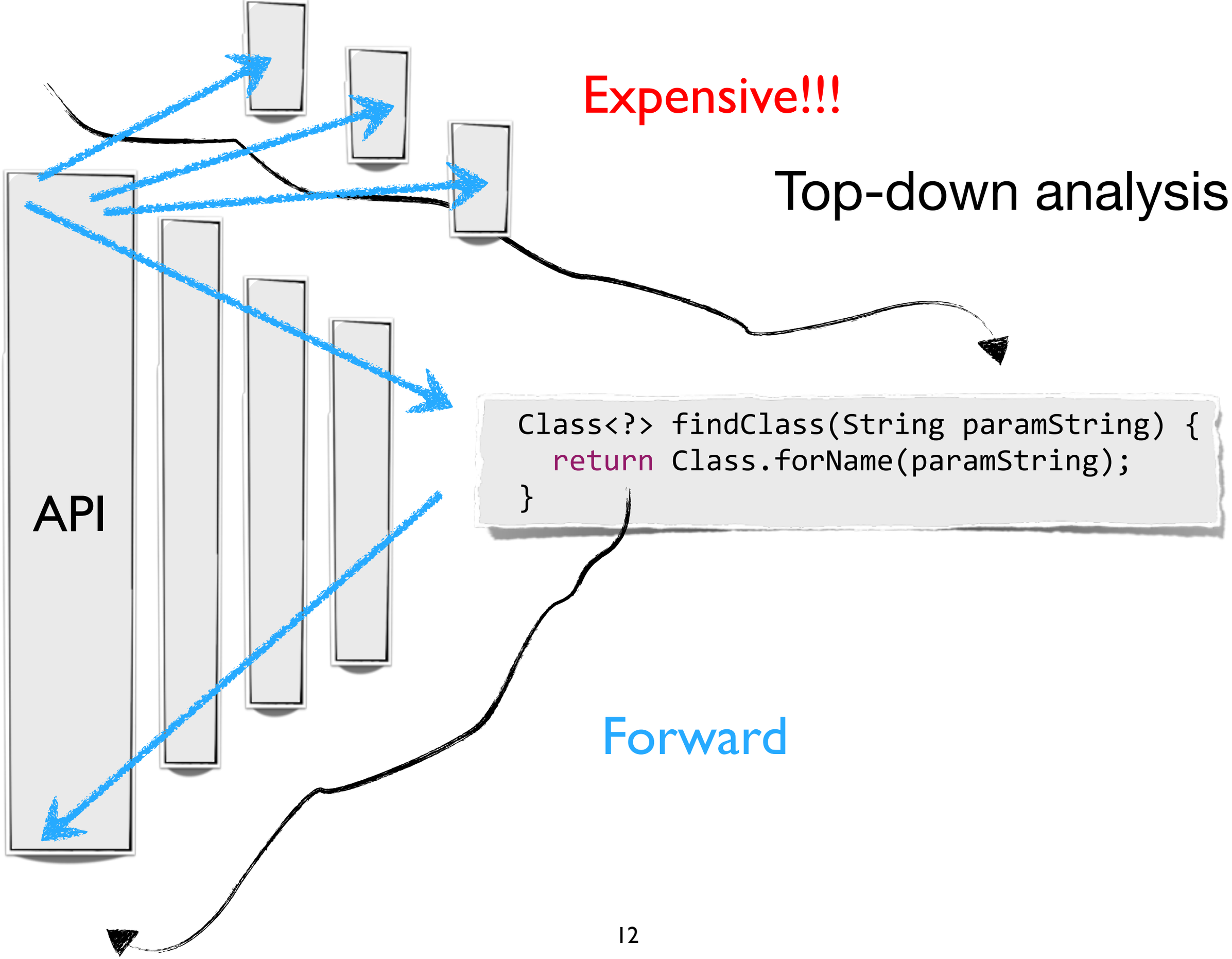
Integrity problem

```
Class<?> findClass(String paramString) {
  return Class.forName(paramString);
}
```

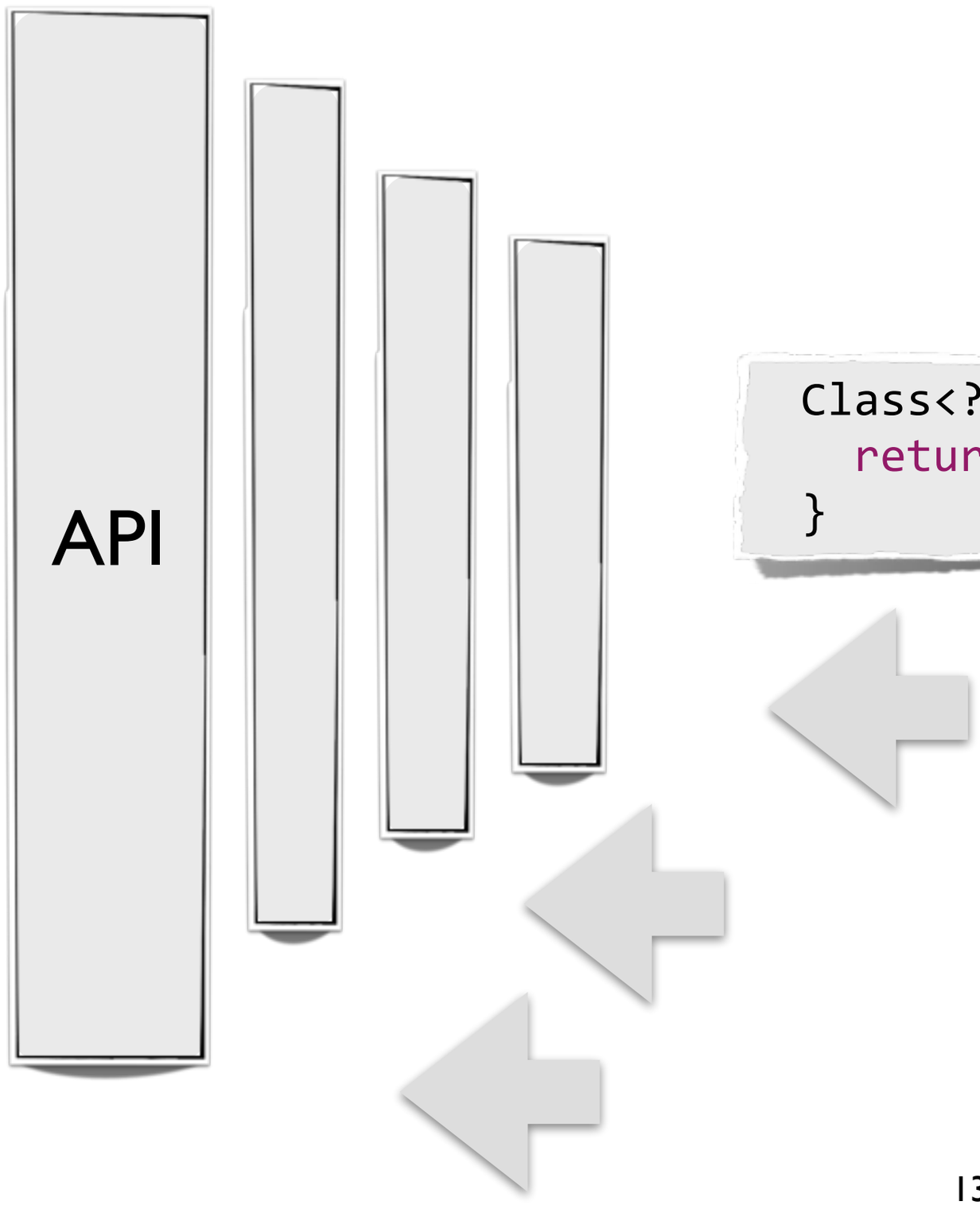
Can the attacker get hold of the returned class?

Confidentiality problem

Taint Analysis!

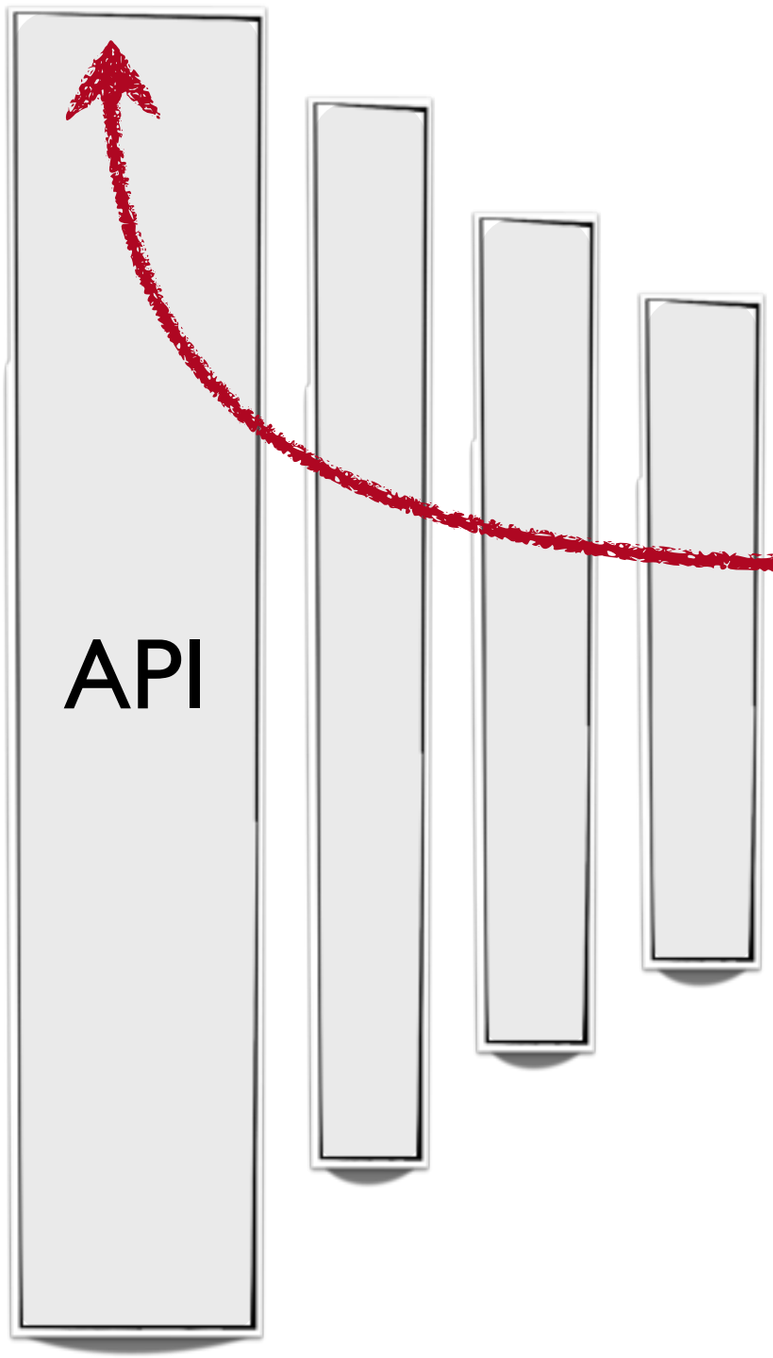

Expensive!!!
Top-down analysis
API
Class<?> findClass(String paramString) {
return Class.forName(paramString);
}
Forward

Bottom-up analysis




```
Class<?> findClass(String paramString) {
  return Class.forName(paramString);
}
```

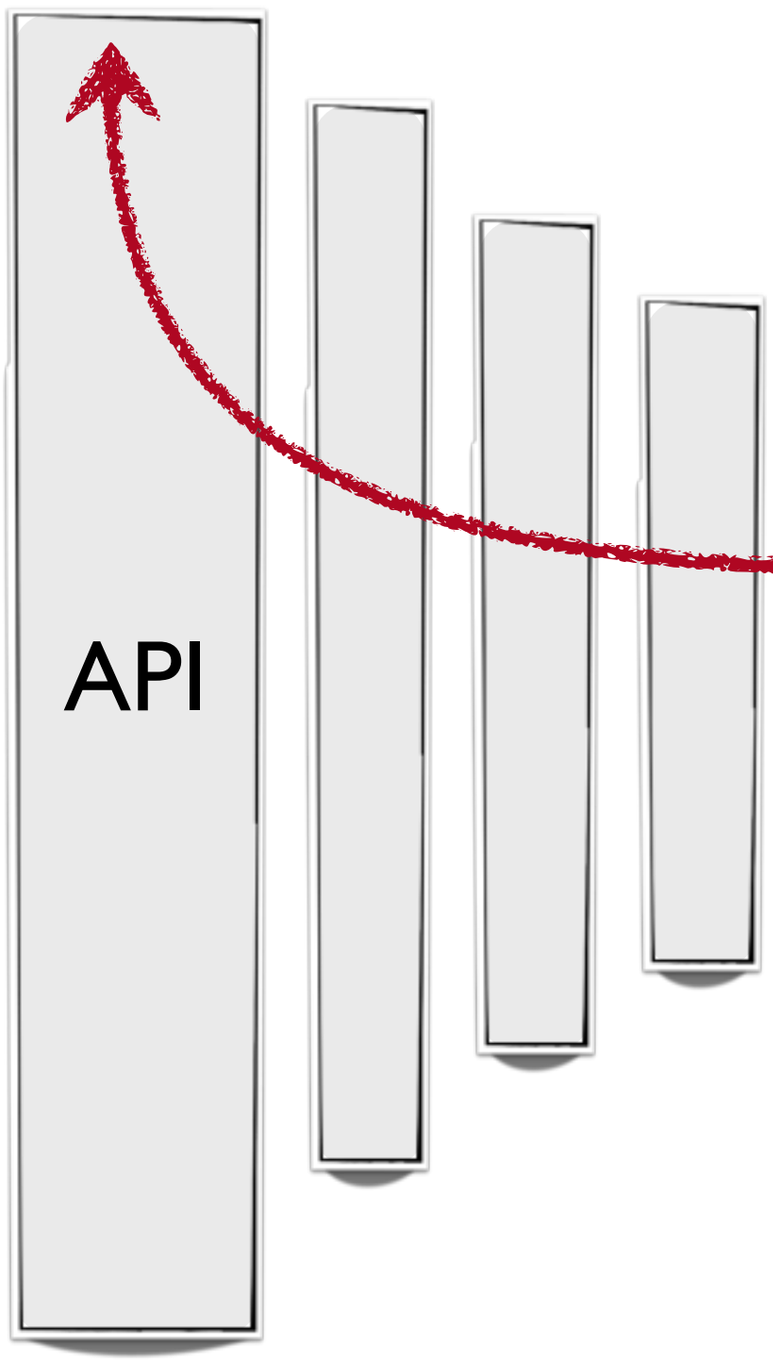
[ASE'15]



Start at stmt of interest

```
Class<?> findClass(String paramString) {
  return Class.forName(paramString);
}
```

[ASE'15]



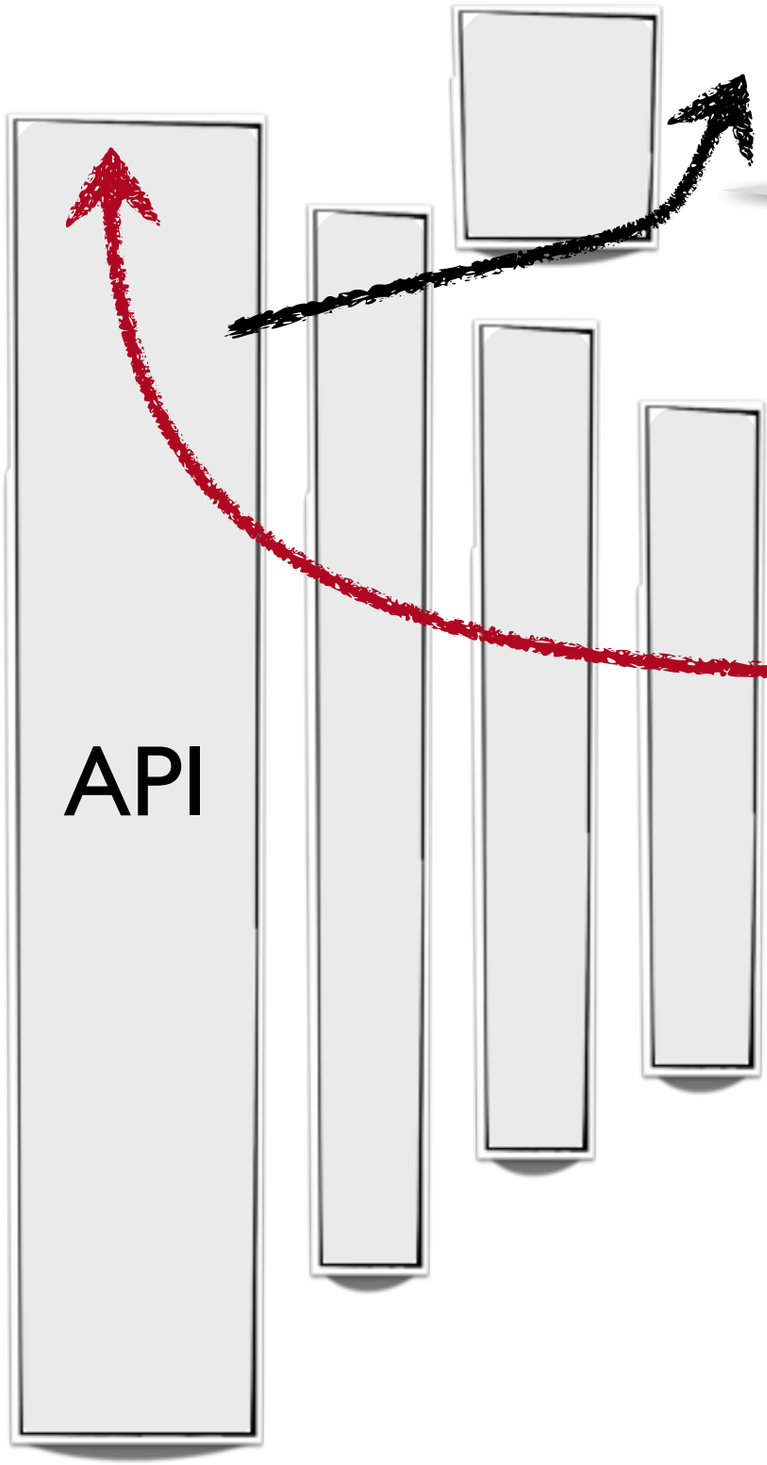
Return to
all callers

```
Class<?> findClass(String paramString) {
  return Class.forName(paramString);
}
```

aka. “unbalanced return”

no calling context!

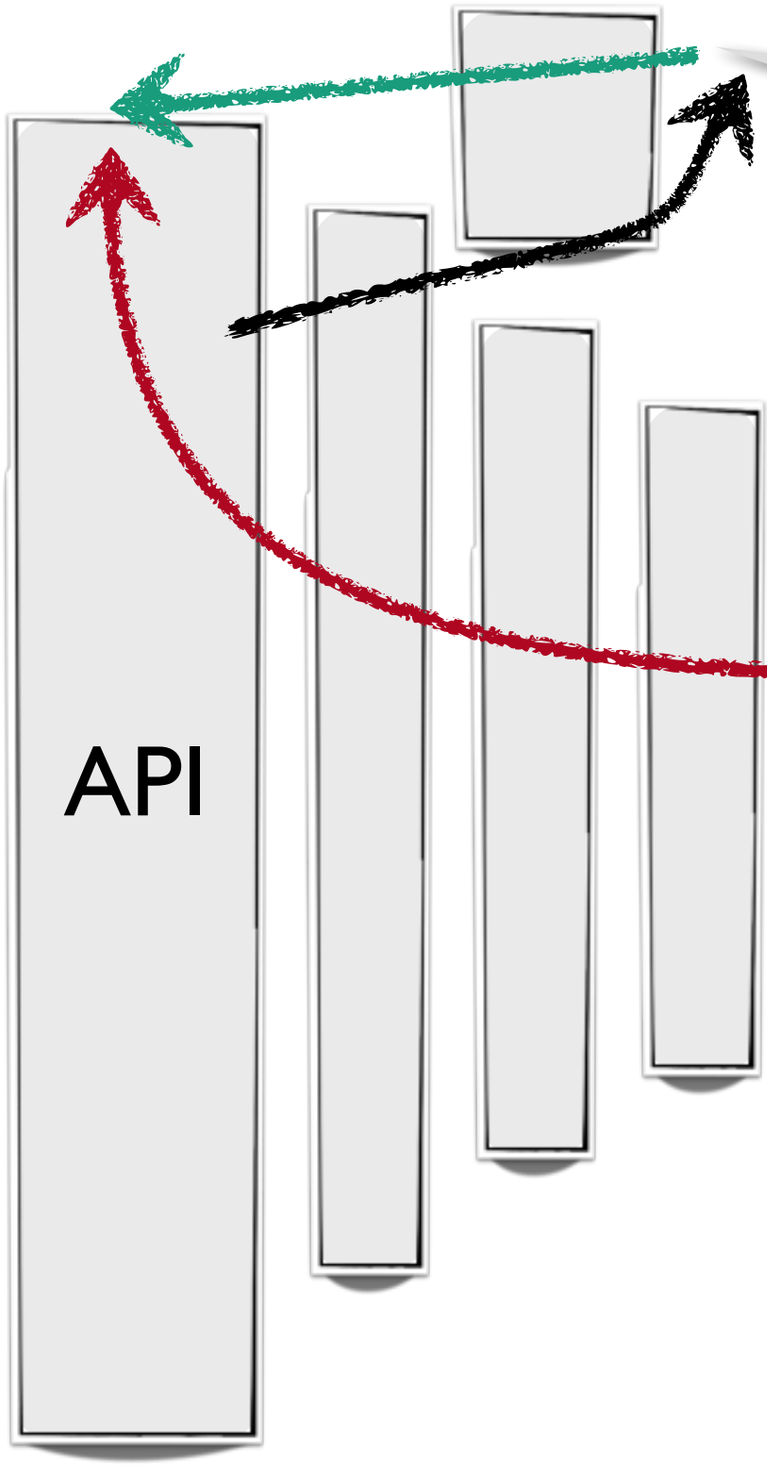
[ASE’15]



Process a call, building call stack

```
Class<?> findClass(String paramString) {
  return Class.forName(paramString);
}
```

[ASE'15]



Balanced return
only to correct call site

```
Class<?> findClass(String paramString) {
  return Class.forName(paramString);
}
```

[ASE'15]

Properties of a demand-driven analysis

- Computed responses to on-demand queries, e.g.: *pointsTo*(`a.f.g`)
- Analysis starts anywhere, “in the middle of the program”
- It typically conducts some combination of backward and forward analysis
- It necessarily has to deal with unbalanced returns

Effects of a demand-driven analysis

- The analysis becomes localised
- Precision therefore may increase
- But: it may report problems in dead code
- Can do recursive queries

On-demand analysis

- Compute **access paths** only when needed
- Compute **pointer information** only when needed
- Compute **string information** only when needed
- Compute **call edges** only when needed

[ASE 2015]

[ECOOP 2016]

[ECOOP 2018]

ongoing…

can query one another!

Lessons

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision

Gaining further efficiency through procedure summaries

taint analysis tracks both keys

```
void main() {
    byte[] pri = privateKey();
    byte[] pub = publicKey();
    byte[] priAlias = foo(pri); //context c1
    byte[] pubAlias = foo(pub); //context c2
    ...
    log(pub);
}

byte[] foo(byte[] ba) {
    //some hard-to-analyze code omitted
    return ba;
}
```

two calling contexts

```
void main() {
    byte[] pri = privateKey();
    byte[] pub = publicKey();
    byte[] priAlias = foo(pri); //context c1
    byte[] pubAlias = foo(pub); //context c2
    ...
    log(pub);
}

byte[] foo(byte[] ba) {
    //some hard-to-analyze code omitted
    return ba;
}
```

need context-sensitivity to avoid false warning

```
void main() {
    byte[] pri = privateKey();
    byte[] pub = publicKey();
    byte[] priAlias = foo(pri); //context c1
    byte[] pubAlias = foo(pub); //context c2
    ...
    log
}

byte[] foo(byte[] ba) {
    //some hard-to-analyze code omitted
    return ba;
}
```

call-strings approach analyzes foo twice

(c1,ba)

(c2,ba)

```
void main() {
    byte[] pri = privateKey();
    byte[] pub = publicKey();
    byte[] priAlias = foo(pri); //context c1
    byte[] pubAlias = foo(pub); //context c2
    ...
    log
}

byte[] foo(byte[] ba) {
    //some hard-to-analyze code omitted
    return ba;
}
```

summary-based analysis instead only once!

ba

<ret>

Extensional vs. intensional summaries

extensional definition of the *increment* function:

$$0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, \ldots$$

intensional definition:

$$\lambda x.\ x \mapsto x + 1$$

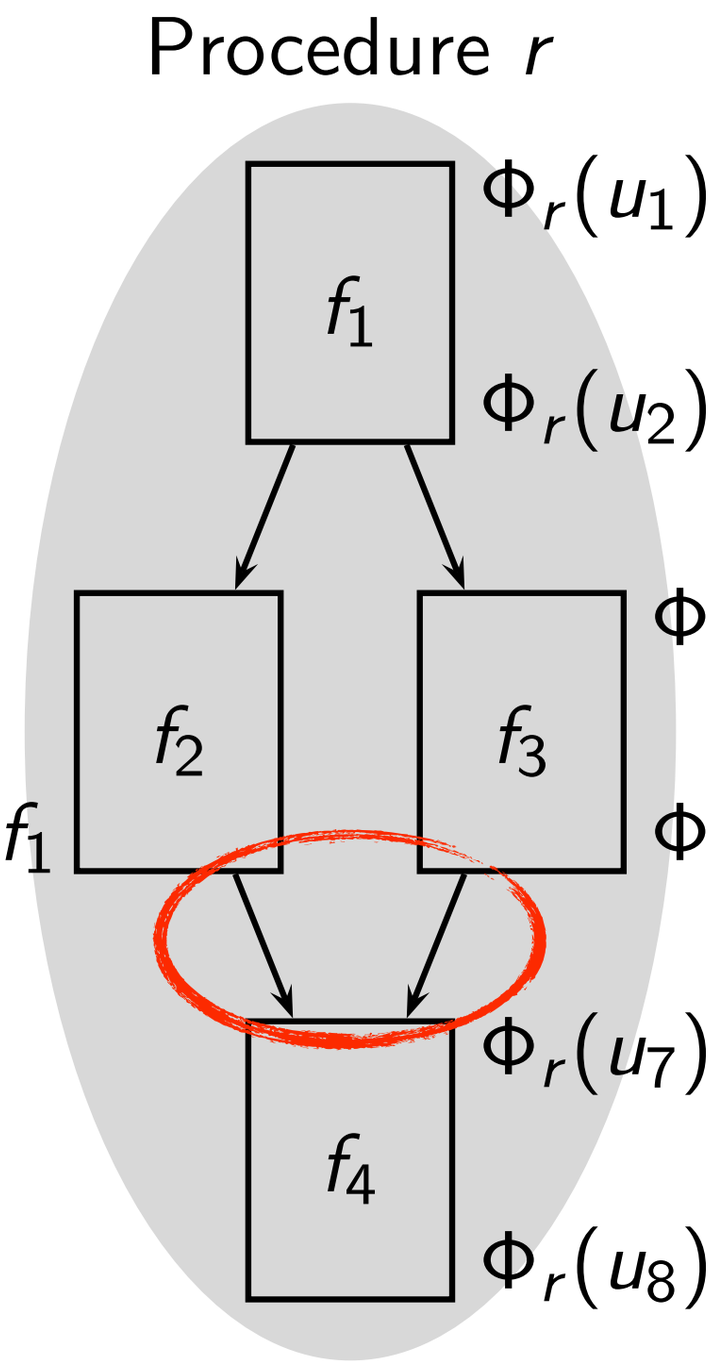
much more compact!

Lessons

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**

How to compute summaries?

assume forward analysis…



$\Phi_r(u_1) \equiv \phi_{id}$

$\Phi_r(u_2) \equiv f_1$

$\Phi_r(u_3) \equiv f_1$

$\Phi_r(u_4) \equiv f_1$

$\Phi_r(u_5) \equiv f_2 \circ f_1$

$\Phi_r(u_6) \equiv f_3 \circ f_1$

$\Phi_r(u_7) \equiv f_2 \circ f_1 \sqcap f_3 \circ f_1$

$\Phi_r(u_8) \equiv f_4 \circ (f_2 \circ f_1 \sqcap f_3 \circ f_1)$

Slide uses illustration by Uday Khedker

Generally assume that flow functions are **monotone**:

$$f(x) \sqcap f(y) \sqsubseteq f(x \sqcap y)$$

To be able to compute intensional summaries, they **need to be distributive**:

$$f(x) \sqcap f(y) = f(x \sqcap y)$$

IFDS, IDE and WPDS compute summaries automatically, but care must be taken when choosing the appropriate heap abstraction!

two allocation sites

```
void main() {
    A a1 = new A(); //alloc1
    A a2 = new A(); //alloc2
```

allocation-site abstraction causes duplicated analysis of callee

```
//context c1
//context c2
```

```
<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
```

({alloc1},o)

({alloc1},<ret>)

({alloc2},o)

({alloc2},<ret>)

```
void main() {
    A a1 = new A(); //alloc1
    A a2 = new A(); //alloc2
}
```

problem: summary for `alloc1` is not reusable for `alloc2`

```
<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
```

`//context c1`

`//context c2`

`({alloc1},o)`

`({alloc2},o)`

`({alloc1},<ret>)`

`({alloc2},<ret>)`

Instead use access paths

l..k field accesses

l . f . g . h

local variable

```
void main() {
    A a1 = new A(); //alloc1
}

<T> T foo(T o) {
    //some hard-to-analyze code omitted
    return o;
}
```

1st advantage:
much smaller abstract domain!

context c1

context c2

o

<ret>

```
void main() {
    A a1 = new A(); //alloc1
    A a2
    A a1                                    c1
    A a2                                    c2
}

<T> T foo(T o) {                o
    //some hard-to-analyze code omitted
    return o;
}                               <ret>
```

2nd advantage:
analyzing foo only once!

Lessons

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**
5. The point of procedure summaries is not to be able to create them but to be able to **reuse** them!
6. Alloc-site-based heap abstraction **hinder summary reuse**
7. Access paths **foster reusable summaries**

Remember:

To be able to compute intensional summaries, they **need to be distributive**:

$$f(x) \sqcap f(y) = f(x \sqcap y)$$

But: distributive frameworks yield **perfect precision**!

meet over all paths

maximal fixed point

$$\forall s.\ MOP(s) \sqsubseteq MFP(s)$$

$$f(x) \sqcap f(y) \sqsubseteq f(x \sqcap y)$$

$$\forall s.\ MOP(s) = MFP(s)$$

$$f(x) \sqcap f(y) = f(x \sqcap y)$$

$$\forall s.\ MOP(s) \sqsubseteq MFP(s)$$

$$f(x) \sqcap f(y) \sqsubseteq f(x \sqcap y)$$

merging earlier gives
an over-approximation

$$f(x) \sqcap f(y) = f(x \sqcap y)$$

$$\forall s.\ MOP(s) \sqsubseteq MFP(s)$$

it does not matter
when we merge!

$$\forall s.\ MOP(s) = MFP(s)$$

$$f(x) \sqcap f(y) = f(x \sqcap y)$$

two calling contexts

```
void main()
    Pair<String> p1 =
        makePair(secretKey(),secretKey());
    Pair<String> p2 =
        makePair(secretKey(),publicKey());
}

<T> T makePair(T a, T b) {
    Pair<T> ret = new Pair<T>();
    ret.left = a;
    ret.right = b;
    return ret;
}
```

two taints

```
void main() {
    Pair<String> p1 =
        makePair(secretKey(),secretKey());
    Pair<String> p2 =
        makePair(secretKey(),publicKey());
}
```

one taint

```
<T> T
    Pair<T> ret = new Pair<T>();
    ret.left = a;
    ret.right = b;
    return ret;
}
```

at first context

```
void main()
    Pair<String> p1 =
        makePair(secretKey(),secretKey());
    Pair<String> p2 =
        makePair(secretKey(),publicKey());
}

<T> T makePair(T a, T b) {
    Pair<T> ret = new Pair<T>();
    ret.left = a;
    ret.right = b;
    return ret;
}
```

a

b

<ret>.left

<ret>.right

at second context

```
void main() {
    makePair(secretKey(),secretKey());
    Pair<String> p2 =
    makePair(secretKey(),publicKey());
}

<T> T makePair(T a, T b) {
    Pair<T> ret = new Pair<T>();
    ret.left = a;
    ret.right = b;
    return ret;
}
```

a

b

<ret>.left

<ret>.right

can reuse this partial summary!

sound only due to distributivity!

```
void main() {
    Pair<String> p1 =
        makePair(secretKey(),secretKey());
    Pair<String> p2 =
        makePair(secretKey(),publicKey());
}

<T> T makePair(T a, T b) {
    Pair<T> ret = new Pair<T>();
    ret.left = a;
    ret.right = b;
    return ret;
}
```

a → <ret>.left

b → <ret>.right

can reuse this partial summary!

Lessons

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**
5. The point of procedure summaries is not to be able to create them but to be able to **reuse** them!
6. Alloc-site-based heap abstraction **hinder summary reuse**
7. Access paths **foster reusable summaries**
8. Intensional summaries **require distributivity**
9. Distributivity yields **perfect precision**
10. Distributive summaries can be **more effectively reused**

My 10 laws of precise and efficient analysis

1. **Precise pointer analyses** are key!
2. **Demand-driven analysis** can aid both performance and precision
3. **Summaries** avoid repeated re-analysis of callee procedures
4. One should strive for **intensional summary definitions**
5. The point of procedure summaries is not to be able to create them but to be able to **reuse** them!
6. Alloc-site-based heap abstraction **hinder summary reuse**
7. Access paths **foster reusable summaries**
8. Intensional summaries **require distributivity**
9. Distributivity yields **perfect precision**
10. Distributive summaries can be **more effectively reused**

Valid until disproven ! ;-)

We are hiring!

PostDocs
and Ph.D. students