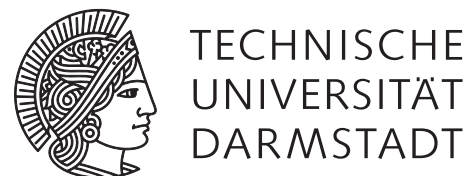
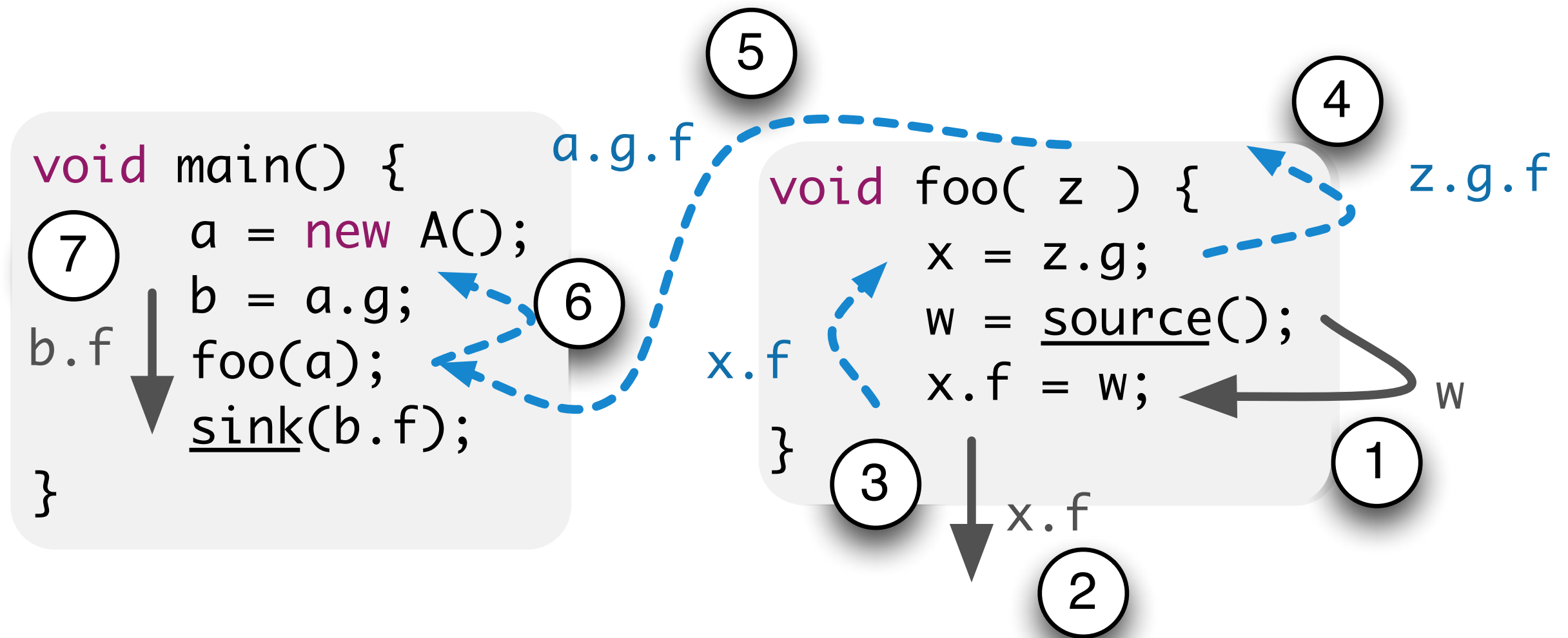


Automated Code Analysis for Large Software Systems (ACA)

Eric Bodden

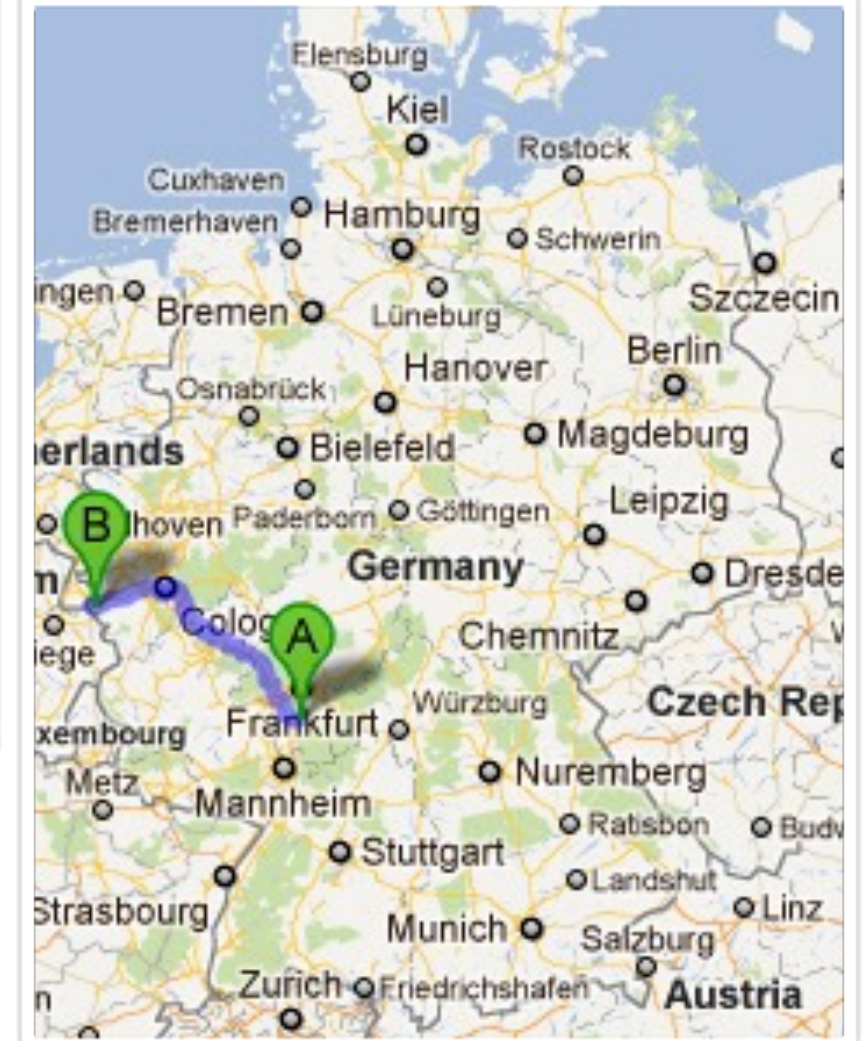


<http://sseblog.ec-spride.de/aca/>



Will it leak?

Studied in Aachen



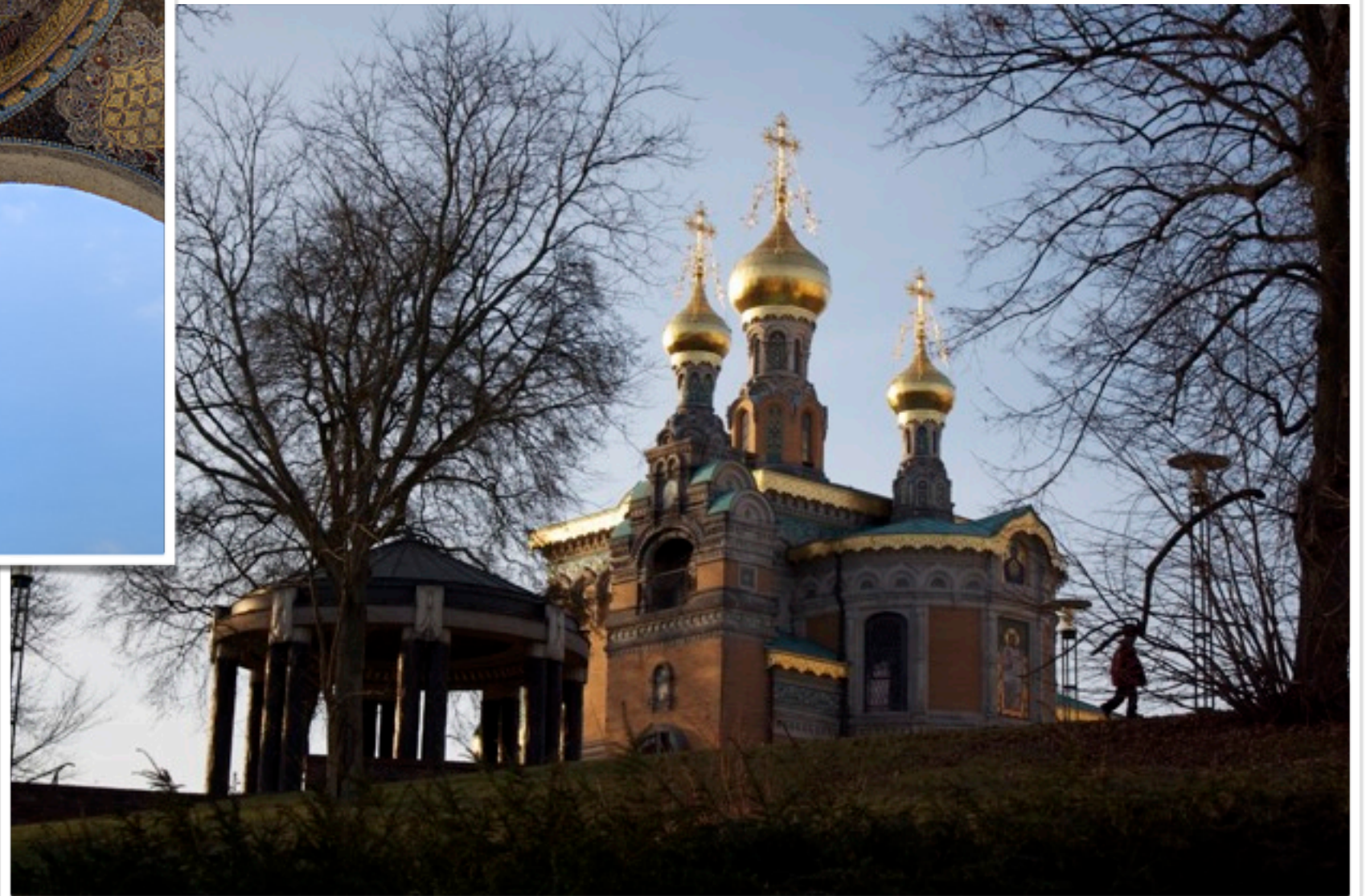
ERASMUS in Canterbury, UK



Ph.D. in Montréal, Québec

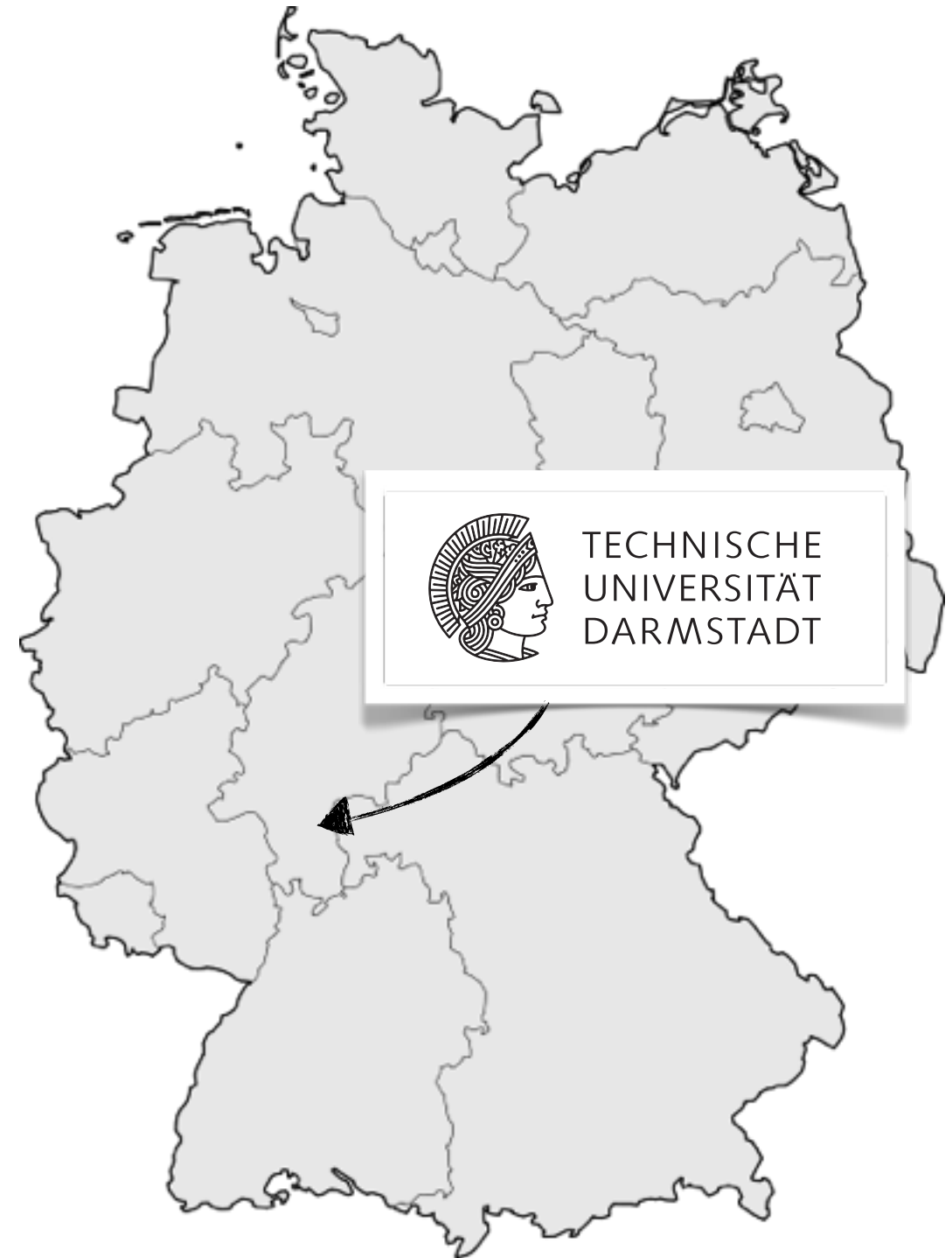


In Darmstadt since 2009



Since 2009:  **CATED**

Since 2011:  EC SPRIDE



Software Security

Legacy Code

Now

Code Analyses



“By Design”



New
Opportunities!

The future

Novel
programming
models

Google



Vulnerability
Detection



Find
Privacy Leaks

Runtime
Enforcement

Simpler Secure
Programming Models

Engineering

Static
Program Analysis

Security related
Programming Models

SPL
Analysis

Mitigation of
Timing Channels

Buffer-overflow
Mitigation



X86

Two institutions, one group



TECHNISCHE
UNIVERSITÄT
DARMSTADT

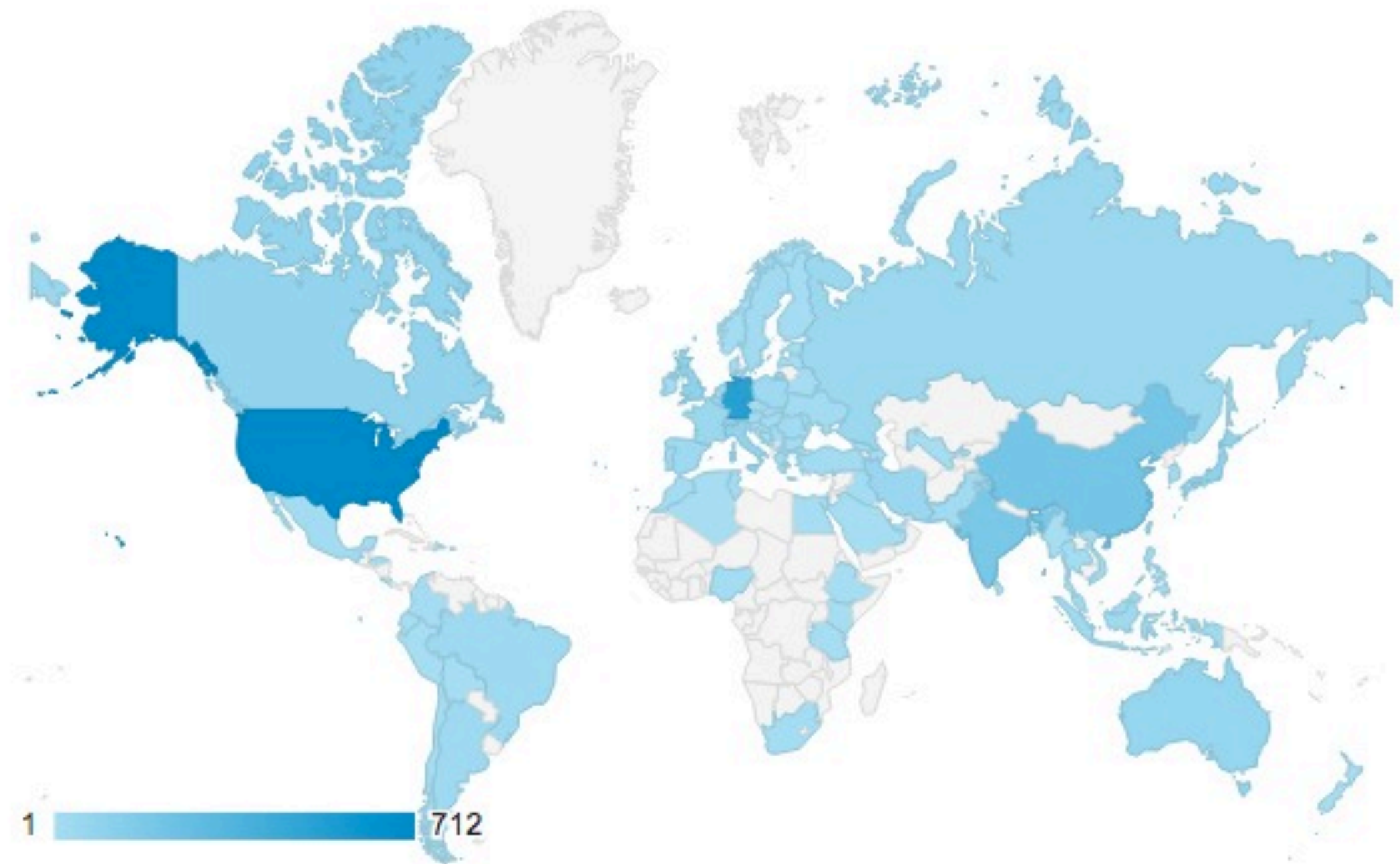


SECURE
SOFTWARE ENGINEERING
GROUP



Fraunhofer
SIT

Develop popular tools



External Partners



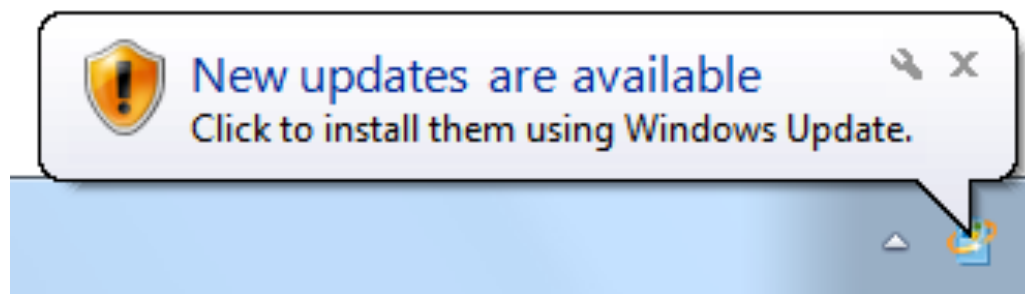
**What this lecture
is about...**



Firewalls



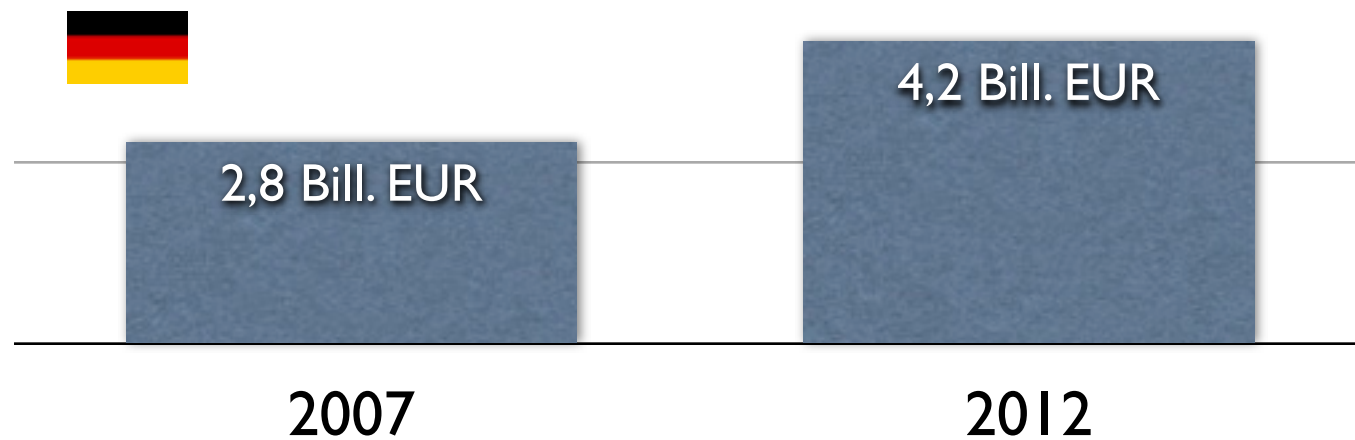
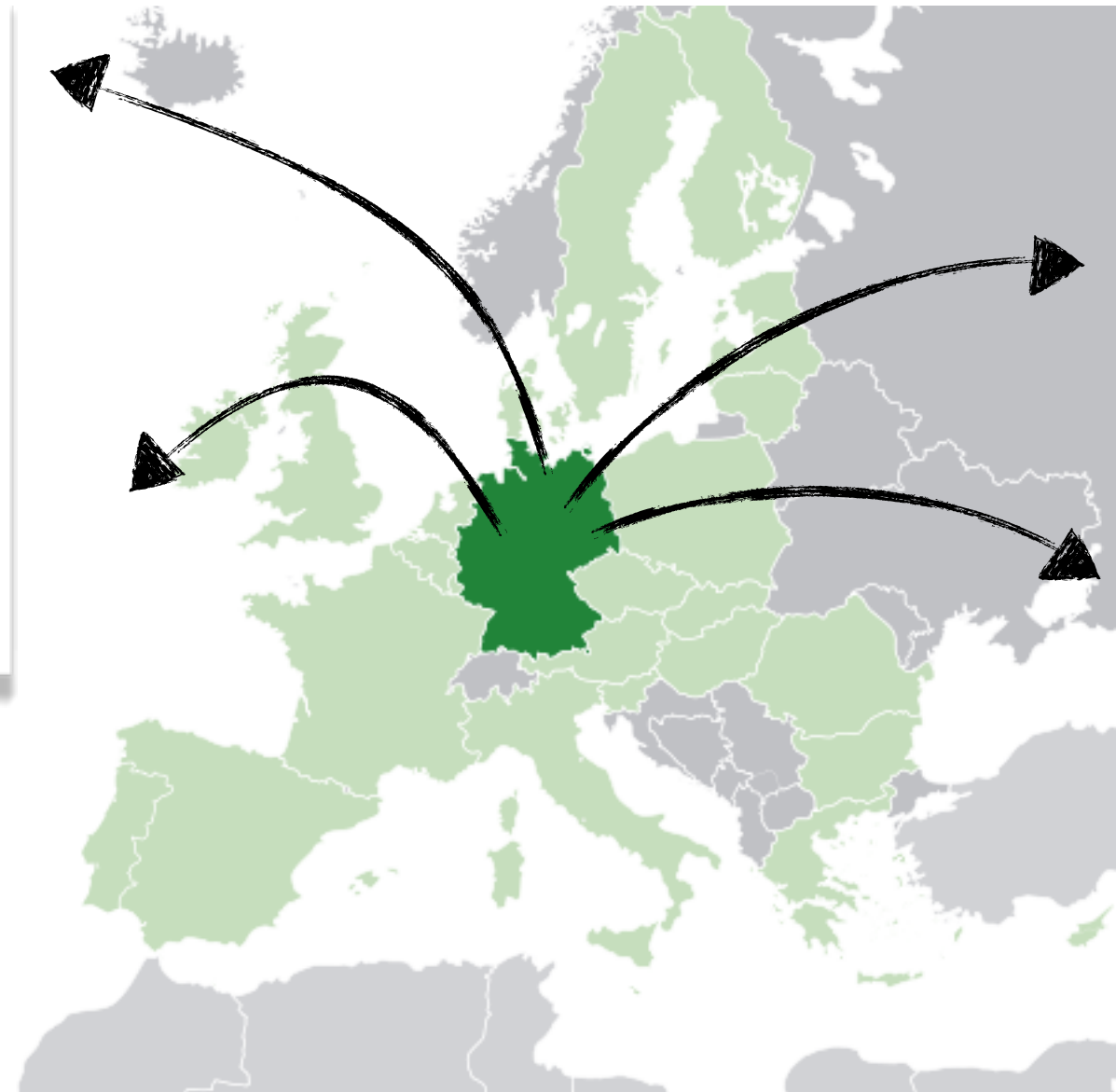
Antivirus



Updates

Often applied only
after months or
even years

Cyber espionage

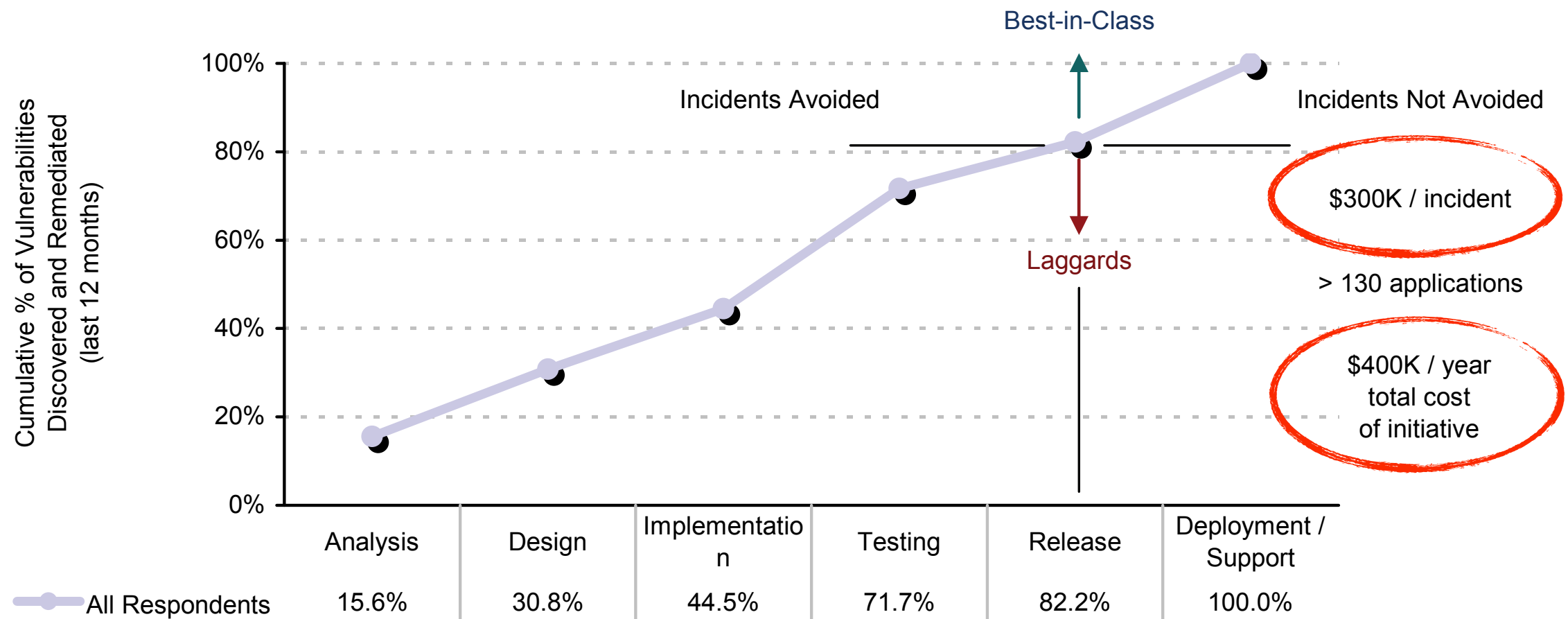


Source: Corporate Trust, TÜV Süd, 2012

Cyber warfare

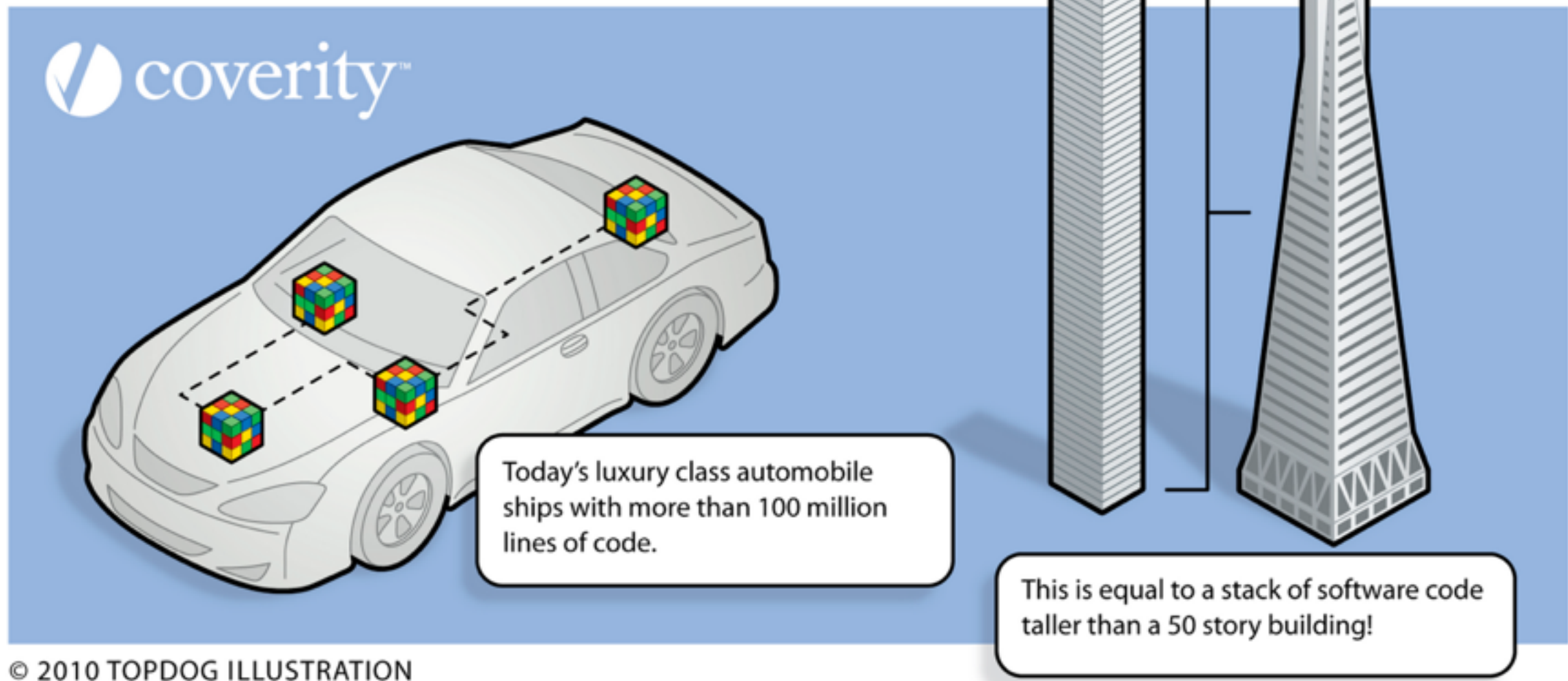


Pays off to find security

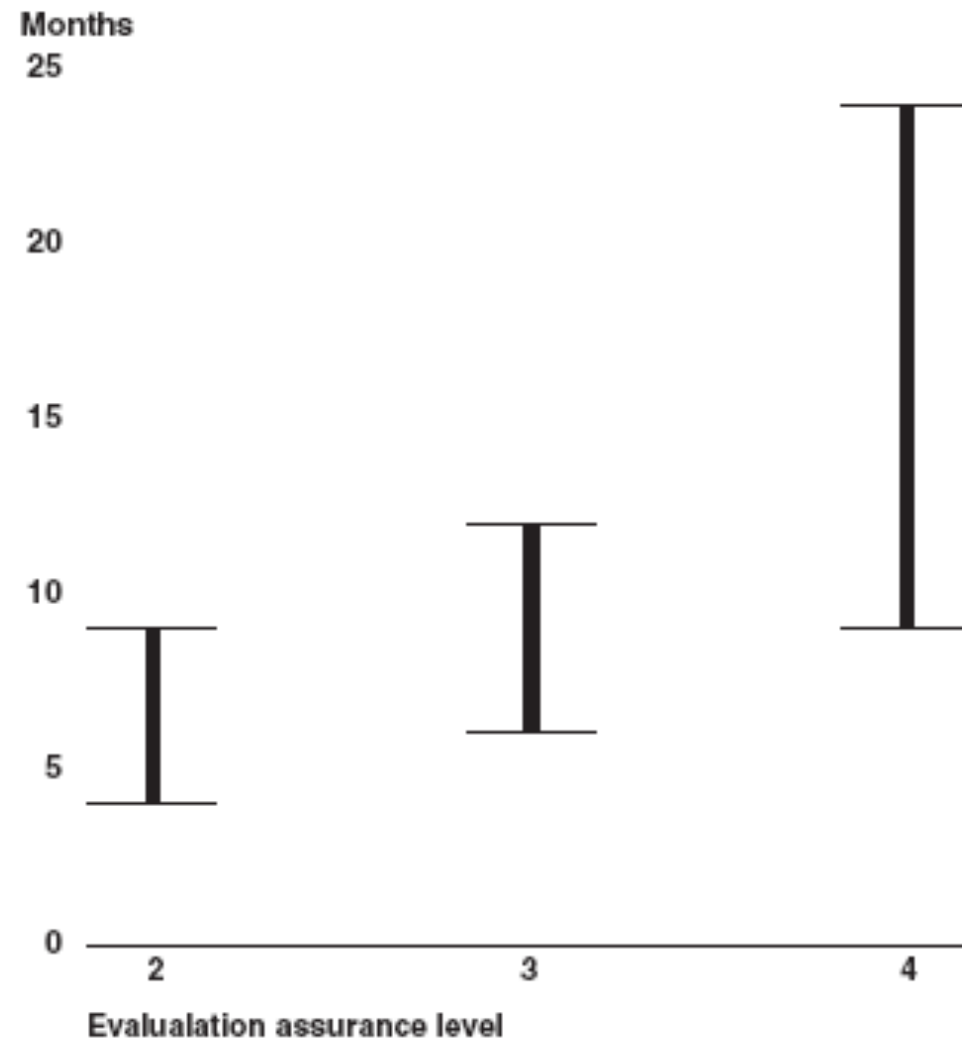


Source: Aberdeen Group, September 2010

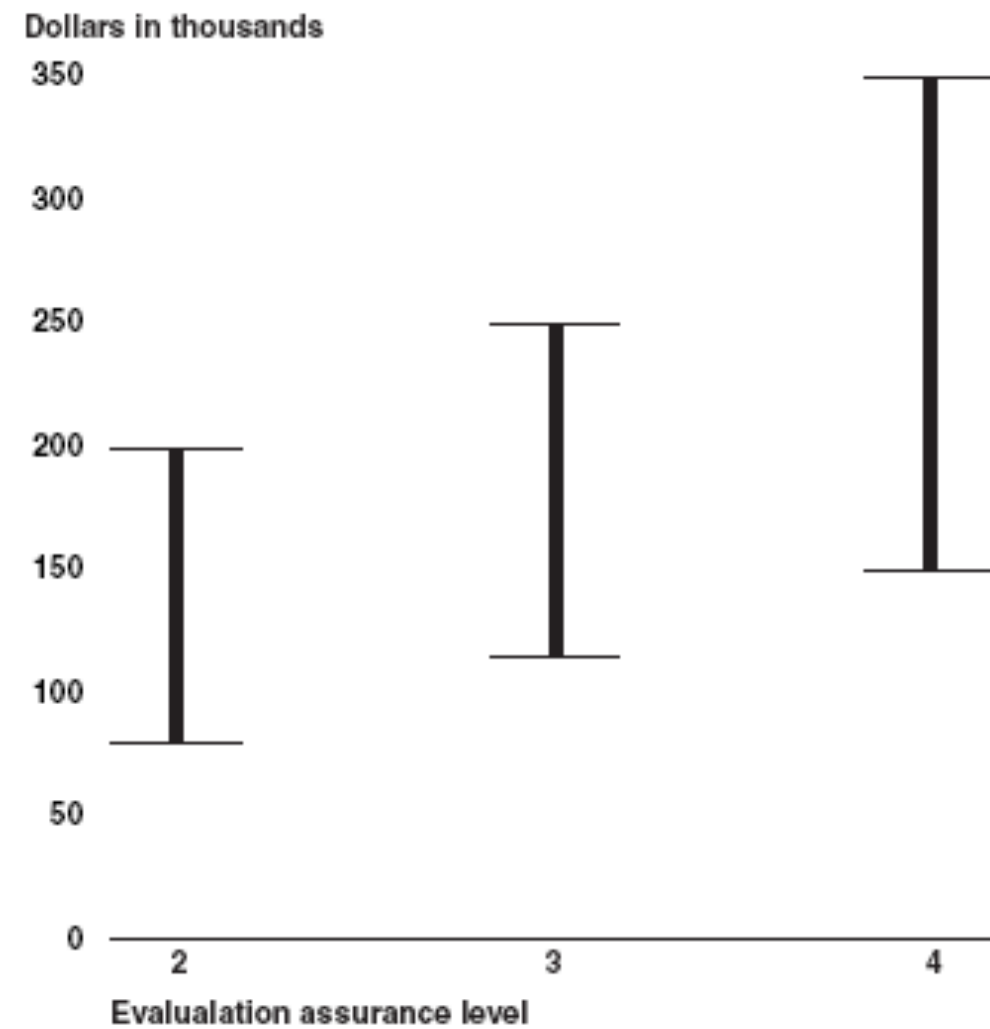
Problem: Complexity



Common-Criteria Certification too expensive, too ineffective

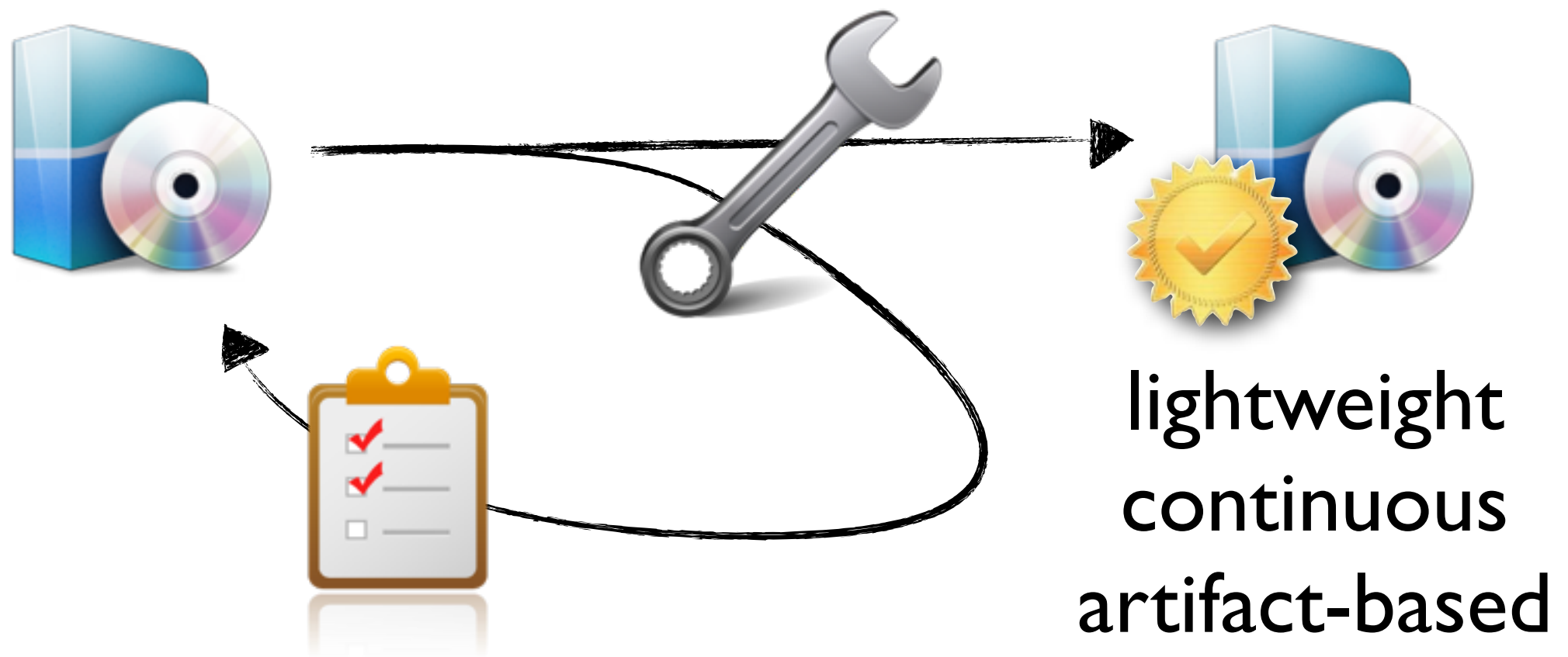


Source: GAO analysis of data provided by laboratories.

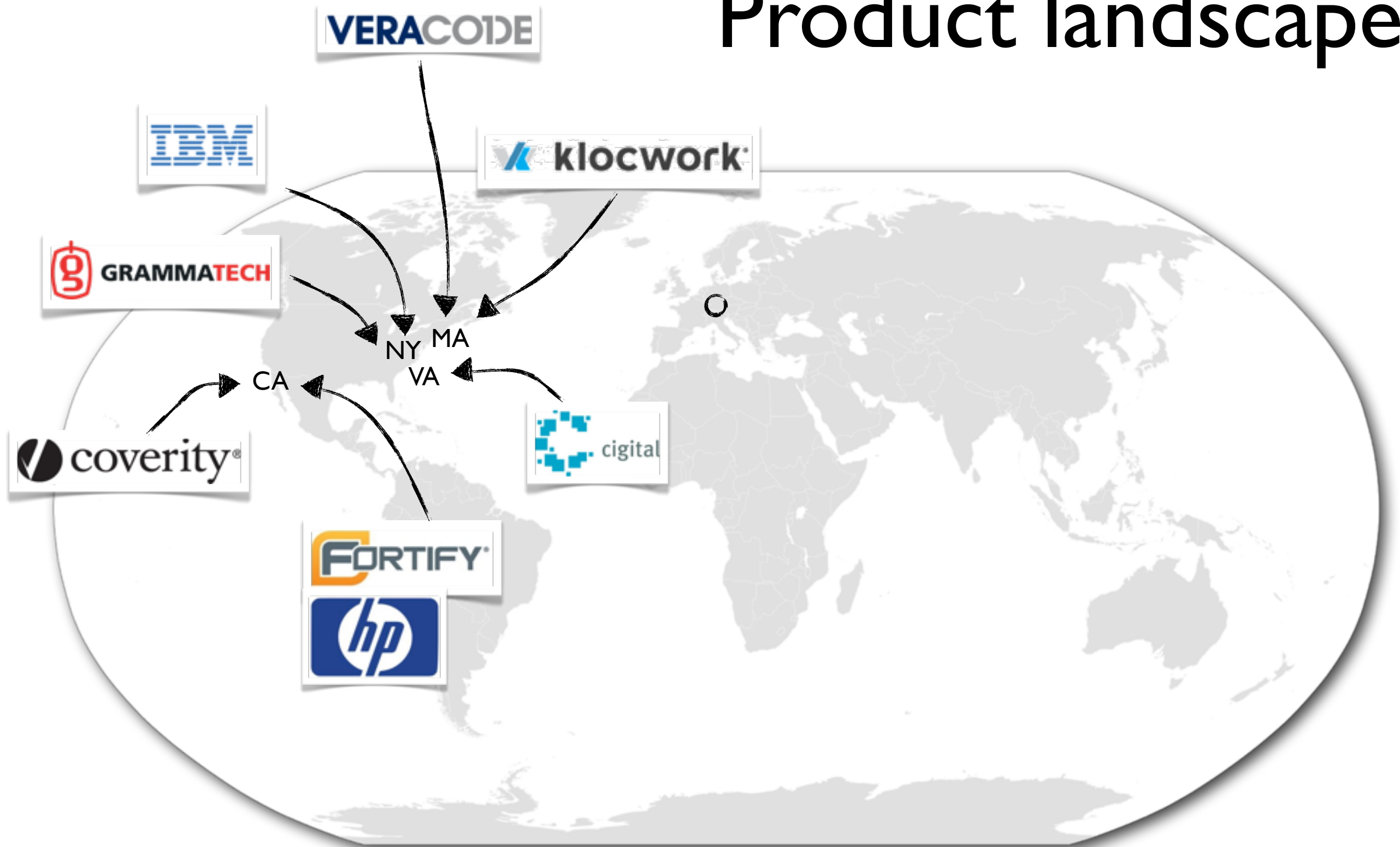


Source: GAO analysis of data provided by laboratories.

Alternative solution discussed here: Automated code analysis



Product landscape



Two attacker models

Goodware

programmer on our side

no obfuscation

but: vulnerabilities

detectable statically

Malware

expect the worst

often obfuscated

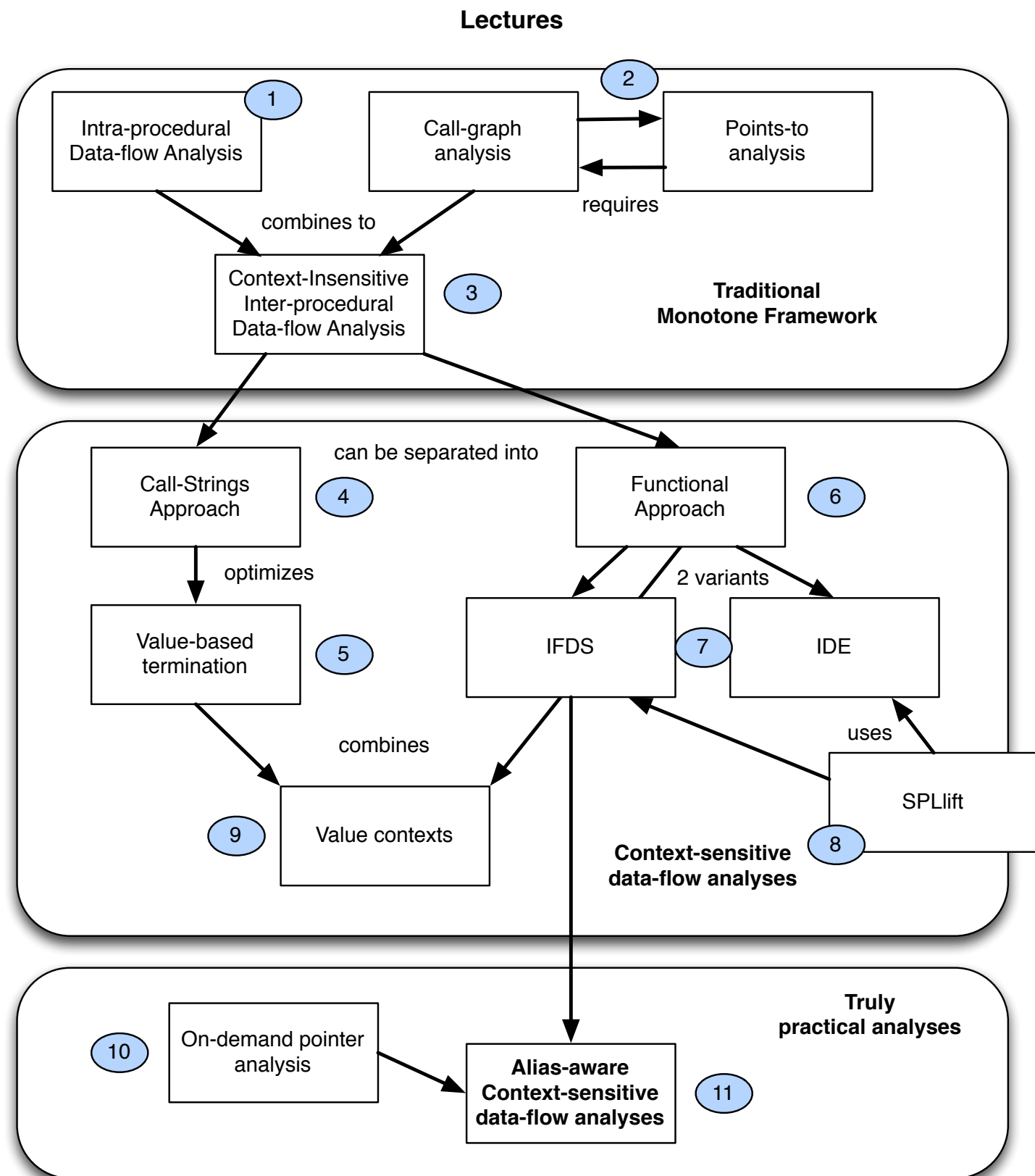
multiple prog. languages

detectable at runtime

Lecture outline

Preliminary Outline

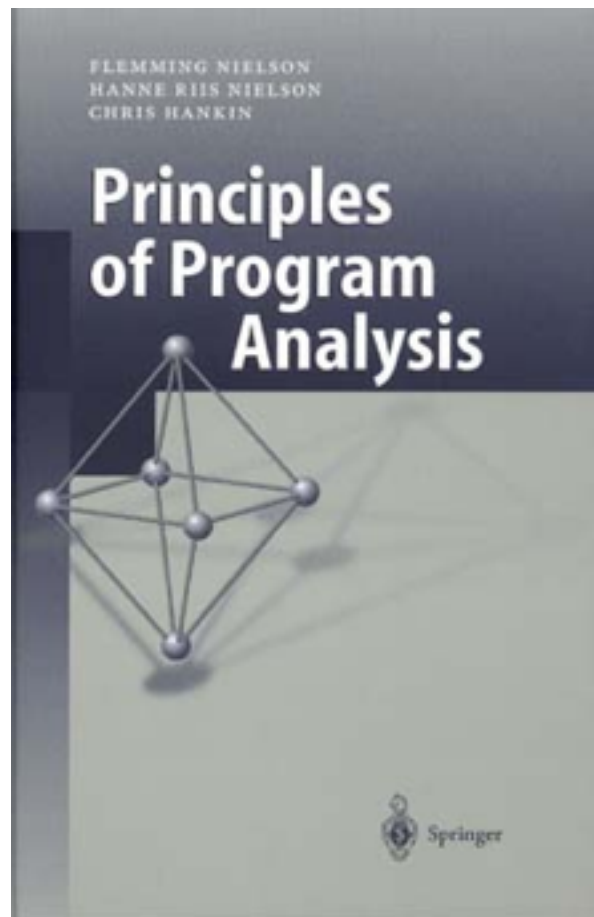
+ special topics



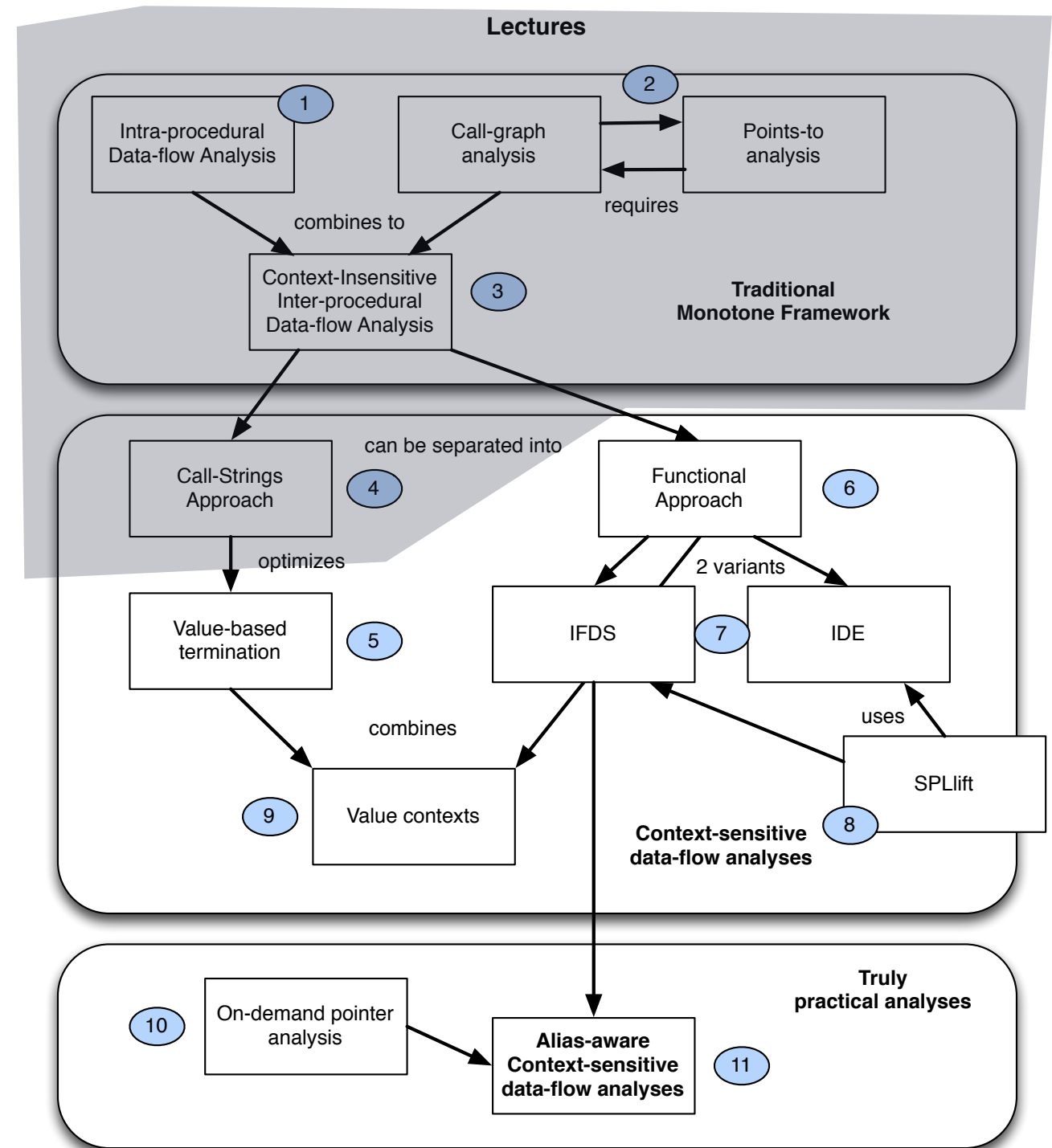
This will be a practical lecture

- Many examples
- Important algorithms and ideas behind them
 - Why do they work? When do they work best?
- Programming exercises
- No hard proofs

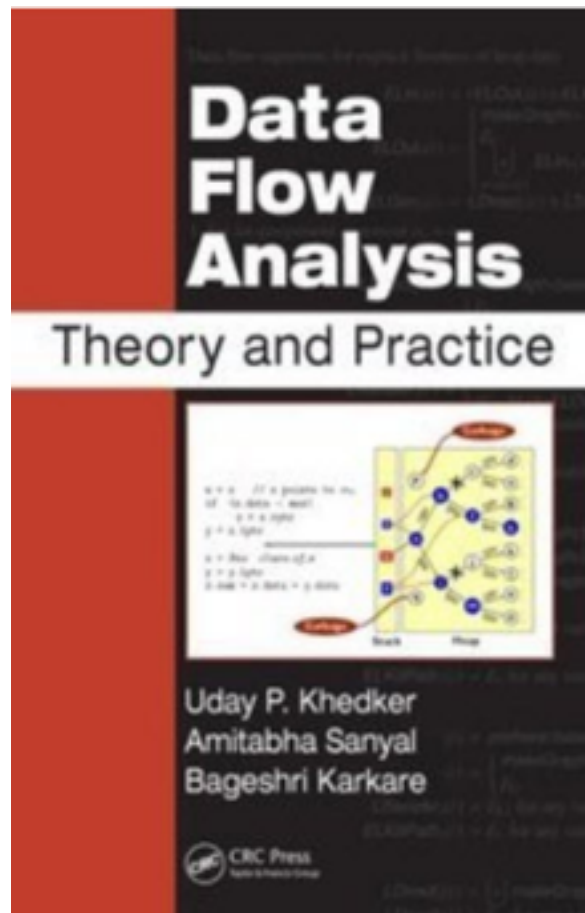
For further reading...



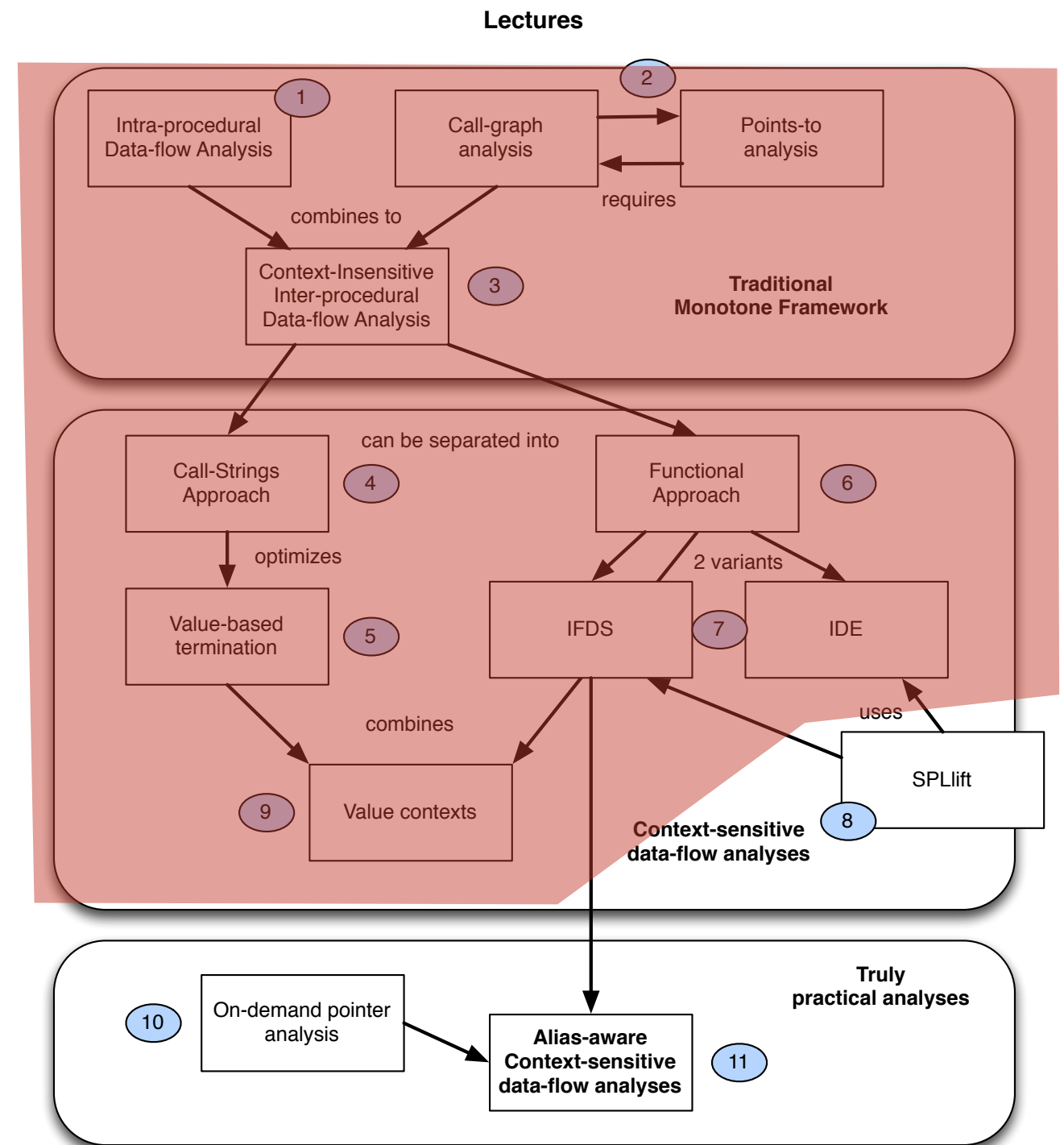
Quite formal
Focuses on
“call-strings approach”



For further reading...



More applied
Focuses on
“functional approach”



Course Setup

- About 90 minutes (almost) each Thursday
- 60min+ lecture with short break
- 20-30min discussion of exercises (when exercises are due)

Signing up

- In TUCaN, sign up both for the module and for the lecture!
- Module: 20-02-0732
- Lecture: 20-02-0732-iv

Course SVN repository

- <https://repository.st.informatik.tu-darmstadt.de/sse/aca/2013/>
 - exercise-sheets/
 - public/ - code templates etc.
 - slides/
 - students/ - your private space for submission of solutions

Exercises

- There will be 7 exercise sheets.
- Each sheet is pass/fail. If pass, bonus = 0.2.
- Maximal bonus is 1, i.e., 5 sheets suffice.
But: all sheets relevant for exam!
- Bonus cannot be used to pass exam!
- Exercises usually due on Tuesday before next sheet is given out (23:59).
- Hand in using Version Control, not Email!

Exercises - SVN Setup

- Find group partners. (Here or using forum.)
 - There should be three people per group.
- Use SVN to create group directory at:
<https://repository.st.informatik.tu-darmstadt.de/sse/aca/2013/students/>
The directory's should consist of your last names, e.g. "arzt-bodden-rasthofer".
- Email the following data to steven.arzt@cased.de
 - your directory name, your names, your Student ID numbers, your RBG login names (!), and your email addresses (!)
- We will then secure your directory and email you back.

DEADLINE: Oct 22nd

Exercises - Handing in

- Check in certain files (details on exercise sheets)
- No need to email us, just check in by the deadline!
- We will push your grade (pass/fail) and comments into your group directory.

Exercises - Discussion

- For questions please use the forums.
- May also ask questions after each lecture.
- I will try to discuss the solution to each sheet on the day the next sheet is given out.

Optional exercises

- There are optional exercises, which are, as the name suggest, optional. But not quite...
- If you fail a sheet (or fail to hand one in), you can make up for it:
 - complete two optional exercises from one or more upcoming sheets

Course Notes

- There will be no fully-fledged script.
- I will provide, though...
 - all slides, and
 - essential notes, e.g. of algorithms, and links to background reading
- Material will be in SVN, password protected

Contacting me

- Please use the forum!
- No office hours:
use email or make appointment

Questions?

What we will cover today

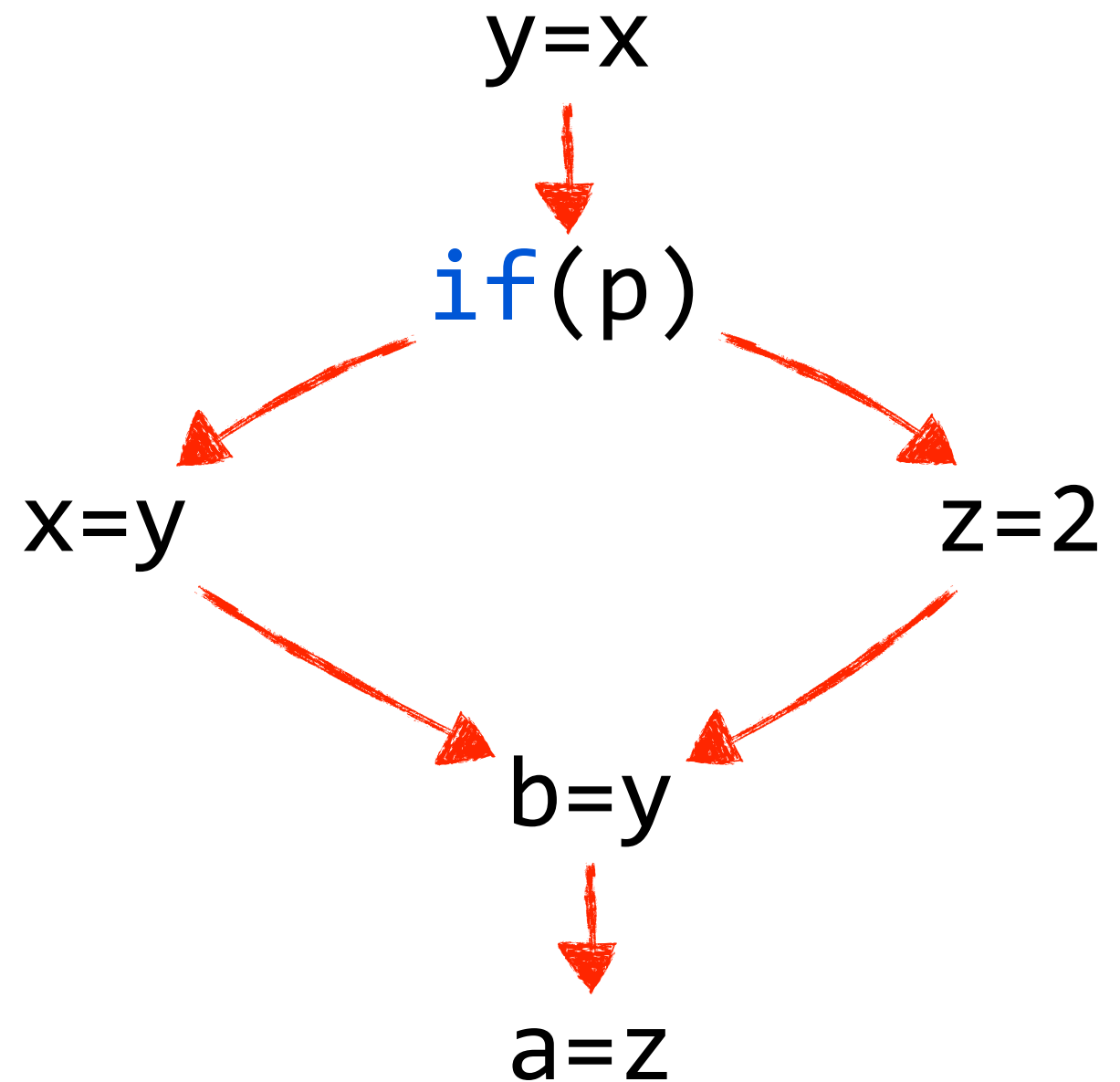
Not quite source code, not quite bytecode:
intermediate representations
for static analysis

General Workflow

- Parse method (as source code or bytecode) and convert into control-flow graph (CFG)
 - Nodes: Simplified statements
 - Edges: Possible control flows between such statements

Example

```
y=x;  
if(p) x=y;  
else z=2;  
b=y;  
a=z;
```



In general, CFG is an over-approximation

```
if(isPrime(2312321)) ..  
if(!isPrime(2312321)) ..
```

```
y=x;  
if(p) x=y;  
if(!p) z=2;  
b=y;  
a=z;
```

x=y

z=2

y=x

if(p)

if(!p)

b=y

a=z

depending on how complex
predicate p is, cannot infer that
branches are mutually exclusive

Lesson learned

- Almost always, control-flow graphs are conservative:
 - if control may flow from statement a to statement b then there is an edge from a to b
 - opposite is not true!
 - this problem cannot be solved (undecidable)
- Real-life CFGs will even contain edges for exceptional control-flow (otherwise unsound)

Important design decision

What statements/nodes
to allow or not?

One extreme: Java source code

Problem: statements (and classes) can be nested...

```
for(..) {  
    for(..) {  
        new Comparator() {  
            public int compareTo(..) {  
                ... and so on  
            }  
        }  
    }  
}
```

Other extreme: Java bytecode

- Advantages:

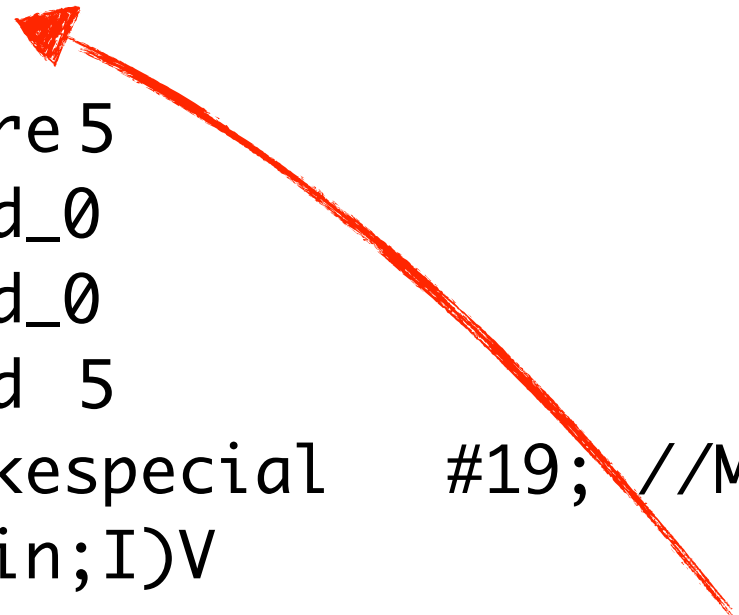
- no nesting; one statement follows the other; looping/branches through jumps (goto)
- nested classes are “flattened” into normal classes

- Disadvantages:

- No local variables: operations performed on operand stack
- More than 200 possible bytecodes!

Other extreme: Java bytecode

```
ldc2_w #15; //double 3.0d
dstore_1
ldc2_w #17; //double 2.0d
dstore_3
dload_1
dload_3
dmul
d2i
istore 5
aload_0
aload_0
iload 5
invokespecial #19; //Method bar:
(LMain;I)V
return
```



```
void foo() {
    double d1 = 3.0;
    double d2 = 2.0;
    int i1 = (int) (d1*d2);
    bar(this, i1);
}
```

pop and multiply two top operands on the stack;
place result on stack again

Other extreme: Java bytecode

```
ldc2_w #15; //double 3.0d
```

```
dstore_1
```

```
ldc2_w #17; //double 2.0d
```

```
dstore_3
```

```
dload_1
```

```
dload_3
```

```
dmul
```

```
d2i
```

```
istore_5
```

```
aload_0
```

```
aload_0
```

```
iload_5
```

```
invokespecial #19; //Method bar:
```

```
(LMain;I)V
```

```
return
```

```
void foo() {  
    double d1 = 3.0;  
    double d2 = 2.0;  
    int i1 = (int) (d1*d2);  
    bar(this, i1);  
}
```

many overloaded versions of essentially
the same operation

Android Bytecode

- Similar to Java bytecode but...
- Logical registers instead of operand stack
- Some values are untyped
 - example: the type of numerical constants is not known before their first use
- Roughly 250 bytecodes
 - including Optimized DEX (ODEX)

Intermediate Representation: Jimple

- Jimple = “like **J**ava, but **simple**”
- Combines the best of both worlds
 - Local variables, like in source code
 - no stack operations
 - Special variables for “this” and parameters
 - Only simple statements, never nested

Golden mean: Jimple IR

```
void foo()  
{  
    Main this;  
    double d1, d2, temp$0;  
    int i1;  
  
    this := @this: Main;  
    d1 = 3.0;  
    d2 = 2.0;  
    temp$0 = d1 * d2;  
    i1 = (int) temp$0;  
    virtualinvoke this.<Main: void bar(Main,int)>(this, i1);  
    return;  
}
```

```
void foo() {  
    double d1 = 3.0;  
    double d2 = 2.0;  
    int i1 = (int) (d1*d2);  
    bar(this, i1);  
}
```

all variables explicitly declared, even “this”

Golden mean: Jimple IR

```
void foo()  
{  
    Main this;  
    double d1, d2, temp$0;  
    int i1;  
  
    this := @this: Main;  
    d1 = 3.0;  
    d2 = 2.0;  
    temp$0 = d1 * d2;  
    i1 = (int) temp$0;  
    virtualinvoke this.<Main: void bar(Main,int)>(this, i1);  
    return;  
}
```

```
void foo() {  
    double d1 = 3.0;  
    double d2 = 2.0;  
    int i1 = (int) (d1*d2);  
    bar(this, i1);  
}
```

special references for “this” and parameters

Golden mean: Jimple IR

```
void foo()  
{  
    Main this;  
    double d1, d2, temp$0;  
    int i1;  
  
    this := @this: Main;  
    d1 = 3.0;  
    d2 = 2.0;  
    temp$0 = d1 * d2;  
    i1 = (int) temp$0;  
    virtualinvoke this.<Main: void bar(Main,int)>(this, i1);  
    return;  
}
```

```
void foo() {  
    double d1 = 3.0;  
    double d2 = 2.0;  
    int i1 = (int) (d1*d2);  
    bar(this,i1);  
}
```

no stack operations; instead assignments

Golden mean: Jimple IR

```
void foo()  
{  
  Main this;  
  double d1, d2, temp$0;  
  int i1;  
  
  this := @this: Main;  
  d1 = 3.0;  
  d2 = 2.0;  
  temp$0 = d1 * d2;  
  i1 = (int) temp$0;  
  virtualinvoke this.<Main: void bar(Main,int)>(this, i1);  
  return;  
}
```

```
void foo() {  
  double d1 = 3.0;  
  double d2 = 2.0;  
  int i1 = (int) (d1*d2);  
  bar(this, i1);  
}
```

I:n

“complex”
statements
broken down

at most one reference on left-hand side,
at most two references on right-hand side
=> “three-address code”

Golden mean: Jimple IR

```
void foo()  
{  
    Main this;  
    double d1, d2, temp$0;  
    int i1;
```

```
    this := @this: Main;
```

```
    d1 = 3.0;
```

```
    d2 = 2.0;
```

```
    temp$0 = d1 * d2;
```

```
    i1 = (int) temp$0;
```

```
    virtualinvoke this.<Main: void bar(Main,int)>(this, i1);
```

```
    return;
```

```
}
```

```
void foo() {  
    double d1 = 3.0;  
    double d2 = 2.0;  
    int i1 = (int) (d1*d2);  
    bar(this, i1);  
}
```

method calls fully resolved,
explicit “this” reference

Java Bytecode vs. Jimple

Bytecode

each instr. has
implicit effect on stack

no types for
stack locations

>200 kinds of
instructions

Jimple

each stmt. acts
explicitly on
named variables

types for
local variables

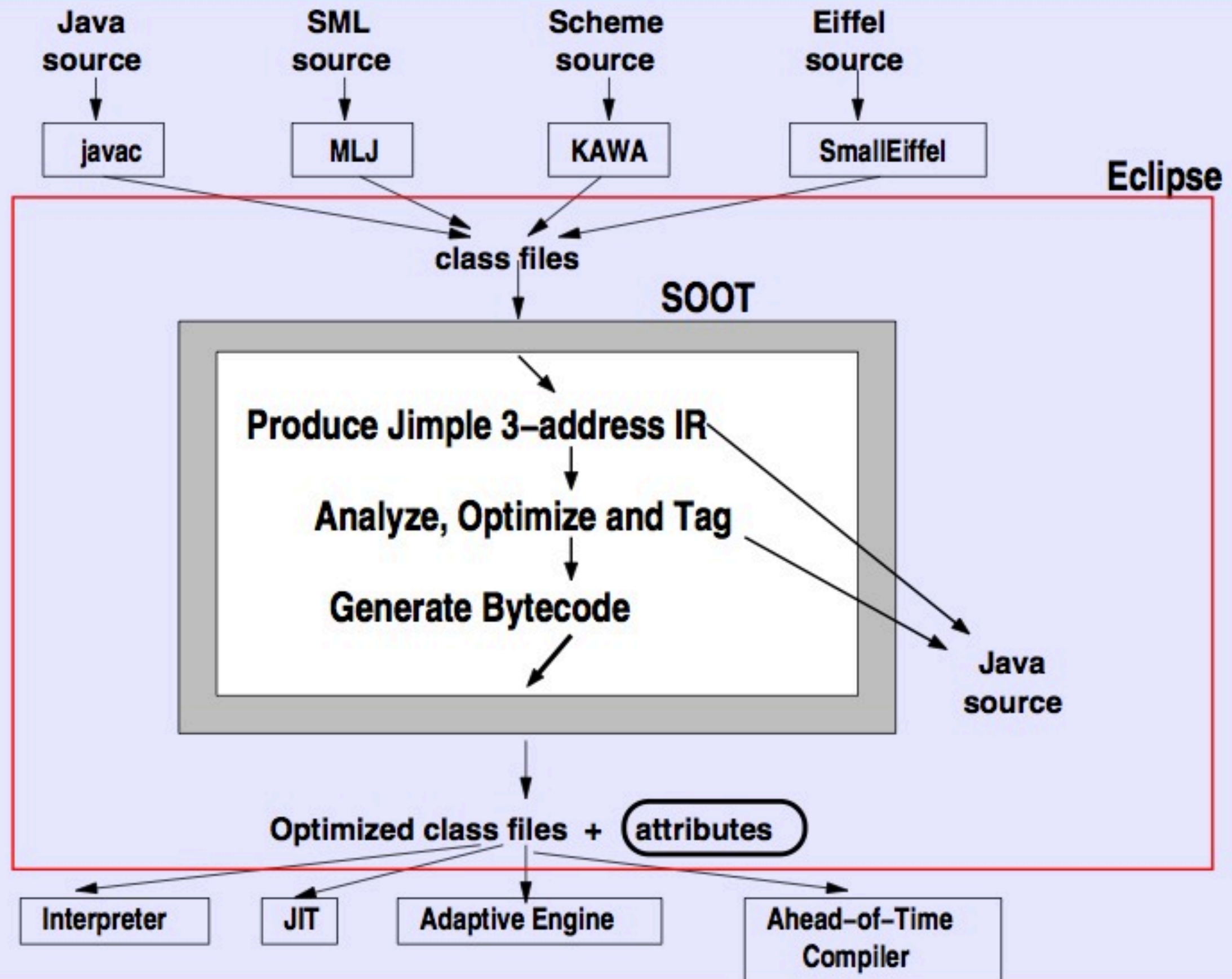
15 types of
statements

Jimple is part of Soot

- a free compiler infrastructure, written in Java (LGPL)
- was originally designed to analyze and transform Java bytecode
- original motivation was to provide a common infrastructure with which researchers could compare analyses (points-to analyses)
- has been extended to include decompilation and visualization

Soot

- Soot has many potential applications:
 - used as a stand-alone tool (command line or Eclipse plugin)
 - extended to include new IRs, analyses, transformations and visualizations
 - as the basis of building new special-purpose tools



Kinds of Jimple Stmts

- Core statements:
NopStmt
DefinitionStmt: IdentityStmt,
AssignStmt
- Intraprocedural control-flow:
IfStmt, GotoStmt,
TableSwitchStmt, LookupSwitchStmt
- Interprocedural control-flow:
InvokeStmt, ReturnStmt,
ReturnVoidStmt

Kinds of Jimple Stmts

- ThrowStmt
throws an exception
- RetStmt
not used; returns from a JSR (deprecated)
- MonitorStmt: EnterMonitorStmt,
ExitMonitorStmt
for mutual exclusion (synchronized blocks)

`this.m();`

Where's the definition of this?

IdentityStmt:

- Used for assigning parameter values and this ref to locals.
- Gives each local at least one definition point.

Simple representation of IdentityStmts:

`r0 := @this;`

`i1 := @parameter0;`

```

public int foo(java.lang.String) { // locals
    r0 := @this;                // IdentityStmt
    r1 := @parameter0;

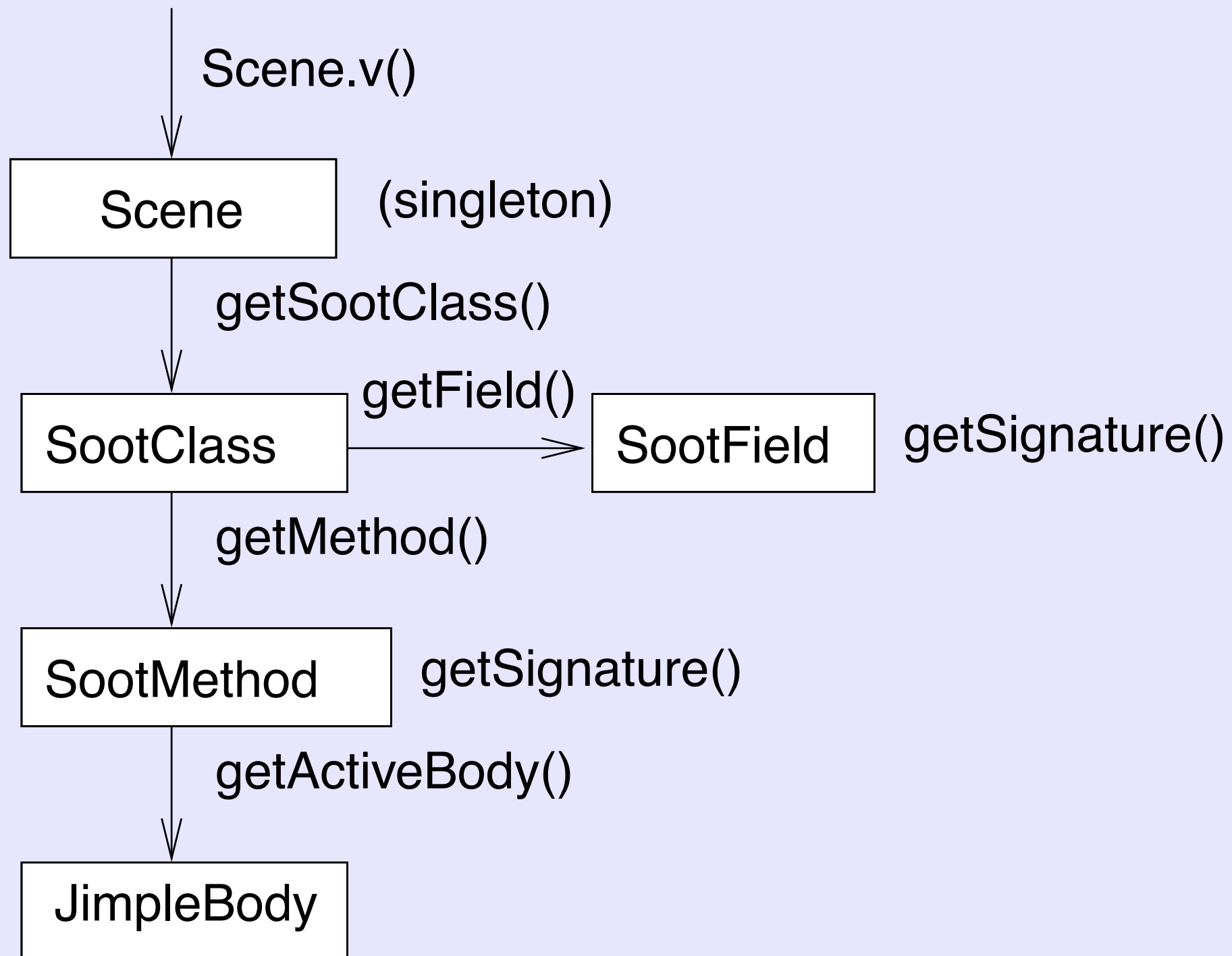
    if r1 != null goto label0; // IfStmt

    $i0 = r1.length();           // AssignStmt
    r1.toUpperCase();            // InvokeStmt
    return $i0;                  // ReturnStmt

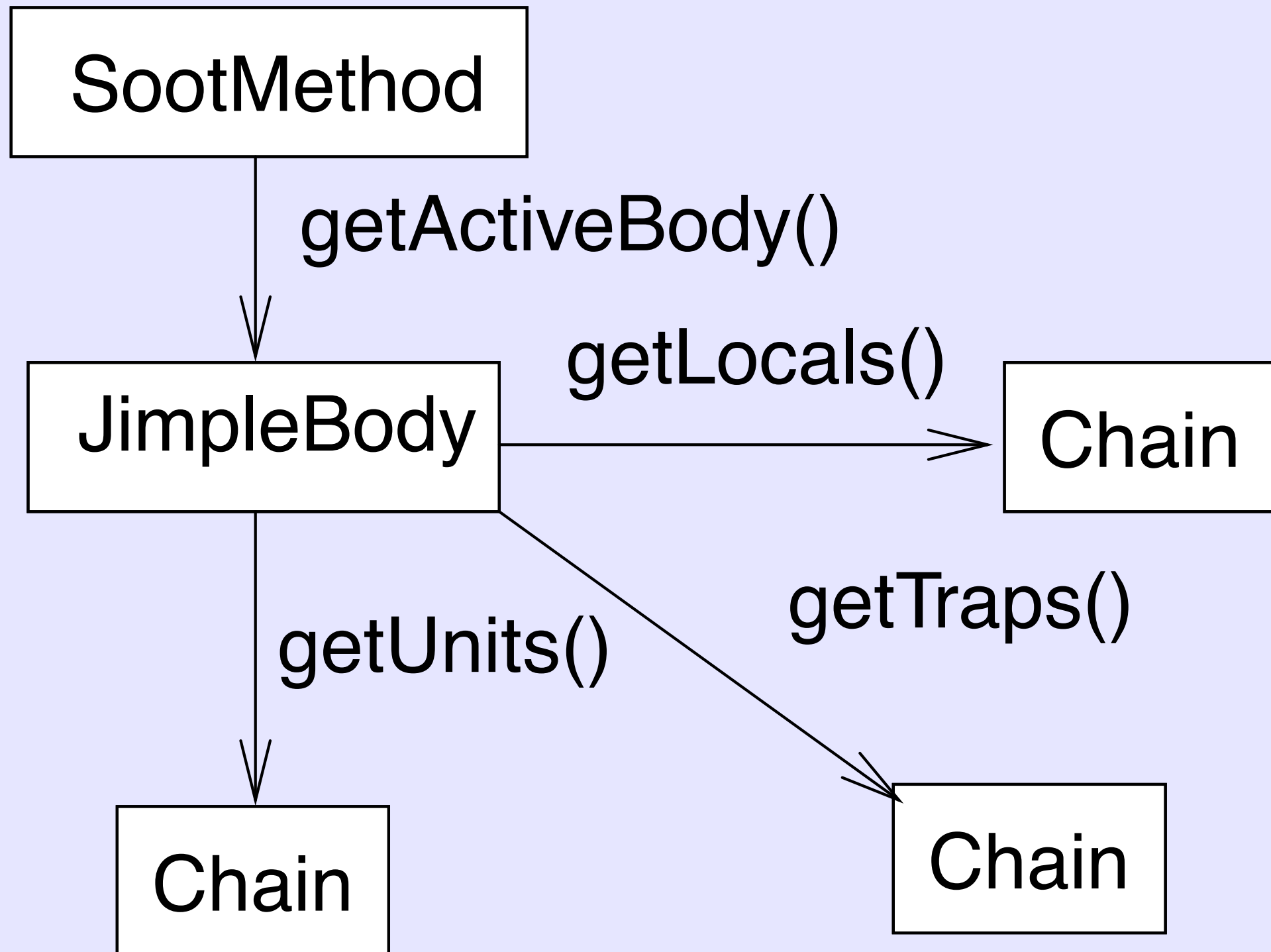
label0:                          // created by Printer
    return 2;
}

```

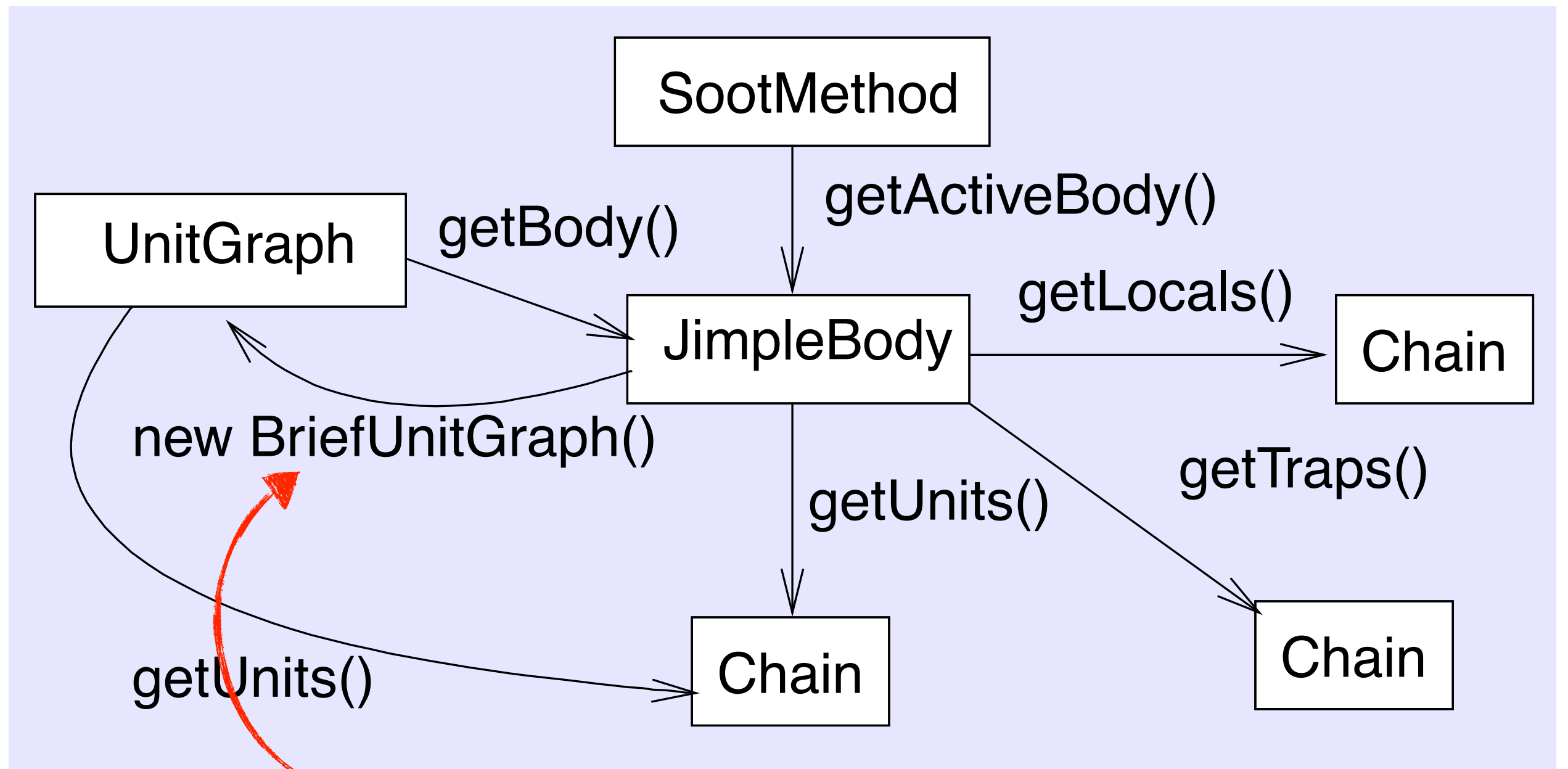
Browsing Jimple



Body-centric view



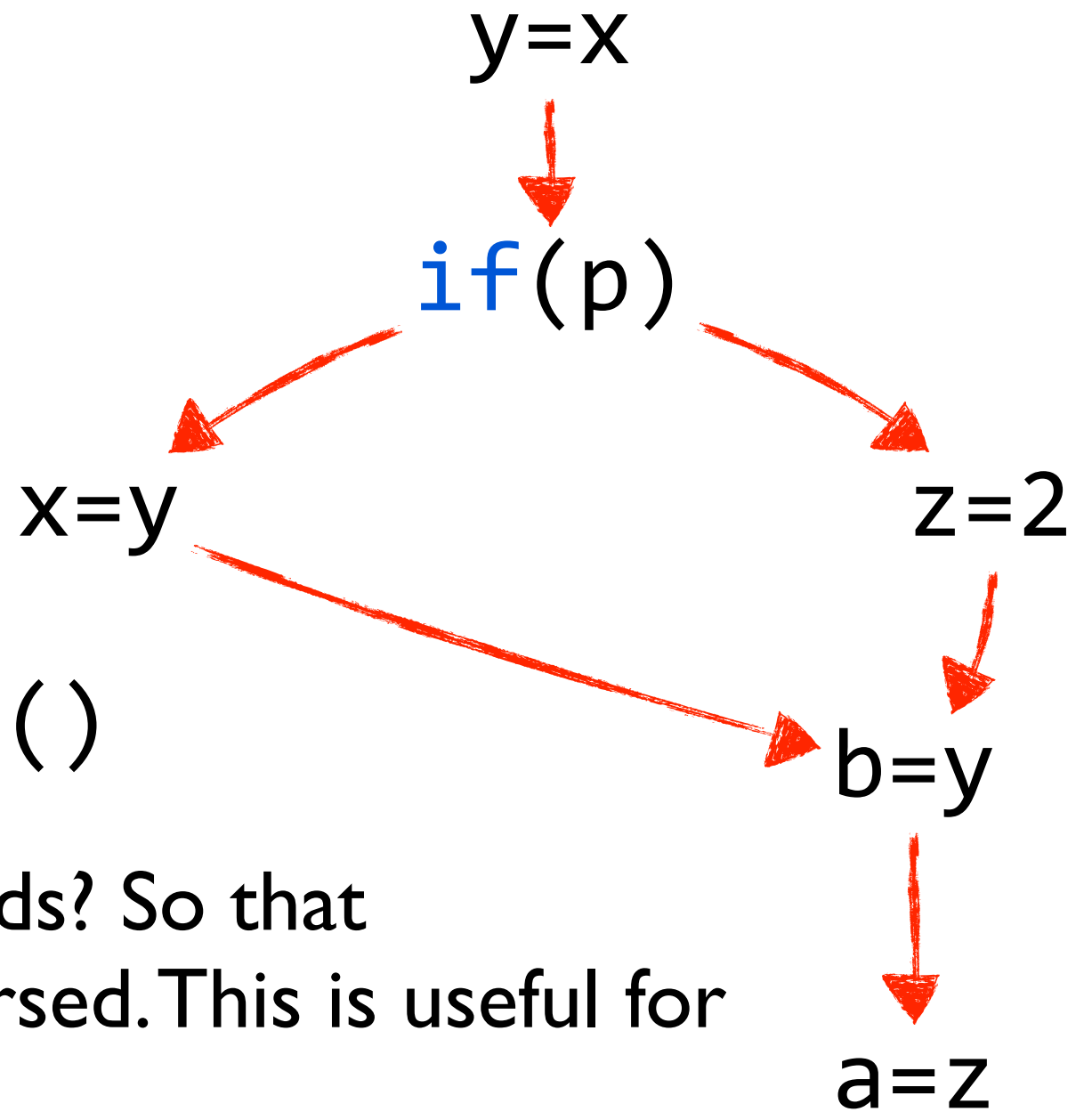
Getting a CFG...



better: new ExceptionalUnitGraph
(models exceptional flow as well)

Main operations on UnitGraph

- `getBody()`
- `getHeads()`, `getTails()`
 - Why allowing for multiple heads? So that UnitGraphs can easily be reversed. This is useful for backward analyses.
- `getPredsOf(u)`, `getSuccsOf(u)`



Summary

- Intermediate representations can abstract from concrete input languages
- Jimple is an intermediate language in three-address code format
 - most things are explicit
 - every statement is atomic, no nesting
 - simplifies notation of flow functions